Spring Data for Apache Cassandra - Reference Documentation

David Webb, Matthew Adams, John Blum, Mark Paluch

Version 1.5.9.RELEASE, 2017-11-27

Table of Contents

Preface . . . 2
1. Knowing Spring . . . 3
2. Knowing NoSQL and Cassandra . . . 4
3. Requirements . . . 5
4. Additional Help Resources . . . 6
4.1. Support . . . 6
4.1.1. Community Forum . . . 6
4.1.2. Professional Support . . . 6
4.2. Following Development . . . 6
4.3. Project Metadata . . . 6
5. New & Noteworthy . . . 8
5.1. What’s new in Spring Data for Apache Cassandra 1.5 . . . 8
6. Dependencies . . . 9
6.1. Dependency management with Spring Boot . . . 10
6.2. Spring Framework . . . 10
7. Working with Spring Data Repositories . . . 11
7.1. Core concepts . . . 11
7.2. Query methods . . . 13
7.3. Defining repository interfaces . . . 14
7.3.1. Fine-tuning repository definition . . . 15
7.3.2. Using Repositories with multiple Spring Data modules . . . 15
7.4. Defining query methods . . . 18
7.4.1. Query lookup strategies . . . 19
7.4.2. Query creation . . . 19
7.4.3. Property expressions . . . 20
7.4.4. Special parameter handling . . . 21
7.4.5. Limiting query results . . . 22
7.4.6. Streaming query results . . . 22
7.4.7. Async query results . . . 23
7.5. Creating repository instances . . . 24
7.5.1. XML configuration . . . 24
7.5.2. JavaConfig . . . 25
7.5.3. Standalone usage . . . 25
7.6. Custom implementations for Spring Data repositories . . . 26
7.6.1. Adding custom behavior to single repositories . . . 26
7.6.2. Adding custom behavior to all repositories . . . 28
7.7. Publishing events from aggregate roots . . . 29
7.8. Spring Data extensions . . . 30

7.8.1. Querydsl Extension . . . 30
7.8.2. Web support . . . 31
7.8.3. Repository populators . . . 38
7.8.4. Legacy web support . . . 40
Reference Documentation . . . 43
8. Cassandra support . . . 44
8.1. Spring CQL and Spring Data for Apache Cassandra modules . . . 44
8.1.1. Choosing an approach for Cassandra database access . . . 45
8.2. Getting Started . . . 45
8.3. Examples Repository . . . 49
8.4. Connecting to Cassandra with Spring . . . 49
8.4.1. Registering a Session instance using Java based metadata . . . 49
8.4.2. XML Configuration . . . 52
8.5. Schema Management . . . 55
8.5.1. Keyspaces and Lifecycle scripts . . . 55
8.5.2. Tables and User-defined types . . . 58
8.6. Introduction to CassandraTemplate . . . 59
8.6.1. Instantiating CassandraTemplate . . . 60
8.7. Saving, Updating, and Removing Rows . . . 60
8.7.1. Working with Primary Keys . . . 61
8.7.2. Type mapping . . . 64
8.7.3. Methods for saving and inserting rows . . . 65
8.7.4. Updating rows in a CQL table . . . 66
8.7.5. Methods for removing rows . . . 66
8.7.6. Methods for truncating tables . . . 67
8.8. Querying CQL Tables . . . 67
8.9. Overriding default mapping with custom converters . . . 69
8.9.1. Saving using a registered Spring Converter . . . 69
8.9.2. Reading using a Spring Converter . . . 69
8.9.3. Registering Spring Converters with the CassandraConverter . . . 70
8.9.4. Converter disambiguation . . . 70
8.10. Executing Commands . . . 71
8.10.1. Methods for executing commands . . . 71
8.11. Exception Translation . . . 72
9. Cassandra repositories . . . 73
9.1. Introduction . . . 73
9.2. Usage . . . 73
9.3. Query methods . . . 75
9.3.1. Projections . . . 77
9.4. Miscellaneous . . . 82
9.4.1. CDI Integration . . . 83

10. Mapping . . . 86
10.1. Convention based Mapping . . . 86
10.2. Data mapping and type conversion . . . 86
10.2.1. Mapping Configuration . . . 88
10.3. Metadata based Mapping . . . 89
10.3.1. Mapping annotation overview . . . 89
10.3.2. Overriding Mapping with explicit Converters . . . 92
Appendix . . . 94
Appendix A: Namespace reference . . . 95
The <repositories /> element . . . 95
Appendix B: Populators namespace reference . . . 96
The <populator /> element . . . 96
Appendix C: Repository query keywords . . . 97
Supported query keywords . . . 97
Appendix D: Repository query return types . . . 98
Supported query return types . . . 98

© 2008-2016 The original author(s).

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

Preface

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

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

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

Chapter 1. Knowing Spring

Spring Data uses the Spring Framework's core functionality, such as the IoC container, validation, type conversion and data binding, expression language, AOP, JMX integration, DAO support, and specifically the 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 no matter what 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 for Apache 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.

Chapter 2. 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 worse, 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 Spring Data for Apache Cassandra. The best way to get acquainted with Cassandra is to read the documentation and follow the examples. It usually doesn't take more then 5-10 minutes to go through them and if you are coming from a RDBMS background, many times these exercises can be an eye opener.

The starting ground for learning about Cassandra is cassandra.apache.org. Also, here is a list of other useful resources:

- Planet Cassandra site has many valuable resources for Cassandra best practices.
- The DataStax site offers commercial support and many resources, including, but not limited to, documentation, DataStax Academy, a Tech Blog and so on.
- The Cassandra Quick Start provides a convenient way to interact with a Apache Cassandra instance in combination with the online shell.

Chapter 3. Requirements

Spring Data for Apache Cassandra 1.x binaries require JDK level 6.0 and above, and Spring Framework 4.3.13.RELEASE and above.

In terms of Cassandra at least 2.0.

Chapter 4. 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 for Apache Cassandra module. However, if you encounter issues or you are just looking for an advice, feel free to use one of the links below:

4.1. Support

There are a few support options available:

4.1.1. Community Forum

Spring Data on Stackoverflow is a tag for all Spring Data (not just Cassandra) users to share information and help each other. Note that registration is needed **only** for posting.

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

- spring-data
- spring-data-cassandra

4.1.2. Professional Support

Professional, from-the-source support, with guaranteed response time, is available from Pivotal Sofware, Inc., the company behind Spring Data and Spring.

4.2. Following Development

For information on the Spring Data for Apache Cassandra source code repository, nightly builds and snapshot artifacts please see the Spring Data for Apache Cassandra homepage. You can help make Spring Data best serve the needs of the Spring community by interacting with developers through the Community on Stackoverflow. To follow developer activity look for the mailing list information on the Spring Data for Apache Cassandra homepage. If you encounter a bug or want to suggest an improvement, please create a ticket on the Spring Data issue tracker. To stay up to date with the latest news and announcements in the Spring eco system, subscribe to the Spring Community Portal. Lastly, you can follow the Spring blog or the project team on Twitter (SpringData).

4.3. Project Metadata

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

- Snapshot repository - https://repo.spring.io/libs-snapshot

Chapter 5. New & Noteworthy

5.1. What’s new in Spring Data for Apache Cassandra 1.5

- Assert compatibility with Cassandra 3.0 and Cassandra Java Driver 3.0.
- Configurable `ProtocolVersion` and `QueryOptions` on `Cluster` level.
- Support for `Optional` as query method result and argument.
- Declarative query methods using query derivation
- Support for User-Defined types and mapped User-Defined types using `@UserDefinedType`.
- The following annotations have been enabled to build own, composed annotations: `@Table`, `@UserDefinedType`, `@PrimaryKey`, `@PrimaryKeyClass`, `@PrimaryKeyColumn`, `@Column`, `@Query`, `@CassandraType`.

Chapter 6. Dependencies

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

Example 1. Using the Spring Data release train BOM

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

The current release train version is `Ingalls-SR9`. The train names are ascending alphabetically and currently available ones are listed here. The version name follows the following pattern: `${name}-${release}` where release can be one of the following:

- `BUILD-SNAPSHOT` - current snapshots
- `M1`, `M2` etc. - milestones
- `RC1`, `RC2` etc. - release candidates
- `RELEASE` - GA release
- `SR1`, `SR2` etc. - service releases

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

Example 2. Declaring a dependency to a Spring Data module

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

6.1. Dependency management with Spring Boot

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

6.2. Spring Framework

The current version of Spring Data modules require Spring Framework in version 4.3.13.RELEASE or better. The modules might also work with an older bugfix version of that minor version. However, using the most recent version within that generation is highly recommended.

Chapter 7. 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.

7.1. Core concepts

The central interface in Spring Data repository abstraction is `Repository` (probably not that much of a surprise). It takes the domain class to manage as well as the id type of the domain class as type arguments. This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one. The `CrudRepository` provides sophisticated CRUD functionality for the entity class that is being managed.

Example 3. CrudRepository interface

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

    <S extends T> S save(S entity); ①

    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 4. PagingAndSortingRepository

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

  Iterable<T> findAll(Sort sort);

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

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

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

In addition to query methods, query derivation for both count and delete queries, is available.

Example 5. Derived Count Query

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

  Long countByLastname(String lastname);
}
```

Example 6. Derived Delete Query

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

  Long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);

}
```

7.2. 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.

7.3. 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`.

7.3.1. Fine-tuning repository definition

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

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

Example 7. Selectively exposing CRUD methods

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

  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.

7.3.2. 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 8. 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 9. 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 10. 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 11. 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 12. Annotation-driven configuration of base packages

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

7.4. 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.

7.4.1. 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.

7.4.2. 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 13. 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.

7.4.3. 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).

7.4.4. 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 14. 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.

7.4.5. 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 15. 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.

7.4.6. 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 16. 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 17. 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.

7.4.7. 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.

7.5. 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.

7.5.1. 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 18. Enabling Spring Data repositories via XML

```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 19. 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.

7.5.2. 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 20. 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.

7.5.3. 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 21. Standalone usage of repository factory

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

7.6. 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.

7.6.1. 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 22. Interface for custom repository functionality

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

Example 23. 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 24. 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 25. 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 26. Manual wiring of custom implementations

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

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

7.6.2. 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 27. 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 28. 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 29. 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 30. Configuring a custom repository base class using XML

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

7.7. Publishing events from aggregate roots

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

Example 31. Exposing domain events from an aggregate root

```
class AnAggregateRoot {

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

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

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

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

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

7.8. Spring Data extensions

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

7.8.1. Querydsl Extension

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

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

Example 32. 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 33. 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);
```

7.8.2. Web support

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

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

Example 34. 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 35. 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 36. 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 37. 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 `@PageableDefault` annotation on the `Pageable` parameter.

Hypermedia support for Pageables

Spring HATEOAS ships with a representation model class `PagedResources` that allows enriching the content of a `Page` instance with the necessary `Page` metadata as well as links to let the clients easily navigate the pages. The conversion of a Page to a `PagedResources` is done by an implementation of the Spring HATEOAS `ResourceAssembler` interface, the `PagedResourcesAssembler`.

Example 38. 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.

7.8.3. Repository populators

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

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

Example 39. 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 40. Declaring a Jackson repository populator

```xml
<?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 41. 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>
```

7.8.4. Legacy web support

Domain class web binding for Spring MVC

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

```
@Controller
@RequestMapping("/users")
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 for Apache Cassandra.

Cassandra support introduces the Cassandra module feature set.

Cassandra repositories introduces the repository support for Cassandra.

Chapter 8. Cassandra support

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

- Spring configuration support using Java-based `@Configuration` classes or the XML namespace to create a Cassandra instance with replica sets using the driver.
- CassandraTemplate helper class that increases productivity by handling common Cassandra operations properly. 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 data oriented tasks you will use the `CassandraTemplate` or the `Repository` support, which leverage the rich mapping functionality. `CassandraTemplate` is commonly used to increment counters or perform ad-hoc CRUD operations. `CassandraTemplate` also provides callback methods making it easy to get a hold of low-level API objects such as `com.datastax.driver.core.Session` allowing you to communicate directly with Cassandra. Spring Data for Apache Cassandra uses consistent naming conventions on objects in various APIs to those found in the DataStax Java Driver so that they are familiar and so you can map your existing knowledge onto the Spring APIs.

8.1. Spring CQL and Spring Data for Apache Cassandra modules

Spring Data for Apache Cassandra comes with two modules: Spring CQL and Spring Data Cassandra.

The value-add provided by the Spring Data Cassandra abstraction is perhaps best shown by the sequence of actions outlined in the table below. The table shows what actions Spring will take care of and which actions are the responsibility of you, the application developer.

Table 2. Spring CQL - who does what?

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the CQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	

Action	Spring	You
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Close the Session.	X	

Spring CQL takes care of all the low-level details that can make Cassandra and CQL such a tedious API to develop with. Spring Data Cassandra adds schema generation, object mapping and Repository support.

8.1.1. Choosing an approach for Cassandra database access

You can choose among several approaches to form the basis for your Cassandra database access. Spring's support for Apache Cassandra comes in different flavors. Once you start using one of these approaches, you can still mix and match to include a feature from a different approach.

- *CqlTemplate* is the classic Spring CQL approach and the most popular. This is the "lowest level" approach and all others use a `CqlTemplate` under the covers.
- *CassandraTemplate* wraps a `CqlTemplate` to provide query result to object mapping and the use of SELECT, INSERT, UPDATE and DELETE methods instead of writing CQL statements. This approach provides better documentation and ease of use.
- *Repository Abstraction* allows you to create Repository declarations in your data access layer. The goal of Spring Data's Repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.

8.2. Getting Started

Spring Apache Cassandra support requires Apache Cassandra 2.1 or higher, Datastax Java Driver 3.0 or higher and Java SE 6 or higher. 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 Apache Cassandra server. Refer to the Apache Cassandra Quick Start guide for an explanation on how to startup Apache Cassandra. Once installed starting Cassandra is typically a matter of executing the following command: `CASSANDRA_HOME/bin/cassandra -f`

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 dependencies omitted -->

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-cassandra</artifactId>
    <version>1.5.9.RELEASE</version>
  </dependency>

</dependencies>
```

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

```
<spring.framework.version>4.3.13.RELEASE</spring.framework.version>
```

If using a milestone release instead of a GA release, 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.

You can also browse the Spring repositories here.

Now we will create a simple Java application that stores and reads a domain object to/from Cassandra.

First, create a simple domain object class to persist.

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

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

@Table
public class Person {

  @PrimaryKey
  private final String id;

  private final String name;
  private final 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 String.format("{ @type = %1$s, id = %2$s, name = %3$s, age = %4$d }",
      getClass().getName(), getId(), getName(), getAge());
  }
}
```

Next, create the main application to run.

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

import java.io.Closeable;
import java.util.UUID;

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 CassandraApplication {

  private static final Logger LOGGER = LoggerFactory.getLogger(CassandraApplication
.class);

  protected static Person newPerson(String name, int age) {
    return newPerson(UUID.randomUUID().toString(), name, age);
  }

  protected static Person newPerson(String id, String name, int age) {
    return new Person(id, name, age);
  }

  public static void main(String[] args) {

    Cluster cluster = Cluster.builder().addContactPoints("localhost").build();
    Session session = cluster.connect("mykeyspace");

    CassandraOperations template = new CassandraTemplate(session);

    Person jonDoe = template.insert(newPerson("Jon Doe", 40));

    Select selectStatement = QueryBuilder.select().from("person");
    selectStatement.where(QueryBuilder.eq("id", jonDoe.getId()));

    LOGGER.info(template.queryForObject(selectStatement, Person.class).getId());

    template.truncate("person");
    session.close();
    cluster.close();
  }
}
```

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 a `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 either use a CQL String or the DataStax `QueryBuilder` API to construct you queries.

8.3. Examples Repository

There is a Github repository with several examples that you can download and play around with to get a feel for how the library works.

8.4. Connecting to Cassandra with Spring

One of the first tasks when using Apache Cassandra and Spring is to create a `com.datastax.driver.core.Session` object using the Spring IoC container. There are two main ways to do this, either using Java-based bean metadata or XML-based bean metadata. These are discussed in the following sections.

NOTE For those not familiar with how to configure the Spring container using Java-based bean metadata instead of XML-based metadata, see the high-level introduction in the reference docs here as well as the detailed documentation here.

8.4.1. Registering a Session instance using Java based metadata

An example of using Java-based bean metadata to register an instance of a `com.datastax.driver.core.Session` is shown below.

Example 42. Registering a com.datastax.driver.core.Session object using Java based bean metadata

```
@Configuration
public class AppConfig {

  /*
   * Use the standard Cassandra driver API to create a
com.datastax.driver.core.Session instance.
   */
  public @Bean Session session() {
    Cluster cluster = Cluster.builder().addContactPoints("localhost").build();
    return cluster.connect("mykeyspace");
  }
}
```

This approach allows you to use the standard `com.datastax.driver.core.Session` API that you may already be used to using.

An alternative is to register an instance of `com.datastax.driver.core.Session` instance with the container using Spring's `CassandraCqlSessionFactoryBean` and `CassandraCqlClusterFactoryBean`. As compared to instantiating a `com.datastax.driver.core.Session` instance directly, the `FactoryBean` approach has the added advantage of also providing the container with an `ExceptionTranslator` implementation that translates Cassandra exceptions to exceptions in Spring's portable `DataAccessException` hierarchy for data access classes annotated. This hierarchy and use of `@Repository` is described in Spring's DAO support features.

An example of a Java-based bean metadata that supports exception translation on `@Repository` annotated classes is shown below:

Example 43. Registering a com.datastax.driver.core.Session object using Spring's CassandraCqlSessionFactoryBean and enabling Spring's exception translation support

```
@Configuration
public class AppConfig {

  /*
   * Factory bean that creates the com.datastax.driver.core.Session instance
   */
  public @Bean CassandraCqlClusterFactoryBean cluster() {

    CassandraCqlClusterFactoryBean cluster = new CassandraCqlClusterFactoryBean();
    cluster.setContactPoints("localhost");

    return cluster;
  }

  /*
   * Factory bean that creates the com.datastax.driver.core.Session instance
   */
   public @Bean CassandraCqlSessionFactoryBean session() {

    CassandraCqlSessionFactoryBean session = new CassandraCqlSessionFactoryBean();
    session.setCluster(cluster().getObject());
    session.setKeyspaceName("mykeyspace");

    return session;
  }
}
```

Using `CassandraTemplate` with object mapping and Repository support requires a `CassandraTemplate`, `CassandraMappingContext`, `CassandraConverter` and enabling Repository support.

Example 44. Registering components to configure object mapping and repository support

```
@Configuration
@EnableCassandraRepositories(basePackages = { "org.spring.cassandra.example.repo"
})
public class CassandraConfig {

  @Bean
  public CassandraClusterFactoryBean cluster() {

    CassandraClusterFactoryBean cluster = new CassandraClusterFactoryBean();
    cluster.setContactPoints("localhost");

    return cluster;
  }

  @Bean
  public CassandraMappingContext mappingContext() {

    BasicCassandraMappingContext mappingContext =  new
BasicCassandraMappingContext();
    mappingContext.setUserTypeResolver(new SimpleUserTypeResolver(cluster()
.getObject(), "mykeyspace"));

    return mappingContext;
  }

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

  @Bean
  public CassandraSessionFactoryBean session() throws Exception {

    CassandraSessionFactoryBean session = new CassandraSessionFactoryBean();
    session.setCluster(cluster().getObject());
    session.setKeyspaceName("mykeyspace");
    session.setConverter(converter());
    session.setSchemaAction(SchemaAction.NONE);

    return session;
  }

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

Creating configuration classes registering Spring Data for Apache Cassandra components can be an exhausting challenge so Spring Data for Apache Cassandra comes with a prebuilt configuration support class. Classes extending from `AbstractCassandraConfiguration` will register beans for Spring Data for Apache Cassandra use. `AbstractCassandraConfiguration` lets you provide various configuration options such as initial entities, default query options, pooling options, socket options and much more. `AbstractCassandraConfiguration` will support you also with schema generation based on initial entities, if any are provided. Extending from `AbstractCassandraConfiguration` requires you to at least provide the Keyspace name by implementing the `getKeyspaceName` method.

Example 45. Registering Spring Data for Apache Cassandra beans using AbstractCassandraConfiguration

```
@Configuration
public class AppConfig extends AbstractCassandraConfiguration {

  /*
   * Provide a contact point to the configuration.
   */
  public String getContactPoints() {
    return "localhost";
  }

  /*
   * Provide a keyspace name to the configuration.
   */
  public getKeyspaceName() {
    return "mykeyspace";
  }
}
```

8.4.2. XML Configuration

Externalize Connection Properties

Create a properties file containing the information needed to connect to Cassandra. `contactpoints` and `keyspace` are required fields; `port` has been added for clarity.

We will call this properties file, `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.

Registering a Session instance using XML based metadata

While you can use Spring's traditional `<beans/>` XML namespace to register an instance of com.datastax.driver.core.Session with the container, the XML can be quite verbose as it is general purpose. XML namespaces are a better alternative to configuring commonly used objects such as the Session instance. The cql and cassandra namespaces allow you to create a Session instance.

To use the Cassandra namespace elements you will need to reference the Cassandra schema:

Example 46. XML schema to configure Cassandra using the cql namespace

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

  <!-- Default bean name is 'cassandraCluster' -->
  <cql:cluster contact-points="localhost" port="9042">
    <cql:keyspace action="CREATE_DROP" name="mykeyspace" />
  </cql:cluster>

  <!-- Default bean name is 'cassandraSession' -->
  <cql:session keyspace-name="mykeyspace" />

</beans>
```

Example 47. XML schema to configure Cassandra using the `cassandra` namespace

```xml
<?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:cassandra="http://www.springframework.org/schema/data/cassandra"
  xsi:schemaLocation="
    http://www.springframework.org/schema/data/cassandra
    http://www.springframework.org/schema/data/cassandra/spring-cassandra.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <!-- Default bean name is 'cassandraCluster' -->
  <cassandra:cluster contact-points="localhost" port="9042">
    <cassandra:keyspace action="CREATE_DROP" name="mykeyspace" />
  </cassandra:cluster>

  <!-- Default bean name is 'cassandraSession' -->
  <cassandra:session keyspace-name="${cassandra.keyspace}" schema-action="NONE" />

</beans>
```

NOTE You may have noticed the slight difference between namespaces: `cql` and `cassandra`. Using the `cql` namespace is limited to low-level CQL support while `cassandra` extends the `cql` namespace with object mapping and schema generation support.

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

While this example shows 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 for Apache Cassandra configuration. This is including, but not limited to Authentication, Load Balancing Policies, Retry Policies and Pooling Options. All of the Spring Data for Apache 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.

Example 48. Configuring Spring Data Components via XML

```xml
<!-- 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>
  <cassandra:user-type-resolver keyspace-name="${cassandra.keyspace}" />
</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 for Apache Cassandra Repositories, add
your base packages to scan here -->
<cassandra:repositories base-package="org.spring.cassandra.example.repo" />
```

8.5. Schema Management

Apache Cassandra is a data store that requires a schema definition prior to any data interaction. Spring Data for Apache Cassandra can support you with this task.

8.5.1. Keyspaces and Lifecycle scripts

The very first thing to start with is a Cassandra Keyspace. A Keyspace is a logical grouping of tables that share the same replication factor and replication strategy. Keyspace management is located in the `Cluster` configuration, which has the notion of `KeyspaceSpecification` and startup/shutdown CQL script execution.

Declaring a Keyspace with a specification allows creating/dropping of the Keyspace. It will derive CQL from the specification so you’re not required to write CQL yourself.

Example 49. Specifying a Cassandra Keyspace via XML

```
<cql:cluster>

    <cql:keyspace action="CREATE_DROP" durable-writes="true" name="my_keyspace">

    <cql:replication class="NETWORK_TOPOLOGY_STRATEGY">
      <cql:data-center name="foo" replication-factor="1" />
      <cql:data-center name="bar" replication-factor="2" />
    </cql:replication>
  </cql:keyspace>

</cql:cluster>
```

Example 50. Specifying a Cassandra Keyspace via JavaConfig

```
@Configuration
public abstract class AbstractCassandraConfiguration extends
AbstractClusterConfiguration
        implements BeanClassLoaderAware {

  @Override
  protected List<CreateKeyspaceSpecification> getKeyspaceCreations() {

    CreateKeyspaceSpecification specification = CreateKeyspaceSpecification
.createKeyspace("my_keyspace")
      .with(KeyspaceOption.DURABLE_WRITES, true)
      .withNetworkReplication(DataCenterReplication.dcr("foo", 1),
DataCenterReplication.dcr("bar", 2));

    return Arrays.asList(specification);
  }

  @Override
  protected List<DropKeyspaceSpecification> getKeyspaceDrops() {
    return Arrays.asList(DropKeyspaceSpecification.dropKeyspace("my_keyspace"));
  }

  // ...
}
```

Startup/shutdown CQL execution follows a slightly different approach that is bound to the `Cluster` lifecycle. You can provide arbitrary CQL that is executed on `Cluster` initialization and shutdown in the `SYSTEM` keyspace.

Example 51. Specifying Startup/Shutdown scripts via XML

```
<cql:cluster>
  <cql:startup-cql><![CDATA[
CREATE KEYSPACE IF NOT EXISTS my_other_keyspace WITH durable_writes = true AND
replication = { 'replication_factor' : 1, 'class' : 'SimpleStrategy' };
    ]]></cql:startup-cql>
  <cql:shutdown-cql><![CDATA[
DROP KEYSPACE my_other_keyspace;
    ]]></cql:shutdown-cql>
</cql:cluster>
```

Example 52. Specifying a Startup/Shutdown scripts via JavaConfig

```
@Configuration
public class CassandraConfiguration extends AbstractCassandraConfiguration {

  @Override
  protected List<String> getStartupScripts() {

    String script = "CREATE KEYSPACE IF NOT EXISTS my_other_keyspace "
      + "WITH durable_writes = true "
      + "AND replication = { 'replication_factor' : 1, 'class' : 'SimpleStrategy'
};";

    return Arrays.asList(script);
  }

  @Override
  protected List<String> getShutdownScripts() {
    return Arrays.asList("DROP KEYSPACE my_other_keyspace;");
  }

  // ...
}
```

NOTE `KeyspaceSpecifications` and lifecycle CQL scripts are available with the `cql` and `cassandra` namespaces.

NOTE Keyspace creation allows rapid bootstrapping without the need of external Keyspace management. This can be useful for certain scenarios but should be used with care. Dropping a Keyspace on application shutdown will remove the Keyspace and all data stored inside the tables.

8.5.2. Tables and User-defined types

Spring Data for Apache Cassandra's approaches data access with mapped entity classes that fit your data model. These entity classes can be used to create Cassandra table specifications and user type definitions.

Schema creation is tied to `Session` initialization with `SchemaAction`. Following actions are supported:

- `SchemaAction.NONE`: No tables/types will be created or dropped. This is the default setting.
- `SchemaAction.CREATE`: Create tables and user-defined types from entities annotated with `@Table` and types annotated with `@UserDefinedType`. Existing tables/types will cause an error if the type is attempted to be created.
- `SchemaAction.CREATE_IF_NOT_EXISTS`: Like `SchemaAction.CREATE` but with `IF NOT EXISTS` applied. Existing tables/types won't cause any errors but may remain stale.
- `SchemaAction.RECREATE`: Drops and recreate existing tables and types that are known to be used. Tables and types that are not configured in the application are not dropped.
- `SchemaAction.RECREATE_DROP_UNUSED`: Drop all tables and types and recreate only known tables and types.

NOTE `SchemaAction.RECREATE`/`SchemaAction.RECREATE_DROP_UNUSED` will drop your tables and you will experience data loss. `RECREATE_DROP_UNUSED` also drops tables and types that are not know to the application.

Enabling Tables and User-Defined Types for Schema Management

Metadata based Mapping explains object mapping using conventions and annotations. Schema management is only active for entities annotated with `@Table` and user-defined types annotated with `@UserDefinedType` to prevent unwanted classes from being created as table/type. Entities are discovered by scanning the class path. Entity scanning requires one or more base packages.

Example 53. Specifying Entity Base Packages via XML

```xml
<cassandra:mapping entity-base-packages="com.foo,com.bar"/>
```

Example 54. Specifying Entity Base Packages via JavaConfig

```
@Configuration
public class CassandraConfiguration extends AbstractCassandraConfiguration {

    @Override
    public String[] getEntityBasePackages() {
        return new String[] { "com.foo", "com.bar" };
    }

    // ...
}
```

8.6. Introduction to CassandraTemplate

The `CassandraTemplate` class, located in the package `org.springframework.data.cassandra`, is the central class in Spring's Cassandra support providing a rich feature set to interact with the database. The template offers convenience operations to create, update, delete and query Cassandra and provides a mapping between your domain objects and Cassandra rows.

NOTE Once configured, `CassandraTemplate` is Thread-safe and can be reused across multiple instances.

The mapping between Cassandra rows and domain classes is done by delegating to an implementation of the `CassandraConverter` interface. Spring provides a default implementation, `MappingCassandraConverter`, but you can also write your own converter. Please refer to the section on Cassandra conversion for more detailed information.

The `CassandraTemplate` class implements the interface `CassandraOperations`. In as much as possible, the methods on `CassandraOperations` are named after methods available with Cassandra to make the API familiar to existing Cassandra developers who are familiar with Cassandra. For example, you will find methods such as "select", "insert", "delete", and "update". The design goal was to make it as easy as possible to transition between the use of the base Cassandra driver and `CassandraOperations`. A major difference in between the two APIs is that `CassandraOperations` can be passed domain objects instead of CQL and query objects.

NOTE The preferred way to reference operations on a `CassandraTemplate` instance is via its interface, `CassandraOperations`.

The default converter implementation used by `CassandraTemplate` is `MappingCassandraConverter`. While the `MappingCassandraConverter` can make use of additional metadata to specify the mapping of objects to rows it is also capable of converting objects that contain no additional metadata by using some conventions for the mapping of fields and table names. These conventions as well as the use of mapping annotations is explained in the Mapping chapter.

Another central feature of `CassandraTemplate` is exception translation of exceptions thrown in the

Cassandra Java driver into Spring's portable Data Access Exception hierarchy. Refer to the section on exception translation for more information.

Now let's look at a examples of how to work with the `CassandraTemplate` in the context of the Spring container.

8.6.1. 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), then 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);
```

8.7. 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 tables managed in Cassandra.

8.7.1. Working with Primary Keys

Cassandra requires at least one partition key field for a CQL Table. A table can declare additionally one or more clustering key fields. When your CQL Table has a composite primary key, you must create a `@PrimaryKeyClass` to define the structure of the composite primary key. In this context, composite primary key means one or more partition columns optionally combined with one or more clustering columns.

Primary keys can make use of any singular simple Cassandra type or mapped User-Defined Type. Collection-typed primary keys are not supported.

Simple Primary Key

A simple primary key consists of one partition key field within an entity class. Since it's one field only, we safely can assume it's a partition key.

Example 55. CQL Table defined in Cassandra

```
CREATE TABLE user (
  user_id text,
  firstname text,
  lastname text,
  PRIMARY KEY (user_id))
;
```

Example 56. Annotated Entity

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

  @PrimaryKey("user_id")
  private String userId;

  private String firstname;
  private String lastname;

  // getters and setters omitted for brevity

}
```

Composite Key

Composite primary keys (or compound keys) consist of more than one primary key fields. That said, a composite primary key can consist of multiple partition keys, a partition key and a clustering key, or a multitude of primary key fields.

Composite keys can be represented in two ways with Spring Data for Apache Cassandra:

1. Embedded in an entity.
2. By using `@PrimaryKeyClass`.

The simplest form 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.

Example 57. CQL Table with a Composite Primary Key

```
CREATE TABLE login_event(
  person_id text,
  event_code int,
  event_time timestamp,
  ip_address text,
  PRIMARY KEY (person_id, event_code, event_time))
  WITH CLUSTERING ORDER BY (event_time DESC)
;
```

Flat Composite Primary Key

Flat composite primary keys are embedded inside the entity as flat fields. Primary key fields are annotated with `@PrimaryKeyColumn` along with other fields in the entity. Selection requires either a query to contain predicates for the individual fields or the use of `MapId`.

Example 58. Using a flat Composite Primary Key

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

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

  @PrimaryKeyColumn(name = "event_code", ordinal = 1, type = PrimaryKeyType
.PARTITIONED)
  private int eventCode;

  @PrimaryKeyColumn(name = "event_time", ordinal = 2, type = PrimaryKeyType
.CLUSTERED, ordering = Ordering.DESCENDING)
  private Date eventTime;

  @Column("ip_address)
  private String ipAddress;

  // getters and setters omitted for brevity
}
```

Primary Key Class

A primary key class is a composite primary key class that is mapped to multiple fields or properties of the entity. It's annotated with `@PrimaryKeyClass` and defines `equals` and `hashCode` methods. The semantics of value equality for these methods should be consistent with the database equality for the database types to which the key is mapped. Primary key classes can be used with Repositories (as the Id type) and to represent an entities' identity in a single complex object.

Example 59. Composite Primary Key Class

```
@PrimaryKeyClass
public class LoginEventKey implements Serializable {

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

  @PrimaryKeyColumn(name = "event_code", ordinal = 1, type = PrimaryKeyType
.PARTITIONED)
  private int eventCode;

  @PrimaryKeyColumn(name = "event_time", ordinal = 2, type = PrimaryKeyType
.CLUSTERED, ordering = Ordering.DESCENDING)
  private Date eventTime;

  // other methods omitted for brevity
}
```

Example 60. Using a Composite Primary Key

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

  @PrimaryKey
  private LoginEventKey key;

  @Column("ip_address)
  private String ipAddress;

  // getters and setters omitted for brevity
}
```

NOTE `PrimaryKeyClass` must implement `Serializable` and should provide implementations of `hashCode()` and `equals()`.

8.7.2. Type mapping

Spring Data for Apache Cassandra relies on the DataStax Java Driver's `CodecRegistry` to ensure type support. As types are added or changed, the Spring Data for Apache Cassandra module will continue to function without requiring changes. See CQL data types and Data mapping and type conversion for the current type mapping matrix.

8.7.3. 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 low-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 set 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);
```

8.7.4. 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);
```

8.7.5. 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);
```

8.7.6. 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);
```

8.8. Querying CQL Tables

There 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()));
}
```

8.9. 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 `MappingCassandraConverter`.

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

8.9.1. Saving using a registered Spring Converter

An example implementation of the `Converter` that converts a `Person` object to a `java.lang.String` using Jackson 2 is shown below:

```
import org.springframework.core.convert.converter.Converter;

import org.springframework.util.StringUtils;
import org.codehaus.jackson.map.ObjectMapper;

static class PersonWriteConverter implements Converter<Person, String> {

  public String convert(Person source) {

    try {
      return new ObjectMapper().writeValueAsString(source);
    } catch (IOException e) {
      throw new IllegalStateException(e);
    }
  }
}
```

8.9.2. Reading using a Spring Converter

An example implementation of the `Converter` that converts a `java.lang.String` into a `Person` object using Jackson 2 is shown below:

```
import org.springframework.core.convert.converter.Converter;

import org.springframework.util.StringUtils;
import org.codehaus.jackson.map.ObjectMapper;

static class PersonReadConverter implements Converter<String, Person> {

  public Person convert(String source) {

   if (StringUtils.hasText(source)) {
     try {
       return new ObjectMapper().readValue(source, Person.class);
     } catch (IOException e) {
       throw new IllegalStateException(e);
     }
    }

    return null;
  }
}
```

8.9.3. Registering Spring Converters with the CassandraConverter

The Spring Data for Apache Cassandra Java Config provides a convenient way to register Spring `Converter`s with the `MappingCassandraConverter`. The configuration snippet below shows how to manually register converters as well as configuring the `CustomConversions`.

```
@Configuration
public static class Config extends AbstractCassandraConfiguration {

  @Override
  public CustomConversions customConversions() {

    List<Converter<?, ?>> converters = new ArrayList<Converter<?, ?>>();
    converters.add(new PersonReadConverter());
    converters.add(new PersonWriteConverter());

    return new CustomConversions(converters);
  }

  // other methods omitted...
}
```

8.9.4. Converter disambiguation

Generally, we inspect the `Converter` implementations for both source and target types they convert from and to. Depending on whether one of those is a type Cassandra can handle natively, Spring Data will register the `Converter` instance as a reading or writing one. Have a look at the following samples:

```
// Write converter as only the target type is one cassandra can handle natively
class MyConverter implements Converter<Person, String> { … }

// Read converter as only the source type is one cassandra can handle natively
class MyConverter implements Converter<String, Person> { … }
```

In case you write a `Converter` whose source and target type are native Cassandra types there's no way for Spring Data to determine whether we should consider it as reading or writing `Converter`. Registering the `Converter` instance as both might lead to unwanted results.

E.g. a `Converter<String, Long>` is ambiguous although it probably does not make sense to try to convert all `String` instances into `Long` instances when writing. To be generally able to force the infrastructure to register a `Converter` for one way only we provide `@ReadingConverter` as well as `@WritingConverter` to be used as the appropriate `Converter` implementation.

8.10. Executing Commands

8.10.1. Methods for executing commands

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

This example uses the basic `AsynchronousQueryListener` that comes with Spring Data for Apache 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);
```

8.11. 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 Apache Cassandra extends this feature to Apache Cassandra 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 9. Cassandra repositories

9.1. Introduction

This chapter covers the details of the Spring Data Repository support for Apache Cassandra. Cassandra's Repository support builds on the core Repository support explained in Working with Spring Data Repositories. So make sure you understand of the basic concepts explained there before proceeding.

9.2. Usage

To access domain entities stored in Apache Cassandra, you can leverage Spring Data's sophisticated Repository support that eases implementing DAOs quite significantly. To do so, simply create an interface for your Repository:

Example 61. Sample Person entity

```
@Table
public class Person {

  @Id
  private String id;
  private String firstname;
  private String lastname;

  // … getters and setters omitted
}
```

We have a simple domain object here. Note that the entity has a property named `id` of type `String`. The default serialization mechanism used in `CassandraTemplate` (which is backing the Repository support) regards properties named id as row id.

Example 62. Basic Repository interface to persist Person entities

```
public interface PersonRepository extends CrudRepository<Person, String> {

  // additional custom finder methods go here
}
```

Right now this interface simply serves typing purposes, but we will add additional methods to it later. In your Spring configuration simply add:

Example 63. General Cassandra repository Spring configuration

```xml
<?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:cassandra="http://www.springframework.org/schema/data/cassandra"
  xsi:schemaLocation="
    http://www.springframework.org/schema/data/cassandra
    http://www.springframework.org/schema/data/cassandra/spring-cassandra.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <cassandra:cluster port="9042"/>
    <cassandra:session keyspace-name="keyspaceName"/>

    <cassandra:mapping
            entity-base-packages="com.acme.*.entities">
    </cassandra:mapping>

    <cassandra:converter/>

    <cassandra:template/>

    <cassandra:repositories base-package="com.acme.*.entities"/>
</beans>
```

The `cassandra:repositories` namespace element will cause the base packages to be scanned for interfaces extending `CrudRepository` and create Spring beans for each one found. By default, the Repositories will be wired with a `CassandraTemplate` Spring bean called `cassandraTemplate`, so you only need to configure `cassandra-template-ref` explicitly if you deviate from this convention.

If you'd rather like to go with JavaConfig use the `@EnableCassandraRepositories` annotation. The annotation carries the same attributes as the namespace element. If no base package is configured the infrastructure will scan the package of the annotated configuration class.

Example 64. JavaConfig for repositories

```
@Configuration
@EnableCassandraRepositories
class ApplicationConfig extends AbstractCassandraConfiguration {

  @Override
  protected String getKeyspaceName() {
    return "keyspace";
  }

  public String[] getEntityBasePackages() {
    return new String[] { "com.oreilly.springdata.cassandra" };
  }
}
```

As our domain Repository extends `CrudRepository` it provides you with basic CRUD operations. Working with the Repository instance is just a matter of injecting the Repository as a dependency into a client.

Example 65. Paging access to Person entities

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class PersonRepositoryTests {

    @Autowired PersonRepository repository;

    @Test
    public void readsPersonTableCorrectly() {

      List<Person> persons = repository.findAll();
      assertThat(persons.isEmpty()).isFalse();
    }
}
```

The sample creates an application context with Spring's unit test support, which will perform annotation-based dependency injection into the test class. Inside the test cases (test methods) we simply use the Repository to query the data store. We invoke the Repository query method that requests the all `Person` instances.

9.3. Query methods

Most of the data access operations you usually trigger on a Repository result in a query being executed against the Apache Cassandra database. Defining such a query is just a matter of declaring a method on the Repository interface.

Example 66. PersonRepository with query methods

```
public interface PersonRepository extends CrudRepository<Person, String> {

    List<Person> findByLastname(String lastname);                    ①

    List<Person> findByFirstname(String firstname, Sort sort);       ②

    Person findByShippingAddress(Address address);                   ③

    Stream<Person> findAllBy();                                      ④
}
```

① The method shows a query for all people with the given `lastname`. The query will be derived from parsing the method name for constraints which can be concatenated with `And`. Thus the method name will result in a query expression of `SELECT * from person WHERE lastname = 'lastname'`.

② Applies dynamic sorting to a query. Just add a `Sort` parameter to your method signature and Spring Data will automatically apply ordering to the query accordingly.

③ Shows that you can query based on properties which are not a primitive type using registered `Converter’s in `CustomConversions`.

④ Uses a Java 8 `Stream` which reads and converts individual elements while iterating the stream.

NOTE Querying non-primary key properties requires secondary indexes.

Table 3. Supported keywords for query methods

Keyword	Sample	Logical result
`After`	`findByBirthdateAfter(Date date)`	`birthdate > date`
`GreaterThan`	`findByAgeGreaterThan(int age)`	`age > age`
`GreaterThanEqual`	`findByAgeGreaterThanEqual(int age)`	`age >= age`
`Before`	`findByBirthdateBefore(Date date)`	`birthdate < date`
`LessThan`	`findByAgeLessThan(int age)`	`age < age`
`LessThanEqual`	`findByAgeLessThanEqual(int age)`	`age ⇐ age`
`In`	`findByAgeIn(Collection ages)`	`age IN (ages…)`
`Like`, `StartingWith`, `EndingWith`	`findByFirstnameLike(String name)`	`firstname LIKE (name as like expression)`
`Containing` on String	`findByFirstnameContaining(String name)`	`firstname LIKE (name as like expression)`

Keyword	Sample	Logical result
`Containing` on Collection	`findByAddressesContaining(Address address)`	`addresses CONTAINING address`
`(No keyword)`	`findByFirstname(String name)`	`firstname = name`
`IsTrue`, `True`	`findByActiveIsTrue()`	`active = true`
`IsFalse`, `False`	`findByActiveIsFalse()`	`active = false`

9.3.1. Projections

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

Imagine a sample repository and aggregate root type like this:

Example 67. A sample aggregate and repository

```
class Person {

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

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

interface PersonRepository extends Repository<Person, UUID> {

  Collection<Person> findByLastname(String lastname);
}
```

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

Interface-based projections

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

Example 68. A projection interface to retrieve a subset of attributes

```
interface NamesOnly {

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

The important bit here is that the properties defined here exactly match properties in the aggregate root. This allows a query method to be added like this:

Example 69. A repository using an interface based projection with a query method

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

  Collection<NamesOnly> findByLastname(String lastname);
}
```

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

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

Example 70. A projection interface to retrieve a subset of attributes

```
interface PersonSummary {

  String getFirstname();
  String getLastname();
  AddressSummary getAddress();

  interface AddressSummary {
    String getCity();
  }
}
```

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

Closed projections

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

Example 71. A closed projection

```
interface NamesOnly {

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

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

Open projections

Accessor methods in projection interfaces can also be used to compute new values by using the `@Value` annotation on it:

Example 72. An Open Projection

```
interface NamesOnly {

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

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

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

Example 73. A projection interface using a default method for custom logic

```
interface NamesOnly {

  String getFirstname();
  String getLastname();

  default String getFullName() {
    return getFirstname.concat(" ").concat(getLastname());
  }
}
```

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

Example 74. Sample Person object

```
@Component
class MyBean {

  String getFullName(Person person) {
    …
  }
}

interface NamesOnly {

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

Note, how the SpEL expression refers to `myBean` and invokes the `getFullName(…)` method forwarding the projection target as method parameter. Methods backed by SpEL expression evaluation can also use method parameters which can then be referred to from the expression. The method parameters are available via an `Object` array named `args`.

Example 75. Sample Person object

```
interface NamesOnly {

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

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

Class-based projections (DTOs)

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

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

Example 76. A projecting DTO

```
class NamesOnly {

  private final String firstname, lastname;

  NamesOnly(String firstname, String lastname) {

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

  String getFirstname() {
    return this.firstname;
  }

  String getLastname() {
    return this.lastname;
  }

  // equals(…) and hashCode() implementations
}
```

TIP

Avoiding boilerplate code for projection DTOs

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

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

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

Dynamic projections

So far we have used the projection type as the return type or element type of a collection. However, it might be desirable to rather select the type to be used at invocation time. To apply dynamic projections, use a query method like this:

Example 77. A repository using a dynamic projection parameter

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

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

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

Example 78. Using a repository with dynamic projections

```
void someMethod(PersonRepository people) {

  Collection<Person> aggregates =
    people.findByLastname("Matthews", Person.class);

  Collection<NamesOnly> aggregates =
    people.findByLastname("Matthews", NamesOnly.class);
}
```

9.4. Miscellaneous

9.4.1. CDI Integration

Instances of the Repository interfaces are usually created by a container, and the Spring container is the most natural choice when working with Spring Data. Spring Data for Apache Cassandra ships with a custom CDI extension that allows using the repository abstraction in CDI environments. The extension is part of the JAR so all you need to do to activate it is dropping the Spring Data for Apache Cassandra JAR into your classpath. You can now set up the infrastructure by implementing a CDI Producer for the `CassandraTemplate`:

```
class CassandraTemplateProducer {

    @Produces
    @Singleton
    public Cluster createCluster() throws Exception {
        CassandraConnectionProperties properties = new CassandraConnectionProperties(
);

        Cluster cluster = Cluster.builder().addContactPoint(properties
.getCassandraHost())
                .withPort(properties.getCassandraPort()).build();
        return cluster;
    }

    @Produces
    @Singleton
    public Session createSession(Cluster cluster) throws Exception {
        return cluster.connect();
    }

    @Produces
    @ApplicationScoped
    public CassandraOperations createCassandraOperations(Session session) throws
Exception {

        MappingCassandraConverter cassandraConverter = new MappingCassandraConverter(
);
        cassandraConverter.setUserTypeResolver(new SimpleUserTypeResolver(session
.getCluster(), session.getLoggedKeyspace()));

        CassandraAdminTemplate cassandraTemplate = new CassandraAdminTemplate(session,
cassandraConverter);
        return cassandraTemplate;
    }

    public void close(@Disposes Session session) {
        session.close();
    }

    public void close(@Disposes Cluster cluster) {
        cluster.close();
    }
}
```

The Spring Data for Apache Cassandra CDI extension will pick up `CassandraOperations` 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:

```java
class RepositoryClient {

  @Inject
  PersonRepository repository;

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

Chapter 10. Mapping

Rich mapping support is provided by the `MappingCassandraConverter` . `MappingCassandraConverter` has a rich metadata model that provides a complete 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. The `MappingCassandraConverter` also allows you to map domain objects to tables without providing any additional metadata, by following a set of conventions.

In this section we will describe the features of the `MappingCassandraConverter`, how to use conventions for mapping domain objects to tables and how to override those conventions with annotation-based mapping metadata.

10.1. Convention based Mapping

`MappingCassandraConverter` uses a few conventions for mapping domain 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 `savingsaccount` table name.
- The converter will use any registered Spring Converters to override the default mapping of object properties to tables fields.
- The properties of an object are used to convert to and from properties in the table.

10.2. Data mapping and type conversion

This section explains how types are mapped to an Apache Cassandra representation and vice versa.

Spring Data for Apache Cassandra supports several types that are provided by Apache Cassandra. In addition to these types, Spring Data for Apache Cassandra provides a set of built-in converters to map additional types. You can provide your own converters to adjust type conversion, see Overriding Mapping with explicit Converters for further details.

Table 4. Type

Type	Cassandra types
`String`	`text` (default), `varchar`, `ascii`
`double`, `Double`	`double`
`float`, `Float`	`float`
`long`, `Long`	`bigint` (default), `counter`
`int`, `Integer`	`int`
`short`, `Short`	`smallint`
`byte`, `Byte`	`tinyint`
`boolean`, `Boolean`	`boolean`
`BigInteger`	`varint`

Type	Cassandra types
`BigDecimal`	`decimal`
`java.util.Date`	`timestamp`
`com.datastax.driver.core.LocalDate`	`date`
`InetAddress`	`inet`
`ByteBuffer`	`blob`
`java.util.UUID`	`timeuuid`
`UDTValue`, mapped User-Defined types	user type
`java.util.Map<K, V>`	`map`
`java.util.List<E>`	`list`
`java.util.Set<E>`	`set`
`Enum`	`text` (default), `bigint`, `varint`, `int`, `smallint`, `tinyint`
`LocalDate` (Joda, Java 8, JSR310-BackPort)	`date`
`LocalDateTime`, `LocalTime`, `Instant` (Joda, Java 8, JSR310-BackPort)	`timestamp`
`DateMidnight` (Joda)	`date`
`ZoneId` (Java 8, JSR310-BackPort)	`text`

Each supported type maps to a default Cassandra data type. Java types can be mapped to other Cassandra types by using `@CassandraType`.

```
.Enum Mapping to Numeric types
```

```
@Table
public class EnumToOrdinalMapping {

  @PrimaryKey String id;

  @CassandraType(type = Name.INT) Condition asOrdinal;
}

public enum Condition {
  NEW, USED
}
```

NOTE `Enum` mapping using ordinal values requires at least Spring 4.3.0. Using earlier Spring versions require custom converters for each `Enum` type.

10.2.1. Mapping Configuration

Unless explicitly configured, an instance of `MappingCassandraConverter` 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 for 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.

Example 79. @Configuration class to configure Cassandra mapping support

```java
@Configuration
public static class Config extends AbstractCassandraConfiguration {

  @Override
  protected String getKeyspaceName() {
    return "bigbank";
  }

  // the following are optional

  @Override
  public CustomConversions customConversions() {

    List<Converter<?, ?>> converters = new ArrayList<Converter<?, ?>>();
    converters.add(new PersonReadConverter());
    converters.add(new PersonWriteConverter());

    return new CustomConversions(converters);
  }

  @Override
  public SchemaAction getSchemaAction() {
    return SchemaAction.RECREATE;
  }

  // other methods omitted...
}
```

`AbstractCassandraConfiguration` requires you to implement methods that define a keyspace. `AbstractCassandraConfiguration` also has a method you can override named `getEntityBasePackages(…)` which tells the `Converter` where to scan for classes annotated with the `@Table` annotation.

You can add additional converters to the `Converter` by overriding the method `customConversions`.

NOTE — `AbstractCassandraConfiguration` will create a `CassandraTemplate` instance and register it with the container under the name `cassandraTemplate`.

10.3. Metadata based Mapping

To take full advantage of the object mapping functionality inside the Spring Data for Apache Cassandra support, you should annotate your mapped objects with the `@Table` annotation. It allows the classpath scanner to find and pre-process your domain objects to extract the necessary metadata. Only annotated entities will be used to perform schema actions. In the worst case, a `SchemaAction.RECREATE_DROP_UNUSED` will drop your tables and you will experience data loss.

Example 80. Example domain object

```
package com.mycompany.domain;

@Table
public class Person {

  @Id
  private String id;

  @CassandraType(type = Name.VARINT)
  private Integer ssn;

  private String firstName;

  private String lastName;
}
```

IMPORTANT — The `@Id` annotation tells the mapper which property you want to use for the Cassandra primary key. Composite primary keys can require a slightly different data model.

10.3.1. Mapping annotation overview

The `MappingCassandraConverter` can use metadata to drive the mapping of objects to rows. An overview of the annotations is provided below:

- `@Id` - applied at the field or property level to mark the property used for identity purpose.
- `@Table` - applied at the class level to indicate this class is a candidate for mapping to the database. You can specify the name of the table where the object will be stored.
- `@PrimaryKey` - Similar to `@Id` but allows you to specify the column name.
- `@PrimaryKeyColumn` - Cassandra-specific annotation for primary key columns that allows you to specify primary key column attributes such as for clustered/partitioned. Can be used on single and multiple attributes to indicate either a single or a compound primary key.

* `@PrimaryKeyClass` - applied at the class level to indicate this class is a compound primary key class. Requires to be referenced with `@PrimaryKey`.
* `@Transient` - by default all private fields are mapped to the row, this annotation excludes the field where it is applied from being stored in the database.
* `@Column` - applied at the field level. Describes the column name as it will be represented in the Cassandra table thus allowing the name to be different than the field name of the class.
* `@CassandraType` - applied at the field level to specify a Cassandra data type. Types are derived from the declaration by default.
* `@UserDefinedType` - applied at the type level to specify a Cassandra user-defined data type (UDT). Types are derived from the declaration by default.

The mapping metadata infrastructure is defined in the separate, spring-data-commons project that is technology agnostic.

Here is an example of a more complex mapping.

Example 81. Mapped `Person` *class*

```
@Table("my_person")
public class Person {

  @PrimaryKeyClass
  public static class Key implements Serializable {

    @PrimaryKeyColumn(ordinal = 0, type = PrimaryKeyType.PARTITIONED)
    private String type;

    @PrimaryKeyColumn(ordinal = 1, type = PrimaryKeyType.PARTITIONED)
    private String value;

    @PrimaryKeyColumn(name = "correlated_type", ordinal = 2, type =
PrimaryKeyType.CLUSTERED)
    private String correlatedType;

    // other getters/setters ommitted
  }

  @PrimaryKey
  private Person.Key key;

  @CassandraType(type = Name.VARINT)
  private Integer ssn;

  @Column("f_name")
  private String firstName;

  @Column(forceQuote = true)
  private String lastName;
  private Address address;

  @CassandraType(type = Name.UDT, userTypeName = "myusertype")
  private UDTValue usertype;

  @Transient
  private Integer accountTotal;

  @CassandraType(type = Name.SET, typeArguments = Name.BIGINT)
  private Set<Long> timestamps;

  private Map<String, InetAddress> sessions;

  public Person(Integer ssn) {
    this.ssn = ssn;
  }

  public String getId() {
    return id;
  }

  // no setter for Id.  (getter is only exposed for some unit testing)

  public Integer getSsn() {
    return ssn;
  }

  // other getters/setters ommitted
}
```

Example 82. Mapped User-Defined type `Address`

```
@UserDefinedType("address")
public class Address {

  private String city;

  @CassandraType(type = Name.VARCHAR)
  private String street;

  private Set<String> zipcodes;

  @CassandraType(type = Name.SET, typeArguments = Name.BIGINT)
  private List<Long> timestamps;

  // other getters/setters ommitted
}
```

NOTE Working with User-Defined Types requires a `UserTypeResolver` configured with the mapping context. See the configuration chapter for how to configure a `UserTypeResolver`.

10.3.2. Overriding Mapping with explicit Converters

When storing and querying your objects it is convenient to have a `CassandraConverter` instance handle the mapping of all Java types to Rows. However, sometimes you may want the `CassandraConverter` to do most of the work but still allow you to selectively handle the conversion for a particular type, or to optimize performance.

To selectively handle the conversion yourself, register one or more `org.springframework.core.convert.converter.Converter` instances with the `CassandraConverter`.

NOTE Spring 3.0 introduced a `o.s.core.convert` package that provides a general type conversion system. This is described in detail in the Spring reference documentation section entitled Spring Type Conversion.

Below is an example of a Spring `Converter` implementation that converts from a Row to a Person POJO.

```java
@ReadingConverter
 public class PersonReadConverter implements Converter<Row, Person> {

  public Person convert(Row source) {
    Person p = new Person(row.getString("id"));
    p.setAge(source.getInt("age"));
    return p;
  }
}
```

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 5. Attributes

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

Appendix B: Populators namespace reference

The <populator /> element

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

Table 6. Attributes

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

Appendix C: Repository query keywords

Supported query keywords

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

Table 7. Query keywords

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

Appendix D: Repository query return types

Supported query return types

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

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

Table 8. Query return types

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

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