Spring Data for Apache Cassandra - Reference Documentation

David Webb, Matthew Adams, John Blum, Mark Paluch, Jay Bryant

– Version 2.1.1.RELEASE, 2018-10-15

Table of Contents

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. This project has noticeable 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 and 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 you are familiar with Cassandra as well as core Spring concepts.

1. Knowing Spring

Spring Data uses the Spring Framework’s core functionality, including:

- IoC container
- validation, type conversion and data binding
- expression language
- AOP
- JMX integration
- DAO support
- DAO Exception Hierarchy

While it is not important to know the Spring APIs, understanding the concepts behind them is important. 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 use all the features of Spring Data for Apache Cassandra, such as the repository support, you must configure some parts of the library by using Spring.

To learn more about Spring, you can refer to the comprehensive documentation that explains the Spring Framework in detail. There are a lot of articles, blog entries, and books on Spring. See the Spring Framework home page for more information.

---

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 you be 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 does not take more then 5-10 minutes to go through them, and, if you come from a RDBMS background, these exercises can often be an eye opener.

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

- The DataStax site offers commercial support and many resources, including, but not limited to, documentation, DataStax Academy, a Tech Blog, and so on.
- The DataStax Academy introduction to Cassandra.
- The Cassandra Quick Start Guide.

---

3. Requirements

Spring Data for Apache Cassandra 2.x binaries require JDK level 8.0 and later and Spring Framework 5.1.1.RELEASE and later.

It requires Cassandra 2.0 or later.

---

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 the Spring Data for Apache Cassandra module. However, if you encounter issues or you need advice, feel free to use one of the links below:

Community Forum

Spring Data on Stack Overflow 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. The two key tags to search for related answers to this project are spring-data and spring-data-cassandra.

Professional Support

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

4.1. Following Development

For information on the Spring Data for Apache Cassandra source code repository, nightly builds, and snapshot artifacts see the Spring Data for Apache Cassandra home page. You can help make Spring Data best serve the needs of the Spring community by interacting with developers through the community on Stack Overflow. To follow developer activity, look for the mailing list information on the Spring Data for Apache Cassandra home page. 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 ecosystem, subscribe to the Spring Community Portal. Finally, you can follow the Spring blog or the project team on Twitter (SpringData).

4.2. 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

5. New & Noteworthy

This chapter summarizes changes and new features for each release.

5.1. What's new in Spring Data for Apache Cassandra 2.1

- New annotations for `@CountQuery` and `@ExistsQuery`.
- Template API extended with `count(…)` and `exists(…)` methods accepting `Query`.
- Fluent API for CRUD operations.
- Cassandra Mapped Tuple support via `@Tuple`.
- Support for Cassandra `time` columns via `LocalTime`.
- Support for `map` columns using User-defined/converted types.
- Lifecycle Events.
- Kotlin extensions for Template API.
- Reactive Paging support through `Mono<Slice<T>>`.

5.2. What's new in Spring Data for Apache Cassandra 2.0

- Upgraded to Java 8.
- Reactive Apache Cassandra support.
- Merged `spring-cql` and `spring-data-cassandra` modules into a single module and re-packaged `org.springframework.cql` to `org.springframework.data.cassandra`.
- Revised `CqlTemplate`, `AsyncCqlTemplate`, `CassandraTemplate` and `AsyncCassandraTemplate` implementations.
- Added routing capabilities via `SessionFactory` and `AbstractRoutingSessionFactory`.
- Introduced `Update` and `Query` objects.
- Renamed CRUD *Repository* interface: `CassandraRepository` using `MapId` is now renamed to `MapIdCassandraRepository`. `TypedIdCassandraRepository` is renamed to `CassandraRepository`.
- Pagination via `PagingState` and `CassandraPageRequest`.
- Interface and DTO projections in Repository query methods.
- Lightweight transaction support via `InsertOptions` and `UpdateOptions` using the Template API.
- Query options for Repository query methods.

- Introduced new annotation for `@AllowFiltering`.
- Index creation on application startup via `@Indexed` and `@SASI`.
- Tooling support for null-safety via Spring's `@NonNullApi` and `@Nullable` annotations.

5.3. What's new in Spring Data for Apache Cassandra 1.5

- Assert compatibility with Cassandra 3.0 and Cassandra Java Driver 3.0.
- Added 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 enable building custom, composed annotations: `@Table`, `@UserDefinedType`, `@PrimaryKey`, `@PrimaryKeyClass`, `@PrimaryKeyColumn`, `@Column`, `@Query`, `@CassandraType`.

---

6. Dependencies

Due to the 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 to rely on the Spring Data Release Train BOM that we ship with the compatible versions defined. In a Maven project, you would declare this dependency in the `<dependencyManagement />` section of your POM, as follows:

Example 1. Using the Spring Data release train BOM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-releasetrain</artifactId>
      <version>Lovelace-SR1</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The current release train version is `Lovelace-SR1`. The train names ascend alphabetically and the currently available trains 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`, and so on: Milestones
- `RC1`, `RC2`, and so on: Release candidates
- `RELEASE`: GA release
- `SR1`, `SR2`, and so on: Service releases

A working example of using the BOMs can be found in our Spring Data examples repository. With that in place, you can declare the Spring Data modules you would like to use without a version in the `<dependencies />` block, as follows:

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 selects a recent version of Spring Data modules for you. If you still want to upgrade to a newer version, configure the property `spring-data-releasetrain.version` to the train name and iteration you would like to use.

6.2. Spring Framework

The current version of Spring Data modules require Spring Framework in version 5.1.1.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.

7. Working with Spring Data Repositories

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

Spring Data repository documentation and your module

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. You should adapt the XML namespace declaration and the types to be extended to the equivalents of the particular module that you use. "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, see the chapter on that module of this document.

7.1. Core concepts

The central interface in the Spring Data repository abstraction is `Repository`. It takes the domain class to manage as well as the ID type of the domain class as type arguments. This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one. The `CrudRepository` provides sophisticated CRUD functionality for the entity class that is being managed.

Example 3. `CrudRepository` *interface*

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

  <S extends T> S save(S entity);       1

  Optional<T> findById(ID primaryKey);  2

  Iterable<T> findAll();                3

  long count();                         4

  void delete(T entity);                5
```

```
  boolean existsById(ID primaryKey);    6

  // … more functionality omitted.
}
```

1 Saves the given entity.

2 Returns the entity identified by the given ID.

3 Returns all entities.

4 Returns the number of entities.

5 Deletes the given entity.

6 Indicates whether an entity with the given ID exists.

We also provide persistence technology-specific abstractions, such as `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 such as `CrudRepository`.

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

Example 4. `PagingAndSortingRepository` *interface*

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

  Iterable<T> findAll(Sort sort);

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

To access the second page of `User` by a page size of 20, you could do something like the following:

```
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. The following list shows the interface definition for a derived count query:

Example 5. Derived Count Query

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

  long countByLastname(String lastname);
}
```

The following list shows the interface definition for a derived delete query:

Example 6. Derived Delete Query

```
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 should handle, as shown in the following example:

   ```
   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 with JavaConfig or with XML configuration.

   a. To use Java configuration, create a class similar to the following:

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

@EnableJpaRepositories
class Config {}
```

   b. To use XML configuration, define a bean similar to the following:

```
<?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 use the repository abstraction for any other store, you need to change this to the appropriate namespace declaration of your store module. In other words, you should exchange `jpa` in favor of, for example, `mongodb`.

   + Also, note that the JavaConfig variant does not configure a package explicitly, because the package of the annotated class is used by default. To customize the package to scan, use one of the `basePackage…` attributes of the data-store-specific repository's `@Enable${store}Repositories` -annotation.

4. Inject the repository instance and use it, as shown in the following example:

```
class SomeClient {

  private final PersonRepository repository;

  SomeClient(PersonRepository repository) {
    this.repository = repository;
  }

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

The sections that follow explain each step in detail:

- Defining Repository Interfaces
- Defining Query Methods
- Creating Repository Instances
- Custom Implementations for Spring Data Repositories

7.3. Defining Repository Interfaces

First, 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 extends `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, copy the methods you want to expose from `CrudRepository` into your domain repository.

Doing so lets you define your own abstractions on top of the provided Spring Data Repositories functionality.

The following example shows how to selectively expose CRUD methods (`findById` and `save`, in this case):

Example 7. Selectively exposing CRUD methods

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

  Optional<T> findById(ID id);

  <S extends T> S save(S entity);
}
```

```
interface UserRepository extends MyBaseRepository<User, Long> {
  User findByEmailAddress(EmailAddress emailAddress);
}
```

In the prior example, you defined a common base interface for all your domain repositories and exposed `findById(…)` as well as `save(…)` .These methods are routed into the base repository implementation of the store of your choice provided by Spring Data (for example, if you use JPA, the implementation is `SimpleJpaRepository`), because they match the method signatures in `CrudRepository` . So the `UserRepository` can now save users, find individual users by ID, and trigger a query to find `Users` by email address.

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

7.3.2. Null Handling of Repository Methods

As of Spring Data 2.0, repository CRUD methods that return an individual aggregate instance use Java 8's `Optional` to indicate the potential absence of a value. Besides that, Spring Data supports returning the following wrapper types on query methods:

- `com.google.common.base.Optional`
- `scala.Option`
- `io.vavr.control.Option`
- `javaslang.control.Option` (deprecated as Javaslang is deprecated)

Alternatively, query methods can choose not to use a wrapper type at all. The absence of a query result is then indicated by returning `null` . Repository methods returning collections, collection alternatives, wrappers, and streams are guaranteed never to return `null` but rather the corresponding empty representation. See "Repository query return types" for details.

Nullability Annotations

You can express nullability constraints for repository methods by using Spring Framework's nullability annotations. They provide a tooling-friendly approach and opt-in `null` checks during runtime, as follows:

- `@NonNullApi` : Used on the package level to declare that the default behavior for parameters and return values is to not accept or produce `null` values.
- `@NonNull` : Used on a parameter or return value that must not be `null` (not needed on a parameter and return value where `@NonNullApi` applies).
- `@Nullable` : Used on a parameter or return value that can be `null` .

Spring annotations are meta-annotated with JSR 305 annotations (a dormant but widely spread JSR). JSR 305 meta-annotations let tooling vendors such as IDEA, Eclipse, and Kotlin provide null-safety support in a generic way, without having to hard-code support for Spring annotations. To enable runtime checking of nullability constraints for query methods, you need to activate non-nullability on the package level by using Spring's `@NonNullApi` in `package-info.java` , as shown in the following example:

Example 8. Declaring Non-nullability in `package-info.java`

```
@org.springframework.lang.NonNullApi
package com.acme;
```

Once non-null defaulting is in place, repository query method invocations get validated at runtime for nullability constraints. If a query execution result violates the defined constraint, an exception is thrown. This happens when the method would return `null` but is declared as non-nullable (the default with the annotation defined on the package the repository resides in). If you want to opt-in to nullable results again, selectively use `@Nullable` on individual methods. Using the result wrapper types mentioned at the start of this section continues to work as expected: An empty result is translated into the value that represents absence.

The following example shows a number of the techniques just described:

Example 9. Using different nullability constraints

```
package com.acme;                                                 1

import org.springframework.lang.Nullable;

interface UserRepository extends Repository<User, Long> {

  User getByEmailAddress(EmailAddress emailAddress);              2

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

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

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

2 Throws an `EmptyResultDataAccessException` when the query executed does not produce a result. Throws an `IllegalArgumentException` when the `emailAddress` handed to the method is `null`.

3 Returns `null` when the query executed does not produce a result. Also accepts `null` as the value for `emailAddress`.

4 Returns `Optional.empty()` when the query executed does not produce a result. Throws an `IllegalArgumentException` when the `emailAddress` handed to the method is `null`.

Nullability in Kotlin-based Repositories

Kotlin has the definition of [nullability constraints] baked into the language. Kotlin code compiles to bytecode, which does not express nullability constraints through method signatures but rather through compiled-in metadata. Make sure to include the `kotlin-reflect` JAR in your project to enable introspection of Kotlin's nullability constraints. Spring Data repositories use the language mechanism to define those constraints to apply the same runtime checks, as follows:

Example 10. Using nullability constraints on Kotlin repositories

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

  fun findByUsername(username: String): User      1

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

1 The method defines both the parameter and the result as non-nullable (the Kotlin default). The Kotlin compiler rejects method invocations that pass `null` to the method. If the query execution yields an empty result, an `EmptyResultDataAccessException` is thrown.

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

7.3.3. Using Repositories with Multiple Spring Data Modules

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

The following example shows a repository that uses module-specific interfaces (JPA in this case):

Example 11. Repository definitions using module-specific interfaces

```
interface MyRepository extends JpaRepository<User, Long> { }

@NoRepositoryBean
interface MyBaseRepository<T, ID extends Serializable> extends JpaRepository<T, ID> {
  …
}

interface UserRepository extends MyBaseRepository<User, Long> {
  …
}
```

`MyRepository` and `UserRepository` extend `JpaRepository` in their type hierarchy. They are valid candidates for the Spring Data JPA module.

The following example shows a repository that uses generic interfaces:

Example 12. Repository definitions using generic interfaces

```
interface AmbiguousRepository extends Repository<User, Long> {
 …
}

@NoRepositoryBean
interface MyBaseRepository<T, ID extends Serializable> extends CrudRepository<T, ID> {
  …
}

interface AmbiguousUserRepository extends MyBaseRepository<User, Long> {
  …
}
```

`AmbiguousRepository` and `AmbiguousUserRepository` extend only `Repository` and `CrudRepository` in their type hierarchy. While this is perfectly fine when using a unique Spring Data module, multiple modules cannot distinguish to which particular Spring Data these repositories should be bound.

The following example shows a repository that uses domain classes with annotations:

Example 13. Repository definitions using domain classes with annotations

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

@Entity
class Person {
  …
}

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

@Document
class User {
  …
}
```

`PersonRepository` references `Person`, which is annotated with the JPA `@Entity` annotation, so this repository clearly belongs to Spring Data JPA. `UserRepository` references `User`, which is annotated with Spring Data MongoDB's `@Document` annotation.

The following bad example shows a repository that uses domain classes with mixed annotations:

Example 14. Repository definitions using domain classes with mixed annotations

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

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

@Entity
@Document
class Person {
  …
}
```

This example shows a domain class using both JPA and Spring Data MongoDB annotations. It defines two repositories, `JpaPersonRepository` and `MongoDBPersonRepository`. One is intended for JPA and the other for MongoDB usage. Spring Data is no longer able to tell the repositories apart, which leads to undefined behavior.

Repository type details and distinguishing domain class annotations are used for strict repository configuration to identify repository candidates for a particular Spring Data module. Using multiple persistence technology-specific annotations on the same domain type is possible and enables reuse of domain types across multiple persistence technologies. However, Spring Data can then no longer determine a unique module with which to bind the repository.

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

The following example shows annotation-driven configuration of base packages:

Example 15. 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:

- By deriving the query from the method name directly.
- By using a manually defined query.

Available options depend on the actual store. However, there must be a strategy that decides what actual query is created. The next section describes the available options.

7.4.1. Query Lookup Strategies

The following strategies are available for the repository infrastructure to resolve the query. With XML configuration, you can configure the strategy at the namespace through the `query-lookup-strategy` attribute. For Java configuration, you can use the `queryLookupStrategy` attribute of the `Enable${store}Repositories` annotation. 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. You can read more about query construction in "Query Creation".
- `USE_DECLARED_QUERY` tries to find a declared query and throws an exception if cannot 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, is 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`. The following example shows how to create a number of queries:

Example 16. Query creation from method names

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

  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

  // Enables the distinct flag for the query
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String
firstname);
  List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String
firstname);

  // Enabling ignoring case for an individual property
  List<Person> findByLastnameIgnoreCase(String lastname);
  // Enabling ignoring case for all suitable properties
  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String
firstname);

  // Enabling static ORDER BY for a query
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

The actual result of parsing the method depends on the persistence store for which you create the query. However, there are some general things to notice:

- The expressions are usually property traversals combined with operators that can be concatenated. You can combine property expressions with `AND` and `OR`. You also get support for operators such as `Between`, `LessThan`, `GreaterThan`, and `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 supports 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. Consider the following method signature:

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

Assume a `Person` has an `Address` with a `ZipCode` . In that case, the method creates the property traversal `x.address.zipCode` . The resolution algorithm starts by 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 continues building the tree down from there, splitting the tail up in the way just described. If the first split does not match, the algorithm moves 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, choose the wrong property, and 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 be as follows:

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

Because we treat the underscore character as a reserved character, we strongly advise following standard Java naming conventions (that is, not using underscores in property names but using camel case instead).

7.4.4. Special parameter handling

To handle parameters in your query, define method parameters as already seen in the preceding examples. Besides that, the infrastructure recognizes certain specific types like `Pageable` and `Sort`, to apply pagination and sorting to your queries dynamically. The following example demonstrates these features:

Example 17. Using `Pageable`, `Slice`, *and* `Sort` *in query methods*

```
Page<User> findByLastname(String lastname, Pageable pageable);

Slice<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);
```

The first method lets you 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), you can instead return a `Slice`. A `Slice` only knows about whether a next `Slice` is available, which might be sufficient when walking through a larger result set.

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

To find out how many pages you get for an entire query, you have to trigger an additional count query. By default, this query is derived from the query you actually trigger.

7.4.5. Limiting Query Results

The results of query methods can be limited by using the `first` or `top` keywords, which can be used interchangeably. An optional numeric value can be appended to `top` or `first` to specify the maximum result size to be returned. If the number is left out, a result size of 1 is assumed. The following example shows how to limit the query size:

Example 18. Limiting the result size of a query with `Top` and `First`

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

The limiting expressions also support the `Distinct` keyword. Also, for the queries limiting the result set to one instance, wrapping the result into with the `Optional` keyword is supported.

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

Limiting the results in combination with dynamic sorting by using a `Sort` parameter lets you 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 wrapping the query results in a `Stream` data store-specific methods are used to perform the streaming, as shown in the following example:

Example 19. Stream the result of a query with Java 8 `Stream<T>`

```
@Query("select u from User u")
Stream<User> findAllByCustomQueryAndStream();

Stream<User> readAllByFirstnameNotNull();

@Query("select u from User u")
Stream<User> streamAllPaged(Pageable pageable);
```

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

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

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

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

7.4.7. Async query results

Repository queries can be run asynchronously by using Spring's asynchronous method execution capability. This means the method returns immediately upon invocation while the actual query execution occurs in a task that has been submitted to a Spring `TaskExecutor`. Asynchronous query execution is different from reactive query execution and should not be mixed. Refer to store-specific documentation for more details on reactive support. The following example shows a number of asynchronous queries:

```
@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 the return type.

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

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

7.5. Creating Repository Instances

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

7.5.1. XML configuration

Each Spring Data module includes a `repositories` element that lets you define a base package that Spring scans for you, as shown in the following example:

Example 21. Enabling Spring Data repositories via XML

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

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

</beans:beans>
```

In the preceding example, Spring is instructed to scan `com.acme.repositories` and all its sub-packages for interfaces extending `Repository` or one of its sub-interfaces. For each interface found, the infrastructure registers the persistence technology-specific `FactoryBean` to create the appropriate proxies that handle invocations of the query methods. Each bean is registered under a bean name that is derived from the interface name, so an interface of `UserRepository` would be registered under `userRepository`. The `base-package` attribute allows wildcards so that you can define a pattern of scanned packages.

Using filters

By default, the infrastructure picks up every interface extending the persistence technology-specific `Repository` sub-interface located under the configured base package and creates a bean instance for it. However, you might want more fine-grained control over which interfaces have bean instances created for them. To do so, use `<include-filter />` and `<exclude-filter />` elements inside the `<repositories />` element. The semantics are exactly equivalent to the elements in Spring's context namespace. For details, see the Spring reference documentation for these elements.

For example, to exclude certain interfaces from instantiation as repository beans, you could use the following configuration:

Example 22. Using exclude-filter element

```
<repositories base-package="com.acme.repositories">
  <context:exclude-filter type="regex" expression=".*SomeRepository" />
</repositories>
```

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

7.5.2. JavaConfig

The repository infrastructure can also be triggered by using a store-specific `@Enable${store}Repositories` annotation on a JavaConfig class. For an introduction into Java-based configuration of the Spring container, see JavaConfig in the Spring reference documentation.

A sample configuration to enable Spring Data repositories resembles the following:

Example 23. Sample annotation based repository configuration

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

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

The preceding example 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. See the sections covering the store-specific configuration.

7.5.3. Standalone usage

You can also use the repository infrastructure outside of a Spring container — for example, in CDI environments. You still need some Spring libraries in your classpath, but, generally, you can set up repositories programmatically as well. The Spring Data modules that provide repository support ship a persistence technology-specific `RepositoryFactory` that you can use as follows:

Example 24. Standalone usage of repository factory

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

7.6. Custom Implementations for Spring Data Repositories

This section covers repository customization and how fragments form a composite repository.

When a query method requires a different behavior or cannot be implemented by query derivation, then it is necessary to provide a custom implementation. Spring Data repositories let you provide custom repository code and integrate it with generic CRUD abstraction and query method functionality.

7.6.1. Customizing Individual Repositories

To enrich a repository with custom functionality, you must first define a fragment interface and an implementation for the custom functionality, as shown in the following example:

Example 25. Interface for custom repository functionality

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

Then you can let your repository interface additionally extend from the fragment interface, as shown in the following example:

Example 26. Implementation of custom repository functionality

```
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

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

The most important part of the class name that corresponds to the fragment interface is the `Impl` postfix.

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

You can let your repository interface extend the fragment interface, as shown in the following example:

Example 27. Changes to your repository interface

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

```
  // Declare query methods here
}
```

Extending the fragment interface with your repository interface combines the CRUD and custom functionality and makes it available to clients.

Spring Data repositories are implemented by using fragments that form a repository composition. Fragments are the base repository, functional aspects (such as QueryDsl), and custom interfaces along with their implementation. Each time you add an interface to your repository interface, you enhance the composition by adding a fragment. The base repository and repository aspect implementations are provided by each Spring Data module.

The following example shows custom interfaces and their implementations:

Example 28. Fragments with their implementations

```
interface HumanRepository {
  void someHumanMethod(User user);
}

class HumanRepositoryImpl implements HumanRepository {

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

interface ContactRepository {

  void someContactMethod(User user);

  User anotherContactMethod(User user);
}

class ContactRepositoryImpl implements ContactRepository {

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

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

The following example shows the interface for a custom repository that extends `CrudRepository`:

Example 29. Changes to your repository interface

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

  // Declare query methods here
}
```

Repositories may be composed of multiple custom implementations that are imported in the order of their declaration. Custom implementations have a higher priority than the base implementation and repository aspects. This ordering lets you override base repository and aspect methods and resolves ambiguity if two fragments contribute the same method signature. Repository fragments are not limited to use in a single repository interface. Multiple repositories may use a fragment interface, letting you reuse customizations across different repositories.

The following example shows a repository fragment and its implementation:

Example 30. Fragments overriding `save(…)`

```
interface CustomizedSave<T> {
  <S extends T> S save(S entity);
}

class CustomizedSaveImpl<T> implements CustomizedSave<T> {

  public <S extends T> S save(S entity) {
    // Your custom implementation
  }
}
```

The following example shows a repository that uses the preceding repository fragment:

Example 31. Customized repository interfaces

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

```
interface PersonRepository extends CrudRepository<Person, Long>, CustomizedSave<Person> {
}
```

Configuration

If you use namespace configuration, the repository infrastructure tries to autodetect custom implementation fragments by scanning for classes below the package in which it found a repository. These classes need to follow the naming convention of appending the namespace element's `repository-impl-postfix` attribute to the fragment interface name. This postfix defaults to `Impl`. The following example shows a repository that uses the default postfix and a repository that sets a custom value for the postfix:

Example 32. Configuration example

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

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

The first configuration in the preceding example tries to look up a class called `com.acme.repository.CustomizedUserRepositoryImpl` to act as a custom repository implementation. The second example tries to lookup `com.acme.repository.CustomizedUserRepositoryMyPostfix`.

Resolution of Ambiguity

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

Given the following two custom implementations for the `CustomizedUserRepository` shown earlier, the first implementation is used. Its bean name is `customizedUserRepositoryImpl`, which matches that of the fragment interface (`CustomizedUserRepository`) plus the postfix `Impl`.

Example 33. Resolution of amibiguous implementations

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

class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

  // Your custom implementation
}
```

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

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

  // Your custom implementation
}
```

If you annotate the `UserRepository` interface with `@Component("specialCustom")`, the bean name plus `Impl` then matches the one defined for the repository implementation in `com.acme.impl.two`, and it is used instead of the first one.

Manual Wiring

If your custom implementation uses annotation-based configuration and autowiring only, the preceding approach shown works well, because it is treated as any other Spring bean. If your implementation fragment bean needs special wiring, you can declare the bean and name it according to the conventions described in the preceding section. The infrastructure then refers to the manually defined bean definition by name instead of creating one itself. The following example shows how to manually wire a custom implementation:

Example 34. Manual wiring of custom implementations

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

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

7.6.2. Customize the Base Repository

The approach described in the preceding section requires customization of each repository interfaces when you want to customize the base repository behavior so that all repositories are affected. To instead change behavior for all repositories, you can create an implementation that extends the persistence technology-specific repository base class. This class then acts as a custom base class for the repository proxies, as shown in the following example:

Example 35. Custom repository base class

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

  private final EntityManager entityManager;

  MyRepositoryImpl(JpaEntityInformation entityInformation,
                           EntityManager entityManager) {
    super(entityInformation, entityManager);

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

  @Transactional
  public <S extends T> S save(S entity) {
    // implementation goes here
  }
}
```

The class needs to have a constructor of the super class which the store-specific repository factory implementation uses. If the repository base class has multiple constructors, override the one taking an `EntityInformation` plus a store specific infrastructure object (such as an `EntityManager` or a template class).

The final step is to make the Spring Data infrastructure aware of the customized repository base class. In Java configuration, you can do so by using the `repositoryBaseClass` attribute of the `@Enable${store}Repositories` annotation, as shown in the following example:

Example 36. Configuring a custom repository base class using JavaConfig

```
@Configuration
@EnableJpaRepositories(repositoryBaseClass = MyRepositoryImpl.class)
class ApplicationConfiguration { … }
```

A corresponding attribute is available in the XML namespace, as shown in the following example:

Example 37. 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 called `@DomainEvents` that you can use on a method of your aggregate root to make that publication as easy as possible, as shown in the following example:

Example 38. Exposing domain events from an aggregate root

```
class AnAggregateRoot {

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

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

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

2 After all events have been published, we have a method annotated with `@AfterDomainEventPublication`. It can be used to potentially clean the list of events to be published (among other uses).

The methods are 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 that enables the construction of statically typed SQL-like queries through its fluent API.

Several Spring Data modules offer integration with Querydsl through `QuerydslPredicateExecutor`, as shown in the following example:

Example 39. QuerydslPredicateExecutor interface

```
public interface QuerydslPredicateExecutor<T> {

  Optional<T> findById(Predicate predicate);   1

  Iterable<T> findAll(Predicate predicate);    2

  long count(Predicate predicate);             3

  boolean exists(Predicate predicate);         4

  // … more functionality omitted.
}
```

1. Finds and returns a single entity matching the `Predicate`.
2. Finds and returns all entities matching the `Predicate`.
3. Returns the number of entities matching the `Predicate`.
4. Returns whether an entity that matches the `Predicate` exists.

To make use of Querydsl support, extend `QuerydslPredicateExecutor` on your repository interface, as shown in the following example

Example 40. Querydsl integration on repositories

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

The preceding example lets you write typesafe queries using Querydsl `Predicate` instances, as shown in the following example:

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

userRepository.findAll(predicate);
```

7.8.2. Web support

This section contains the documentation for the Spring Data web support as it is implemented in the current (and later) versions of Spring Data Commons. As the newly introduced support changes many things, we kept the documentation of the former behavior in [web.legacy].

Spring Data modules that support the repository programming model ship with a variety of web support. The web related components require Spring MVC JARs to be on the classpath. Some of them even provide integration with Spring HATEOAS. In general, the integration support is enabled by using the `@EnableSpringDataWebSupport` annotation in your JavaConfig configuration class, as shown in the following example:

Example 41. Enabling Spring Data web support

```
@Configuration
@EnableWebMvc
@EnableSpringDataWebSupport
class WebConfiguration {}
```

The `@EnableSpringDataWebSupport` annotation registers a few components we will discuss in a bit. It will also detect Spring HATEOAS on the classpath and register integration components for it as well if present.

Alternatively, if you use XML configuration, register either `SpringDataWebConfiguration` or `HateoasAwareSpringDataWebConfiguration` as Spring beans, as shown in the following example (for `SpringDataWebConfiguration`):

Example 42. Enabling Spring Data web support in XML

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

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

Basic Web Support

The configuration shown in the previous section registers a few basic components:

- A `DomainClassConverter` to let Spring MVC 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` lets you use domain types in your Spring MVC controller method signatures directly, so that you need not manually lookup the instances through the repository, as shown in the following example:

Example 43. A Spring MVC controller using domain types in method signatures

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

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

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

As you can see, the method receives a `User` instance directly, and no further lookup is necessary. The instance can be resolved by letting Spring MVC convert the path variable into the `id` type of the domain class first and eventually access the instance through calling `findById(…)` on the repository instance registered for the domain type.

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

HandlerMethodArgumentResolvers for Pageable and Sort

The configuration snippet shown in the previous section also registers a `PageableHandlerMethodArgumentResolver` as well as an instance of `SortHandlerMethodArgumentResolver`. The registration enables `Pageable` and `Sort` as valid controller method arguments, as shown in the following example:

Example 44. Using Pageable as controller method argument

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

  private final UserRepository repository;

  UserController(UserRepository repository) {
    this.repository = repository;
  }

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

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

The preceding method signature causes Spring MVC try to derive a `Pageable` instance from the request parameters by 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 — for example, `?sort=firstname&sort=lastname,asc`.

To customize this behavior, register a bean implementing the `PageableHandlerMethodArgumentResolverCustomizer` interface or the

`SortHandlerMethodArgumentResolverCustomizer` interface, respectively. Its `customize()` method gets called, letting you change settings, as shown in the following example:

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

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

If 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}_`. The followig example shows the resulting method signature:

```
String showUsers(Model model,
      @Qualifier("thing1") Pageable first,
      @Qualifier("thing2") Pageable second) { … }
```

you have to populate `thing1_page` and `thing2_page` and so on.

The default `Pageable` passed into the method is equivalent to a `new PageRequest(0, 20)` but can be customized by 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, called the `PagedResourcesAssembler`. The following example shows how to use a `PagedResourcesAssembler` as a controller method argument:

Example 45. Using a PagedResourcesAssembler as controller method argument

```
@Controller
class PersonController {

  @Autowired PersonRepository repository;
```

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

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

Enabling the configuration as shown in the preceding example lets the `PagedResourcesAssembler` be used as a controller method argument. Calling `toResources(…)` on it has the following effects:

- The content of the `Page` becomes the content of the `PagedResources` instance.
- The `PagedResources` object gets a `PageMetadata` instance attached, and it is populated with information from the `Page` and the underlying `PageRequest`.
- The `PagedResources` may get `prev` and `next` links attached, depending on the page's state. The links point to the URI to which the method maps. The pagination parameters added to the method match the setup of the `PageableHandlerMethodArgumentResolver` to make sure the links can be resolved later.

Assume we have 30 Person instances in the database. You can now trigger a request (`GET http://localhost:8080/persons`) and see output similar to the following:

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

Web Databinding Support

Spring Data projections (described in Projections) can be used to bind incoming request payloads by either using JSONPath expressions (requires Jayway JsonPath or XPath expressions (requires XmlBeam), as shown in the following example:

Example 46. HTTP payload binding using JSONPath or XPath expressions

```
@ProjectedPayload
public interface UserPayload {

  @XBRead("//firstname")
  @JsonPath("$..firstname")
  String getFirstname();

  @XBRead("/lastname")
  @JsonPath({ "$.lastname", "$.user.lastname" })
  String getLastname();
}
```

The type shown in the preceding example can be used as a Spring MVC handler method argument or by using `ParameterizedTypeReference` on one of `RestTemplate`'s methods. The preceding method declarations would try to find `firstname` anywhere in the given document. The `lastname` XML lookup is performed on the top-level of the incoming document. The JSON variant of that tries a top-level `lastname` first but also tries `lastname` nested in a `user` sub-document if the former does not return a value. That way, changes in the structure of the source document can be mitigated easily without having clients calling the exposed methods (usually a drawback of class-based payload binding).

Nested projections are supported as described in Projections. If the method returns a complex, non-interface type, a Jackson `ObjectMapper` is used to map the final value.

For Spring MVC, the necessary converters are registered automatically as soon as `@EnableSpringDataWebSupport` is active and the required dependencies are available on the classpath. For usage with `RestTemplate`, register a `ProjectingJackson2HttpMessageConverter` (JSON) or `XmlBeamHttpMessageConverter` manually.

For more information, see the web projection example in the canonical Spring Data Examples repository.

Querydsl Web Support

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

Consider the following query string:

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

Given the `User` object from previous examples, a query string can be resolved to the following value by using the `QuerydslPredicateArgumentResolver`.

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

The feature is automatically enabled, along with `@EnableSpringDataWebSupport`, when Querydsl is found on the classpath.

Adding a `@QuerydslPredicate` to the method signature provides a ready-to-use `Predicate`, which can be run by using the `QuerydslPredicateExecutor`.

Type information is typically resolved from the method's return type. Since that information does not necessarily match the domain type, it might be a good idea to use the `root` attribute of `QuerydslPredicate`.

The following exampe shows how to use `@QuerydslPredicate` in a method signature:

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

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

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

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

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

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

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

Example 47. Data defined in JSON

```
[ { "_class" : "com.acme.Person",
 "firstname" : "Dave",
  "lastname" : "Matthews" },
  { "_class" : "com.acme.Person",
 "firstname" : "Carter",
  "lastname" : "Beauford" } ]
```

You can 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, declare a populator similar to the following:

Example 48. Declaring a Jackson repository populator

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

  <repository:jackson2-populator locations="classpath:data.json" />

</beans>
```

The preceding declaration causes the `data.json` file to be read and deserialized by a Jackson `ObjectMapper`.

The type to which the JSON object is unmarshalled is determined by inspecting the `_class` attribute of the JSON document. The infrastructure eventually selects the appropriate repository to handle the object that was deserialized.

To instead use XML to define the data the repositories should be populated with, you can use the `unmarshaller-populator` element. You configure it to use one of the XML marshaller options available in Spring OXM. See the Spring reference documentation for details. The following example shows how to unmarshal a repository populator with JAXB:

Example 49. Declaring an unmarshalling repository populator (using JAXB)

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

  <repository:unmarshaller-populator locations="classpath:data.json"
    unmarshaller-ref="unmarshaller" />

  <oxm:jaxb2-marshaller contextPath="com.acme" />

</beans>
```

Reference Documentation

8. Introduction

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.

Reactive Cassandra Support explains reactive Cassandra specifics.

Cassandra Repositories introduces repository support for Cassandra.

8.1. Spring CQL and Spring Data for Apache Cassandra Modules

Spring Data for Apache Cassandra allows interaction on both the CQL and the entity level.

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

Table 2. Spring Data for Apache Cassandra (CQL Core)- 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 run the statement.	X	
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	

The core CQL support takes care of all the low-level details that can make Cassandra and CQL such a tedious API with which to develop. Using mapped entity objects allows schema generation, object mapping, and repository support.

8.1.1. Choosing an Approach for Cassandra Database Access

You can choose among several approaches to use as a 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. The following approaches work well:

- `CqlTemplate` and `ReactiveCqlTemplate` are the classic Spring CQL approach and the most popular. This is the "lowest-level" approach. Note that components like `CassandraTemplate` use `CqlTemplate` under-the-hood.
- `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.
- `ReactiveCassandraTemplate` wraps a `ReactiveCqlTemplate` 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 lets you 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.

---

9. Cassandra Support

Spring Data support for Apache Cassandra contains a wide range of features:

- Spring configuration support with Java-based `@Configuration` classes or the XML namespace.
- The `CqlTemplate` helper class that increases productivity by properly handling common Cassandra data access operations.
- The `CassandraTemplate` helper class that provides 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 that is extensible to support other metadata formats.
- Java-based query, criteria, and update DSLs.

- Automatic implementation of `Repository` interfaces including support for custom finder methods.

For most data-oriented tasks, you can use the `CassandraTemplate` or the `Repository` support, both of which use the rich object-mapping functionality. `CqlTemplate` is commonly used to increment counters or perform ad-hoc CRUD operations. `CqlTemplate` also provides callback methods that make it easy to get low-level API objects, such as `com.datastax.driver.core.Session`, which lets you 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 that you can map your existing knowledge onto the Spring APIs.

9.1. Getting Started

Spring Data for Apache Cassandra requires Apache Cassandra 2.1 or later and Datastax Java Driver 3.0 or later. An easy way to quickly set up and bootstrap a working environment is to create a Spring-based project in STS or use Spring Initializer.

First, you need to set up a running Apache Cassandra server. See the Apache Cassandra Quick Start Guide for an explanation on how to start 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 and press Yes when prompted. Then enter a project and a package name, such as `org.spring.data.cassandra.example`.

Then you can add the following dependency declaration to your pom.xml file's `dependencies` section.

```
<dependencies>

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-cassandra</artifactId>
    <version>2.1.1.RELEASE</version>
  </dependency>

</dependencies>
```

Also, you should change the version of Spring in the pom.xml file to be as follows:

```
<spring.framework.version>5.1.1.RELEASE</spring.framework.version>
```

If using a milestone release instead of a GA release, you also need to add the location of the Spring Milestone repository for Maven to your pom.xml file so that it is at the same level of your `<dependencies/>` element, as follows:

```
<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 all Spring repositories here.

Now you can create a simple Java application that stores and reads a domain object to and from Cassandra.

To do so, first create a simple domain object class to persist, as the following example shows:

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

import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.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, as the following example shows:

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

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 org.springframework.data.cassandra.core.query.Criteria;
import org.springframework.data.cassandra.core.query.Query;

import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Session;

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));

    LOGGER.info(template.selectOne(Query.query(Criteria.where("id").is(jonDoe.getId())),
Person.class).getId());

    template.truncate(Person.class);
    session.close();
    cluster.close();
  }
}
```

Even in this simple example, there are a few notable things to point out:

- You can create an instance of `CassandraTemplate` with a Cassandra `Session`, obtained from `Cluster`.
- You must annotate your POJO as a Cassandra `@Table` entity 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 raw CQL or the DataStax `QueryBuilder` API to construct your queries.

9.2. Examples Repository

To get a feel for how the library works, you can download and play around with several examples. .

9.3. Connecting to Cassandra with Spring

One of the first tasks when using Apache Cassandra with Spring is to create a `com.datastax.driver.core.Session` object by using the Spring IoC container. You can do so either by using Java-based bean metadata or by using XML-based bean metadata. These are discussed in the following sections.

> 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.

9.3.1. Registering a Session Instance by Using Java-based Metadata

The following example shows how to use Java-based bean metadata to register an instance of a `com.datastax.driver.core.Session`:

Example 50. Registering a com.datastax.driver.core.Session object by 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 lets you use the standard `com.datastax.driver.core.Session` API that you may already know.

An alternative is to register an instance of `com.datastax.driver.core.Session` with the container by 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. This hierarchy and the use of `@Repository` is described in Spring's DAO support features.

The following example shows Java-based bean metadata that supports exception translation on `@Repository` annotated classes:

Example 51. Registering a com.datastax.driver.core.Session object by 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
   */
  @Bean
  public CassandraCqlClusterFactoryBean cluster() {

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

    return cluster;
  }

  /*
   * Factory bean that creates the com.datastax.driver.core.Session instance
   */
  @Bean
  public 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.

The following example shows how to register components to configure object mapping and repository support:

Example 52. 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 that register Spring Data for Apache Cassandra components can be an exhausting challenge, so Spring Data for Apache Cassandra comes with a pre-built configuration support class. Classes that extend from `AbstractCassandraConfiguration` 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 many more. `AbstractCassandraConfiguration` also supports you 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. The following example shows how to register beans by using `AbstractCassandraConfiguration`:

Example 53. Registering Spring Data for Apache Cassandra beans by 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 String getKeyspaceName() {
    return "mykeyspace";
  }
}
```

9.3.2. XML Configuration

This section describes how to configure Spring Data Cassandra with XML.

Externalizing Connection Properties

To externalize connection properties, you should first create a properties file that contains the information needed to connect to Cassandra. `contactpoints` and `keyspace` are the equired fields. We added `port` for clarity.

The following example shows our properties file, called `cassandra.properties`:

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

In the next two examples, we use Spring to load these properties into the Spring context.

Registering a Session Instance by 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, because 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 let you create a `Session` instance.

The following example shows how to configure the `cql` namespace:

Example 54. XML schema to configure Cassandra by 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>
```

To use the Cassandra namespace elements, you need to reference the Cassandra schema, as the following example shows:

Example 55. XML schema to configure Cassandra by using the `cassandra` 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: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>
```

You may have noticed the 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 the preceding 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 includes but is 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 that mapping any existing driver configuration should be straight forward. The following example shows how to configure Spring Data components by using XML

Example 56. Configuring Spring Data components by using 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 foundation of all Spring
Data Cassandra -->
<cassandra:template id="cassandraTemplate" />

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

9.4. 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 schema creation.

9.4.1. Keyspaces and Lifecycle Scripts

The 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 `KeyspaceSpecification` and startup and shutdown CQL script execution.

Declaring a keyspace with a specification allows creating and dropping of the Keyspace. It derives CQL from the specification so that you need not write CQL yourself. The following example specifies a Cassadra keyspace by using XML:

Example 57. Specifying a Cassandra keyspace by using 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>
```

You can also specify a Cassandra keyspace by using Java configuration, as the following example shows:

Example 58. Specifying a Cassandra keyspace by using Java configuration

```
@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 and 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, as the following XML example shows:

Example 59. Specifying Startup and Shutdown scripts with 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>
```

The following example shows how to specify startup and shutdown scripts with Java configuration:

Example 60. Specifying Startup and Shutdown scripts with Java configuration

```
@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;");
  }

  // ...
}
```

Both the `cql` and `cassandra` namespaces include `KeyspaceSpecifications` and the lifecycle CQL scripts.

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 removes the keyspace and all data from the tables in the keyspace.

9.4.2. Tables and User-defined Types

Spring Data for Apache Cassandra approaches data access with mapped entity classes that fit your data model. You can use these entity classes to create Cassandra table specifications and user type definitions.

Schema creation is tied to `Session` initialization by `SchemaAction`. The following actions are supported:

- `SchemaAction.NONE`: No tables or types are created or dropped. This is the default setting.
- `SchemaAction.CREATE`: Create tables, indexes, and user-defined types from entities annotated with `@Table` and types annotated with `@UserDefinedType`. Existing tables or types cause an error if you tried to create the type.
- `SchemaAction.CREATE_IF_NOT_EXISTS`: Like `SchemaAction.CREATE` but with `IF NOT EXISTS` applied. Existing tables or types do not cause any errors but may remain stale.
- `SchemaAction.RECREATE`: Drops and recreates 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`: Drops all tables and types and recreates only known tables and types.

`SchemaAction.RECREATE` and `SchemaAction.RECREATE_DROP_UNUSED` drop your tables and lose all data. `RECREATE_DROP_UNUSED` also drops tables and types that are not known to the application.

Enabling Tables and User-Defined Types for Schema Management

Metadata-based Mapping explains object mapping with conventions and annotations. To prevent unwanted classes from being created as a table or a type, schema management is only active for entities annotated with `@Table` and user-defined types annotated with `@UserDefinedType`. Entities are discovered by scanning the classpath. Entity scanning requires one or more base packages. Tuple-typed columns that use `TupleValue` do not provide any typing details. Consequently, you must annotate such column properties with `@CassandraType(type = TUPLE, typeArguments = …)` to specify the desired column type.

The following example shows how to specify entity base packages in XML configuration:

Example 61. Specifying entity base packages with XML configuration

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

The following example shows how to specify entity base packages in Java configuration:

Example 62. Specifying entity base packages with Java configuration

```
@Configuration
public class CassandraConfiguration extends AbstractCassandraConfiguration {

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

    // ...
}
```

9.5. `CqlTemplate`

The `CqlTemplate` class is the central class in the core CQL package. It handles the creation and release of resources. It performs the basic tasks of the core CQL workflow, such as statement creation and execution, and leaves application code to provide CQL and extract results. The `CqlTemplate` class executes CQL queries and update statements, performs iteration over `ResultSet` instances and extraction of returned parameter values. It also catches CQL exceptions and translates them to the generic, more informative, exception hierarchy defined in the `org.springframework.dao` package.

When you use the `CqlTemplate` for your code, you need only implement callback interfaces, which have a clearly defined contract. Given a `Connection`, the `PreparedStatementCreator` callback interface creates a prepared statement with the provided CQL and any necessary parameter arguments. The `RowCallbackHandler` interface extracts values from each row of a `ResultSet`.

The `CqlTemplate` can be used within a DAO implementation through direct instantiation with a `SessionFactory` reference or be configured in the Spring container and given to DAOs as a bean reference. `CqlTemplate` is a foundational building block for `CassandraTemplate`.

All CQL issued by this class is logged at the `DEBUG` level under the category corresponding to the fully-qualified class name of the template instance (typically `CqlTemplate`, but it may be

different if you use a custom subclass of the `CqlTemplate` class).

You can control fetch size, consistency level, and retry policy defaults by configuring these parameters on the CQL API instances: `CqlTemplate`, `AsyncCqlTemplate`, and `ReactiveCqlTemplate`. Defaults apply if the particular query option is not set.

`CqlTemplate` comes in different execution model flavors. The basic `CqlTemplate` uses a blocking execution model. You can use `AsyncCqlTemplate` for asynchronous execution and synchronization with `ListenableFuture` instances or `ReactiveCqlTemplate` for reactive execution.

9.5.1. Examples of `CqlTemplate` Class Usage

This section provides some examples of the `CqlTemplate` class in action. These examples are not an exhaustive list of all of the functionality exposed by the `CqlTemplate`. See the Javadoc for that.

Querying (SELECT) with `CqlTemplate`

The following query gets the number of rows in a relation:

```
int rowCount = cqlTemplate.queryForObject("SELECT COUNT(*) FROM t_actor", Integer.class);
```

The following query uses a bind variable:

```
int countOfActorsNamedJoe = cqlTemplate.queryForObject(
        "SELECT COUNT(*) FROM t_actor WHERE first_name = ?", Integer.class, "Joe");
```

The following example queries for a `String`:

```
String lastName = cqlTemplate.queryForObject(
        "SELECT last_name FROM t_actor WHERE id = ?",
        String.class, 1212L);
```

The following example queries and populates a single domain object:

```
Actor actor = cqlTemplate.queryForObject(
        "SELECT first_name, last_name FROM t_actor WHERE id = ?",
        new RowMapper<Actor>() {
            public Actor mapRow(Row row, int rowNum) {
                Actor actor = new Actor();
                actor.setFirstName(row.getString("first_name"));
                actor.setLastName(row.getString("last_name"));
                return actor;
            },
      new Object[]{1212L},
        });
```

The following example queries and populates multiple domain objects:

```
List<Actor> actors = cqlTemplate.query(
        "SELECT first_name, last_name FROM t_actor",
        new RowMapper<Actor>() {
            public Actor mapRow(Row row int rowNum) {
                Actor actor = new Actor();
                actor.setFirstName(row.getString("first_name"));
                actor.setLastName(row.getString("last_name"));
                return actor;
            }
        });
```

If the last two snippets of code actually existed in the same application, it would make sense to remove the duplication present in the two `RowMapper` anonymous inner classes and extract them out into a single class (typically a `static` nested class) that can then be referenced by DAO methods.

For example, it might be better to write the last code snippet as follows:

```
public List<Actor> findAllActors() {
    return cqlTemplate.query("SELECT first_name, last_name FROM t_actor",
ActorMapper.INSTANCE);
}
```

```
enum ActorMapper implements RowMapper<Actor> {

    INSTANCE;

    public Actor mapRow(Row row, int rowNum) {
        Actor actor = new Actor();
        actor.setFirstName(row.getString("first_name"));
        actor.setLastName(row.getString("last_name"));
        return actor;
    }
}
```

`INSERT`, `UPDATE`, and `DELETE` with `CqlTemplate`

You can use the `execute(…)` method to perform `INSERT`, `UPDATE`, and `DELETE` operations. Parameter values are usually provided as variable arguments or, alternatively, as an object array.

The following example shows how to perform an `INSERT` operation with `CqlTemplate`:

```
cqlTemplate.execute(
        "INSERT INTO t_actor (first_name, last_name) VALUES (?, ?)",
        "Leonor", "Watling");
```

The following example shows how to perform an `UPDATE` operation with `CqlTemplate`:

```
cqlTemplate.execute(
        "UPDATE t_actor SET last_name = ? WHERE id = ?",
        "Banjo", 5276L);
```

The following example shows how to perform an `DELETE` operation with `CqlTemplate`:

```
cqlTemplate.execute(
        "DELETE FROM actor WHERE id = ?",
        Long.valueOf(actorId));
```

Other `CqlTemplate` operations

You can use the `execute(..)` method to execute any arbitrary CQL. As a result, the method is often used for DDL statements. It is heavily overloaded with variants that take callback interfaces, bind variable arrays, and so on.

The following example shows how to create and drop a table by using different API objects that are all passed to the `execute()` methods:

```
cqlOperations.execute("CREATE TABLE test_table (id uuid primary key, event text)");

DropTableSpecification dropper = DropTableSpecification.dropTable("test_table");
String cql = DropTableCqlGenerator.toCql(dropper);

cqlTemplate.execute(cql);
```

9.6. 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. Spring Data 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 to let you write portable and descriptive exception handling code without resorting to coding against and handling specific Cassandra exceptions. All of Spring's data access exceptions are inherited from the `DataAccessException` class, so you can be sure that you can catch all database-related exceptions within a single try-catch block.

9.7. Controlling Cassandra Connections

Applications connect to Apache Cassandra by using `Cluster` and `Session` objects. A Cassandra `Session` keeps track of multiple connections to the individual nodes and is designed to be a thread-safe, long-lived object. Usually, you can use a single `Session` for the whole application.

Spring acquires a Cassandra `Session` through a `SessionFactory`. `SessionFactory` is part of Spring Data for Apache Cassandra and is a generalized connection factory. It lets the container or framework hide connection handling and routing issues from the application code.

The following example shows how to configure a default `SessionFactory`:

```
Session session = … // get a Cassandra Session

CqlTemplate template = new CqlTemplate();

template.setSessionFactory(new DefaultSessionFactory(session));
```

`CqlTemplate` and other Template API implementations obtain a `Session` for each operation. Due to their long-lived nature, sessions are not closed after invoking the desired operation. Responsibility for proper resource disposal lies with the container or framework that uses the session.

You can find various `SessionFactory` implementations within the `org.springframework.data.cassandra.core.cql.session` package.

9.8. Introduction to `CassandraTemplate`

The `CassandraTemplate` class, located in the `org.springframework.data.cassandra` package, is the central class in Spring's Cassandra support and provides 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 rows in Cassandra tables.

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

The mapping between rows in Cassandra and application 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 custom converter. See the section on Cassandra conversion for more detailed information.

The `CassandraTemplate` class implements the `CassandraOperations` interface. In as much as possible, the methods on `CassandraOperations` are named after methods available in Cassandra to make the API familiar to developers who are already familiar with Cassandra.

For example, you can 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 between the two APIs is that `CassandraOperations` can be passed domain objects instead of CQL and query objects.

The preferred way to reference operations on a `CassandraTemplate` instance is through the `CassandraOperations` interface.

The default converter implementation used by `CassandraTemplate` is `MappingCassandraConverter`. While `MappingCassandraConverter` can use additional metadata to specify the mapping of objects to rows, it can also convert 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, are 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. See the section on exception translation for more information.

The Template API has different execution model flavors. The basic `CassandraTemplate` uses a blocking (imperative-synchronous) execution model. You can use `AsyncCassandraTemplate` for asynchronous execution and synchronization with `ListenableFuture` instances or `ReactiveCassandraTemplate` for reactive execution.

9.8.1. Instantiating CassandraTemplate

`CassandraTemplate` should always be configured as a Spring bean, although we show an example earlier where you can instantiate it directly. However, because we are assuming the context of making a Spring module, we assume the presence of the Spring container.

There are two ways to get a `CassandraTemplate`, depending on how you load you Spring `ApplicationContext`:

- Autowiring
- Bean Lookup with `ApplicationContext`

Autowiring

You can autowire a `CassandraOperations` into your project, as the following example shows:

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

As with all Spring autowiring, this assumes there is only one bean of type `CassandraOperations` in the `ApplicationContext`. If you have multiple `CassandraTemplate` beans (which is the case if you work with multiple keyspaces in the same project), then you can use the `@Qualifier` annotation to designate the bean you want to autowire.

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

Bean Lookup with `ApplicationContext`

You can also look up the `CassandraTemplate` bean from the `ApplicationContext`, as shown in the following example:

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

9.9. 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.

9.9.1. 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 continues to function without requiring changes. See CQL data types and "Data Mapping and Type Conversion" for the current type mapping matrix.

9.9.2. Methods for Inserting and Updating rows

`CassandraTemplate` has several convenient methods for saving and inserting your objects. To have more fine-grained control over the conversion process, you can register Spring

`Converter` instances with the `MappingCassandraConverter` (for example, `Converter<Row, Person>`).

The difference between insert and update operations is that `INSERT` operations do not insert `null` values.

The simple case of using the `INSERT` operation is to save a POJO. In this case, the table name is determined by the simple class name (not the fully qualified class name). The table to store the object can be overridden by using mapping metadata.

When inserting or updating, the `id` property must be set. Apache Cassandra has no means to generate an ID.

The following example uses the save operation and retrieves its contents:

Example 63. Inserting and retrieving objects by using the `CassandraTemplate`

```
import static org.springframework.data.cassandra.core.query.Criteria.where;
import static org.springframework.data.cassandra.core.query.Query.query;
…

Person bob = new Person("Bob", 33);
cassandraTemplate.insert(bob);

Person queriedBob = cassandraTemplate.selectOneById(query(where("age").is(33)),
Person.class);
```

You can use the following operations to insert and save:

- `void` **insert** `(Object objectToSave)` : Inserts the object in an Apache Cassandra table.
- `WriteResult` **insert** `(Object objectToSave, InsertOptions options)` : Inserts the object in an Apache Cassandra table and applies `InsertOptions` .

You can use the following update operations:

- `void` **update** `(Object objectToSave)` : Updates the object in an Apache Cassandra table.

- `WriteResult update (Object objectToSave, UpdateOptions options)` : Updates the object in an Apache Cassandra table and applies `UpdateOptions` .

You can also use the old fashioned way and write your own CQL statements, as the following example shows:

```
String cql = "INSERT INTO person (age, name) VALUES (39, 'Bob')";

cassandraTemplate().getCqlOperations().execute(cql);
```

You can also configure additional options such as TTL, consistency level, and lightweight transactions when using `InsertOptions` and `UpdateOptions` .

Which Table Are My Rows Inserted into?

You can manage the table name that is used for operating on the tables in two ways. The default table name is the simple class name changed to start with a lower-case letter. So, an instance of the `com.example.Person` class would be stored in the `person` table. The second way is to specify a table name in the `@Table` annotation.

Inserting, Updating, and Deleting Individual Objects in a Batch

The Cassandra protocol supports inserting a collection of rows in one operation by using a batch.

The following methods in the `CassandraTemplate` interface support this functionality:

- `batchOps` : Creates a new `CassandraBatchOperations` to populate the batch.

`CassandraBatchOperations`

- `insert` : Takes a single object, an array (var-args), or an `Iterable` of objects to insert.
- `update` : Takes a single object, an array (var-args), or an `Iterable` of objects to update.
- `delete` : Takes a single object, an array (var-args), or an `Iterable` of objects to delete.
- `withTimestamp` : Applies a TTL to the batch.
- `execute` : Executes the batch.

9.9.3. Updating Rows in a Table

For updates, you can select to update a number of rows.

The following example shows updating a single account object by adding a one-time $50.00 bonus to the balance with the `+` assignment:

Example 64. Updating rows using `CasandraTemplate`

```
import static org.springframework.data.cassandra.core.query.Criteria.where;
import org.springframework.data.cassandra.core.query.Query;
import org.springframework.data.cassandra.core.query.Update;

...

boolean applied = cassandraTemplate.update(Query.query(where("id").is("foo")),
  Update.create().increment("balance", 50.00), Account.class);
```

In addition to the `Query` discussed earlier, we provide the update definition by using an `Update` object. The `Update` class has methods that match the update assignments available for Apache Cassandra.

Most methods return the `Update` object to provide a fluent API for code styling purposes.

Methods for Executing Updates for Rows

The update method can update rows, as follows:

- `boolean` **update** `(Query query, Update update, Class<?> entityClass)` : Updates a selection of objects in the Apache Cassandra table.

Methods for the Update class

The `Update` class can be used with a little 'syntax sugar', as its methods are meant to be chained together. Also, you can kick-start the creation of a new `Update` instance with the static method `public static Update update(String key, Object value)` and by using static imports.

The `Update` class has the following methods:

- `AddToBuilder` **addTo** `(String columnName)` `AddToBuilder` entry-point:
  - Update `prepend(Object value)` : Prepends a collection value to the existing collection by using the `+` update assignment.
  - Update `prependAll(Object… values)` : Prepends all collection values to the existing collection by using the `+` update assignment.

- Update `append(Object value)` : Appends a collection value to the existing collection by using the `+` update assignment.
- Update `append(Object… values)` : Appends all collection values to the existing collection by using the `+` update assignment.
- Update `entry(Object key, Object value)` : Adds a map entry by using the `+` update assignment.
- Update `addAll(Map<? extends Object, ? extends Object> map)` : Adds all map entries to the map by using the `+` update assignment.

- `Update` **remove** `(String columnName, Object value)` : Removes the value from the collection by using the `-` update assignment.
- `Update` **clear** `(String columnName)` : Clears the collection.
- `Update` **increment** `(String columnName, Number delta)` : Updates by using the `+` update assignment.
- `Update` **decrement** `(String columnName, Number delta)` : Updates by using the `-` update assignment.
- `Update` **set** `(String columnName, Object value)` : Updates by using the `=` update assignment.
- `SetBuilder` **set** `(String columnName)` `SetBuilder` entry-point:
  - Update `atIndex(int index).to(Object value)` : Sets a collection at the given index to a value using the `=` update assignment.
  - Update `atKey(String object).to(Object value)` : Sets a map entry at the given key to a value the `=` update assignment.

The following listing shows a few update examples:

```
// UPDATE … SET key = 'Spring Data';
Update.update("key", "Spring Data")

// UPDATE … SET key[5] = 'Spring Data';
Update.empty().set("key").atIndex(5).to("Spring Data");

// UPDATE … SET key = key + ['Spring', 'DATA'];
Update.empty().addTo("key").appendAll("Spring", "Data");
```

Note that `Update` is immutable once created. Invoking methods creates new immutable (intermediate) `Update` objects.

9.9.4. Methods for Removing Rows

You can use the following overloaded methods to remove an object from the database:

- `boolean` **delete** `(Query query, Class<?> entityClass)` : Deletes the objects selected by `Query` .
- `T` **delete** `(T entity)` : Deletes the given object.
- `T` **delete** `(T entity, QueryOptions queryOptions)` : Deletes the given object applying `QueryOptions` .
- `boolean` **deleteById** `(Object id, Class<?> entityClass)` : Deletes the object using the given Id.

9.10. Querying Rows

You can express your queries by using the `Query` and `Criteria` classes, which have method names that reflect the native Cassandra predicate operator names, such as `lt` , `lte` , `is` , and others.

The `Query` and `Criteria` classes follow a fluent API style so that you can easily chain together multiple method criteria and queries while having easy-to-understand code. Static imports are used in Java when creating `Query` and `Criteria` instances to improve readability.

9.10.1. Querying Rows in a Table

In earlier sections, we saw how to retrieve a single object by using the `selectOneById` method on `CassandraTemplate` . Doing so returns a single domain object. We can also query for a collection of rows to be returned as a list of domain objects. Assuming we have a number of `Person` objects with name and age values stored as rows in a table and that each person has an account balance, we can now run a query by using the following code:

Example 65. Querying for rows using `CassandraTemplate`

```
import static org.springframework.data.cassandra.core.query.Criteria.where;
import static org.springframework.data.cassandra.core.query.Query.query;

…

List<Person> result = cassandraTemplate.select(query(where("age").is(50))
  .and(where("balance").gt(1000.00d)).withAllowFiltering(), Person.class);
```

The `select`, `selectOneById`, and `stream` methods take a `Query` object as a parameter. This object defines the criteria and options used to perform the query. The criteria is specified by using a `Criteria` object that has a static factory method named `where` that instantiates a new `Criteria` object. We recommend using a static import for `org.springframework.data.cassandra.core.query.Criteria.where` and `Query.query`, to make the query more readable.

This query should return a list of `Person` objects that meet the specified criteria. The `Criteria` class has the following methods that correspond to the operators provided in Apache Cassandra:

Methods for the Criteria class

- `CriteriaDefinition` **gt** `(Object value)`: Creates a criterion by using the `>` operator.
- `CriteriaDefinition` **gte** `(Object value)`: Creates a criterion by using the `>=` operator.
- `CriteriaDefinition` **in** `(Object… values)`: Creates a criterion by using the `IN` operator for a varargs argument.
- `CriteriaDefinition` **in** `(Collection<?> collection)`: Creates a criterion by using the `IN` operator using a collection.
- `CriteriaDefinition` **is** `(Object value)`: Creates a criterion by using field matching (`column = value`).
- `CriteriaDefinition` **lt** `(Object value)`: Creates a criterion by using the `<` operator.
- `CriteriaDefinition` **lte** `(Object value)`: Creates a criterion by using the ⇐ operator.
- `CriteriaDefinition` **like** `(Object value)`: Creates a criterion by using the `LIKE` operator.
- `CriteriaDefinition` **contains** `(Object value)`: Creates a criterion by using the `CONTAINS` operator.
- `CriteriaDefinition` **containsKey** `(Object key)`: Creates a criterion by using the `CONTAINS KEY` operator.

`Criteria` is immutable once created.

Methods for the Query class

The `Query` class has some additional methods that you can use to provide options for the query:

- `Query` **by** `(CriteriaDefinition… criteria)` : Used to create a `Query` object.
- `Query` **and** `(CriteriaDefinition criteria)` : Used to add additional criteria to the query.
- `Query` **columns** `(Columns columns)` : Used to define columns to be included in the query results.
- `Query` **limit** `(long limit)` : Used to limit the size of the returned results to the provided limit (used for paging).
- `Query` **pageRequest** `(Pageable pageRequest)` : Used to associate `Sort` , `PagingState` , and `fetchSize` with the query (used for paging).
- `Query` **pagingState** `(PagingState pagingState)` : Used to associate a `PagingState` with the query (used for paging).
- `Query` **queryOptions** `(QueryOptions queryOptions)` : Used to associate `QueryOptions` with the query.
- `Query` **sort** `(Sort sort)` : Used to provide a sort definition for the results.
- `Query` **withAllowFiltering** `()` : Used to render `ALLOW FILTERING` queries.

`Query` is immutable once created. Invoking methods creates new immutable (intermediate) `Query` objects.

9.10.2. Methods for Querying for Rows

The `Query` class has the following methods that return rows:

- `List<T>` **select** `(Query query, Class<T> entityClass)` : Query for a list of objects of type `T` from the table.
- `T` **selectOneById** `(Query query, Class<T> entityClass)` : Query for a single object of type `T` from the table.
- `Slice<T>` **slice** `(Query query, Class<T> entityClass)` : Starts or continues paging by querying for a `Slice` of objects of type `T` from the table.
- `T` **selectOne** `(Query query, Class<T> entityClass)` : Query for a single object of type `T` from the table.
- `Stream<T>` **stream** `(Query query, Class<T> entityClass)` : Query for a stream of objects of type `T` from the table.
- `List<T>` **select** `(String cql, Class<T> entityClass)` : Ad-hoc query for a list of objects of type `T` from the table by providing a CQL statement.

- `T` **selectOneById** `(String cql, Class<T> entityClass)` : Ad-hoc query for a single object of type `T` from the table by providing a CQL statement.
- `Stream<T>` **stream** `(String cql, Class<T> entityClass)` : Ad-hoc query for a stream of objects of type `T` from the table by providing a CQL statement.

The query methods must specify the target type `T` that is returned.

9.10.3. Fluent Template API

The `CassandraOperations` interface is one of the central components when it comes to more low-level interaction with Apache Cassandra. It offers a wide range of methods. You can find multiple overloads for every method. Most of them cover optional (nullable) parts of the API.

`FluentCassandraOperations` provide a more narrow interface for common methods of `CassandraOperations` providing a more readable, fluent API. The entry points (`query(…)` , `insert(…)` , `update(…)` , and `delete(…)`) follow a natural naming scheme based on the operation to execute. Moving on from the entry point, the API is designed to offer only context-dependent methods that guide the developer towards a terminating method that invokes the actual `CassandraOperation` . The following example shows the fluent API:

```
List<SWCharacter> all = ops.query(SWCharacter.class)
  .inTable("star_wars")                                    1
  .all();
```

1 Skip this step if `SWCharacter` defines the table name with `@Table` or if using the class name as the table name is not a problem.

If a table in Cassandra holds entities of different types, such as a `Jedi` within a Table of `SWCharacters` , you can use different types to map the query result. You can use `as(Class<?> targetType)` to map results to a different target type, while `query(Class<?> entityType)` still applies to the query and table name. The following example uses the `query` and `as` methods:

```
List<Jedi> all = ops.query(SWCharacter.class)      1
  .as(Jedi.class)                                  2
  .matching(query(where("jedi").is(true)))
  .all();
```

1 The query fields are mapped against the `SWCharacter` type.

2 Resulting rows are mapped into `Jedi`.

You can directly apply Projections to resulting documents by providing only the `interface` type through `as(Class<?>)`.

The terminating methods (`first()`, `one()`, `all()`, and `stream()`) handle switching between retrieving a single entity and retrieving multiple entities as `List` or `Stream` and similar operations.

The new fluent template API methods (that is, `query(..)`, `insert(..)`, `update(..)`, and `delete(..)`) use effectively thread-safe supporting objects to compose the CQL statement. However, it comes at the added cost of additional young-gen JVM heap overhead, since the design is based on final fields for the various CQL statement components and construction on mutation. You should be careful when possibly inserting or deleting a large number of objects (such as inside of a loop, for instance).

9.11. Overriding Default Mapping with Custom Converters

To have more fine-grained control over the mapping process, you can register Spring `Converters` with `CassandraConverter` implementations, such as `MappingCassandraConverter`.

`MappingCassandraConverter` first checks to see whether any Spring `Converters` 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 need to create an implementation of the Spring `Converter` interface and register it with the `MappingCassandraConverter`.

For more information on Spring's type conversion service, see the reference docs here.

9.11.1. Saving by Using a Registered Spring Converter

You can combine converting and saving in a single process, basically using the converter to do the saving.

The following example uses a `Converter` to convert a `Person` object to a `java.lang.String` with Jackson 2:

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

import org.springframework.util.StringUtils;
import com.fasterxml.jackson.databind.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);
    }
  }
}
```

9.11.2. Reading by Using a Spring Converter

Similar to how you can combine saving and converting, you can also combine reading and converting.

The following example uses a `Converter` that converts a `java.lang.String` into a `Person` object with Jackson 2:

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

import org.springframework.util.StringUtils;
import com.fasterxml.jackson.databind.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;
  }
}
```

9.11.3. Registering Spring Converters with `CassandraConverter`

Spring Data for Apache Cassandra Java configuration provides a convenient way to register Spring `Converter` instances: `MappingCassandraConverter`. The following configuration snippet shows how to manually register converters as well as configure `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...
}
```

9.11.4. Converter Disambiguation

Generally, we inspect the `Converter` implementations for both the source and target types they convert from and to. Depending on whether one of those is a type Cassandra can handle natively, Spring Data registers the `Converter` instance as a reading or a writing converter.

Consider 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> { … }
```

If you implement a `Converter` whose source and target types are native Cassandra types, Spring Data cannot determine whether we should consider it as a reading or a writing `Converter`. Registering the `Converter` instance as both might lead to unwanted results.

For example, 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 generally be able to force the infrastructure to register a `Converter` for one way only, we provide `@ReadingConverter` as well as `@WritingConverter` to indicate the appropriate `Converter` implementation.

---

10. Reactive Cassandra Support

The reactive Cassandra support contains a wide range of features:

- Spring configuration support using Java-based `@Configuration` classes.
- `ReactiveCqlTemplate` helper class that increases productivity by properly handling common Cassandra data access operations.
- `ReactiveCassandraTemplate` helper class that increases productivity by using `ReactiveCassandraOperations` in a reactive manner. It includes integrated object mapping between tables and POJOs.
- Exception translation into Spring's portable Data Access Exception Hierarchy.
- Feature rich object mapping integrated with Spring's Conversion Service.
- Java-based Query, Criteria, and Update DSLs.
- Automatic implementation of `Repository` interfaces, including support for custom finder methods.

For most data-oriented tasks, you can use the `ReactiveCassandraTemplate` or the repository support, which use the rich object mapping functionality. `ReactiveCqlTemplate` is commonly used to increment counters or perform ad-hoc CRUD operations. `ReactiveCqlTemplate` also provides callback methods that make it easy to get low-level API objects, such as `com.datastax.driver.core.Session`, which let you 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 immediately familiar and so that you can map your existing knowledge onto the Spring APIs.

10.1. Getting Started

Spring Data for Apache Cassandra requires Apache Cassandra 2.1 or later and Datastax Java Driver 3.0 or later. An easy way to quickly set up and bootstrap a working environment is to create a Spring-based project in STS or use Spring Initializer.

First, you need to set up a running Apache Cassandra server. See the Apache Cassandra Quick Start Guide for an explanation on how to start 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 and press Yes when prompted. Then enter a project and a package name, such as `org.spring.data.cassandra.example`.

Then you can add the following dependency declaration to your pom.xml file's `dependencies` section.

```
<dependencies>

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-cassandra</artifactId>
    <version>2.1.1.RELEASE</version>
  </dependency>

</dependencies>
```

Also, you should change the version of Spring in the pom.xml file to be as follows:

```
<spring.framework.version>5.1.1.RELEASE</spring.framework.version>
```

If using a milestone release instead of a GA release, you also need to add the location of the Spring Milestone repository for Maven to your pom.xml file so that it is at the same level of your `<dependencies/>` element, as follows:

```
<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 all Spring repositories here.

Now you can create a simple Java application that stores and reads a domain object to and from Cassandra.

To do so, first create a simple domain object class to persist, as the following example shows:

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

import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.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, as the following example shows:

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

import java.util.UUID;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.cassandra.core.ReactiveCassandraOperations;
import org.springframework.data.cassandra.core.ReactiveCassandraTemplate;
import org.springframework.data.cassandra.core.query.Criteria;
import org.springframework.data.cassandra.core.query.Query;

import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Session;

import reactor.core.publisher.Mono;

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");

    ReactiveCassandraOperations template = new ReactiveCassandraTemplate(new
DefaultBridgedReactiveSession(session));

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

    jonDoe.flatMap(it ->
template.selectOne(Query.query(Criteria.where("id").is(it.getId())), Person.class))
        .doOnNext(it -> LOGGER.info(it.toString()))
        .then(template.truncate(Person.class))
        .block();

    session.close();
    cluster.close();
  }
}
```

Even in this simple example, there are a few notable things to point out:

- A fully synchronous flow does not benefit from a reactive infrastructure, because a reactive programming model requires synchronization.
- You can create an instance of `ReactiveCassandraTemplate` with a Cassandra `Session` obtained from `Cluster`.
- You must annotate your POJO as a Cassandra `@Table` and annotate the `@PrimaryKey`. Optionally, you can override these mapping names to match your Cassandra database table and column names.
- You can either use raw CQL or the DataStax `QueryBuilder` API to construct your queries.

10.2. Examples Repository

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

10.3. Connecting to Cassandra with Spring

One of the first tasks when using Apache Cassandra with Spring is to create a `com.datastax.driver.core.Session` object by using the Spring IoC container. You can do so either by using Java-based bean metadata or by using XML-based bean metadata. These are discussed in the following sections.

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.

10.3.1. Registering a Session instance using Java-based metadata

You can configure Reactive Cassandra support by using Java Configuration classes. Reactive Cassandra support adapts a `Session` to provide a reactive execution model on top of an asynchronous driver.

A reactive `Session` is configured similarly to an imperative `Session`. We provide supporting configuration classes that come with predefined defaults and require only environment-

specific information to configure Spring Data for Apache Cassandra. The base class for reactive support is `AbstractReactiveCassandraConfiguration`. This configuration class extends the imperative `AbstractCassandraConfiguration`, so the reactive support also configures the imperative API support. The following example shows how to register Apache Cassandra beans in a configuration class:

Example 66. Registering Spring Data for Apache Cassandra beans using `AbstractReactiveCassandraConfiguration`

```
@Configuration
public class AppConfig extends AbstractReactiveCassandraConfiguration {

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

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

The configuration class in the preceding example is schema-management-enabled to create CQL objects during startup. See Schema Management for further details.

10.4. `ReactiveCqlTemplate`

The `ReactiveCqlTemplate` class is the central class in the core CQL package. It handles the creation and release of resources. It performs the basic tasks of the core CQL workflow, such as statement creation and execution, leaving application code to provide CQL and extract results. The `ReactiveCqlTemplate` class executes CQL queries and update statements and performs iteration over `ResultSet` instances and extraction of returned parameter values. It also catches CQL exceptions and translates them into the generic, more informative, exception hierarchy defined in the `org.springframework.dao` package.

When you use the `ReactiveCqlTemplate` in your code, you need only implement callback interfaces, which have a clearly defined contract. Given a `Connection`, the `ReactivePreparedStatementCreator` callback interface creates a prepared statement with the

provided CQL and any necessary parameter arguments. The `RowCallbackHandler` interface extracts values from each row of a `ReactiveResultSet`.

The `ReactiveCqlTemplate` can be used within a DAO implementation through direct instantiation with a `ReactiveSessionFactory` reference or be configured in the Spring container and given to DAOs as a bean reference. `ReactiveCqlTemplate` is a foundational building block for `ReactiveCassandraTemplate`.

All CQL issued by this class is logged at the `DEBUG` level under the category corresponding to the fully-qualified class name of the template instance (typically `ReactiveCqlTemplate`, but it may be different if you use a custom subclass of the `ReactiveCqlTemplate` class).

10.4.1. Examples of `ReactiveCqlTemplate` Class Usage

This section provides some examples of `ReactiveCqlTemplate` class usage. These examples are not an exhaustive list of all of the functionality exposed by the `ReactiveCqlTemplate`. See the attendant Javadocs for that.

Querying (SELECT) with `ReactiveCqlTemplate`

The following query gets the number of rows in a relation:

```
Mono<Integer> rowCount = reactiveCqlTemplate.queryForObject("SELECT COUNT(*) FROM
t_actor", Integer.class);
```

The following query uses a bind variable:

```
Mono<Integer> countOfActorsNamedJoe = reactiveCqlTemplate.queryForObject(
        "SELECT COUNT(*) FROM t_actor WHERE first_name = ?", Integer.class, "Joe");
```

The following example queries for a `String`:

```
Mono<String> lastName = reactiveCqlTemplate.queryForObject(
        "SELECT last_name FROM t_actor WHERE id = ?",
        String.class, 1212L);
```

The following exmaple queries and populates a single domain object:

```
Mono<Actor> actor = reactiveCqlTemplate.queryForObject(
        "SELECT first_name, last_name FROM t_actor WHERE id = ?",
        new RowMapper<Actor>() {
            public Actor mapRow(Row row, int rowNum) {
                Actor actor = new Actor();
                actor.setFirstName(row.getString("first_name"));
                actor.setLastName(row.getString("last_name"));
                return actor;
            },
        new Object[]{1212L},
        });
```

The following example queries and populates a number of domain objects:

```
Flux<Actor> actors = reactiveCqlTemplate.query(
        "SELECT first_name, last_name FROM t_actor",
        new RowMapper<Actor>() {
            public Actor mapRow(Row row int rowNum) {
                Actor actor = new Actor();
                actor.setFirstName(row.getString("first_name"));
                actor.setLastName(row.getString("last_name"));
                return actor;
            }
        });
```

If the last two snippets of code actually existed in the same application, it would make sense to remove the duplication present in the two `RowMapper` anonymous inner classes and extract them into a single class (typically a `static` nested class) that can then be referenced by DAO methods as needed.

For example, it might be better to write the last code snippet as follows:

```
public Flux<Actor> findAllActors() {
    return reactiveCqlTemplate.query("SELECT first_name, last_name FROM t_actor",
ActorMapper.INSTANCE);
}

enum ActorMapper implements RowMapper<Actor> {
```

```
    INSTANCE;

    public Actor mapRow(Row row, int rowNum) {
        Actor actor = new Actor();
        actor.setFirstName(row.getString("first_name"));
        actor.setLastName(row.getString("last_name"));
        return actor;
    }
}
```

`INSERT`, `UPDATE`, and `DELETE` with `ReactiveCqlTemplate`

You can use the `execute(…)` method to perform `INSERT`, `UPDATE`, and `DELETE` operations. Parameter values are usually provided as variable arguments or, alternatively, as an object array.

The following example shows how to perform an `INSERT` operation with `ReactiveCqlTemplate`:

```
Mono<Boolean> applied = reactiveCqlTemplate.execute(
        "INSERT INTO t_actor (first_name, last_name) VALUES (?, ?)",
        "Leonor", "Watling");
```

The following example shows how to perform an `UPDATE` operation with `ReactiveCqlTemplate`:

```
Mono<Boolean> applied = reactiveCqlTemplate.execute(
        "UPDATE t_actor SET last_name = ? WHERE id = ?",
        "Banjo", 5276L);
```

The following example shows how to perform an `DELETE` operation with `ReactiveCqlTemplate`:

```
Mono<Boolean> applied = reactiveCqlTemplate.execute(
        "DELETE FROM actor WHERE id = ?",
        Long.valueOf(actorId));
```

10.5. 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. Spring Data 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 to let you write portable and descriptive exception handling code without resorting to coding against and handling specific Cassandra exceptions. All of Spring's data access exceptions are inherited from the `DataAccessException` class, so you can be sure that you can catch all database-related exceptions within a single try-catch block.

`ReactiveCqlTemplate` and `ReactiveCassandraTemplate` propagate exceptions as early as possible. Exceptions that occur during execution of the reactive sequence are emitted as error signals.

10.6. Introduction to `ReactiveCassandraTemplate`

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

Once configured, `ReactiveCassandraTemplate` is thread-safe and can be reused across multiple instances.

The mapping between rows in a Cassandra table 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 custom converter. See "Mapping" for more detailed information.

The `ReactiveCassandraTemplate` class implements the `ReactiveCassandraOperations` interface. As often as possible, the methods names `ReactiveCassandraOperations` match names in Cassandra to make the API familiar to developers who are familiar with Cassandra.

For example, you can 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 `ReactiveCassandraOperations`. A major difference between the two APIs is that `ReactiveCassandraOperations` can be passed domain objects instead of CQL and query objects.

The preferred way to reference operations on a `ReactiveCassandraTemplate` instance is through its interface, `ReactiveCassandraOperations`.

The default converter implementation for `ReactiveCassandraTemplate` is `MappingCassandraConverter`. While the `MappingCassandraConverter` can make use of additional metadata to specify the mapping of objects to rows, it can also convert objects that contain no additional metadata by using conventions for the mapping of fields and table names. These conventions, as well as the use of mapping annotations, are explained in "Mapping".

Another central feature of `CassandraTemplate` is exception translation. Exceptions thrown by the Cassandra Java driver are translated into Spring's portable Data Access Exception hierarchy. See "Exception Translation" for more information.

10.6.1. Instantiating `ReactiveCassandraTemplate`

`ReactiveCassandraTemplate` should always be configured as a Spring bean, although an earlier example showed how to instantiate it directly. However, this section assumes that the template is used in a Spring module, so it also assumes that the Spring container is being used.

There are two ways to get a `ReactiveCassandraTemplate`, depending on how you load you Spring `ApplicationContext`:

- Autowiring
- Bean Lookup with `ApplicationContext`

Autowiring

You can autowire a `ReactiveCassandraTemplate` into your project, as the following example shows:

```
@Autowired
private ReactiveCassandraOperations reactiveCassandraOperations;
```

Like all Spring autowiring, this assumes there is only one bean of type `ReactiveCassandraOperations` in the `ApplicationContext`. If you have multiple `ReactiveCassandraTemplate` beans (which can be the case if you are working with multiple keyspaces in the same project), then you can use the `@Qualifier` annotation to designate which bean you want to autowire.

```
@Autowired
@Qualifier("keyspaceTwoTemplateBeanId")
private ReactiveCassandraOperations reactiveCassandraOperations;
```

Bean Lookup with `ApplicationContext`

You can also look up the `ReactiveCassandraTemplate` bean from the `ApplicationContext`, as shown in the following example:

```
ReactiveCassandraOperations reactiveCassandraOperations =
applicationContext.getBean("reactiveCassandraOperations",
ReactiveCassandraOperations.class);
```

10.7. Saving, Updating, and Removing Rows

`ReactiveCassandraTemplate` provides a simple way for you to save, update, and delete your domain objects and map those objects to tables managed in Cassandra.

10.7.1. Methods for Inserting and Updating rows

`CassandraTemplate` has several convenient methods for saving and inserting your objects. To have more fine-grained control over the conversion process, you can register Spring `Converter` instances with the `MappingCassandraConverter` (for example, `Converter<Row, Person>`).

> The difference between insert and update operations is that `INSERT`

operations do not insert `null` values.

The simple case of using the `INSERT` operation is to save a POJO. In this case, the table name is determined by the simple class name (not the fully qualified class name). The table to store the object can be overridden by using mapping metadata.

When inserting or updating, the `id` property must be set. Apache Cassandra has no means to generate an ID.

The following example uses the save operation and retrieves its contents:

Example 67. Inserting and retrieving objects by using the `CassandraTemplate`

```
import static org.springframework.data.cassandra.core.query.Criteria.where;
import static org.springframework.data.cassandra.core.query.Query.query;
…

Person bob = new Person("Bob", 33);
cassandraTemplate.insert(bob);

Mono<Person> queriedBob =
reactiveCassandraTemplate.selectOneById(query(where("age").is(33)), Person.class);
```

You can use the following operations to insert and save:

- `void` **insert** `(Object objectToSave)` : Inserts the object in an Apache Cassandra table.
- `WriteResult` **insert** `(Object objectToSave, InsertOptions options)` : Inserts the object in an Apache Cassandra table and applies `InsertOptions` .

You can use the following update operations:

- `void` **update** `(Object objectToSave)` : Updates the object in an Apache Cassandra table.
- `WriteResult` **update** `(Object objectToSave, UpdateOptions options)` : Updates the object in an Apache Cassandra table and applies `UpdateOptions` .

You can also use the old fashioned way and write your own CQL statements, as the following example shows:

```
String cql = "INSERT INTO person (age, name) VALUES (39, 'Bob')";

Mono<Boolean> applied = reactiveCassandraTemplate.getReactiveCqlOperations().execute(cql);
```

You can also configure additional options such as TTL, consistency level, and lightweight transactions when using `InsertOptions` and `UpdateOptions`.

Which Table Are My Rows Inserted into?

You can manage the table name that is used for operating on the tables in two ways. The default table name is the simple class name changed to start with a lower-case letter. So, an instance of the `com.example.Person` class would be stored in the `person` table. The second way is to specify a table name in the `@Table` annotation.

10.7.2. Updating Rows in a Table

For updates, you can select to update a number of rows.

The following example shows updating a single account object by adding a one-time $50.00 bonus to the balance with the `+` assignment:

Example 68. Updating rows using `CasandraTemplate`

```
import static org.springframework.data.cassandra.core.query.Criteria.where;
import org.springframework.data.cassandra.core.query.Query;
import org.springframework.data.cassandra.core.query.Update;

...

Mono<Boolean> wasApplied =
reactiveCassandraTemplate.update(Query.query(where("id").is("foo")),
  Update.create().increment("balance", 50.00), Account.class);
```

In addition to the `Query` discussed earlier, we provide the update definition by using an `Update` object. The `Update` class has methods that match the update assignments available for Apache Cassandra.

Most methods return the `Update` object to provide a fluent API for code styling purposes.

For more detail, see “Methods for Executing Updates for Rows”.

11. Cassandra Repositories

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". You should understand the basic concepts explained there before proceeding.

11.1. Usage

To access domain entities stored in Apache Cassandra, you can use Spring Data's sophisticated repository support, which significantly eases implementing DAOs. To do so, create an interface for your repository, as the following example shows:

Example 69. Sample Person entity

```
@Table
public class Person {

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

  // … getters and setters omitted
}
```

Note that the entity has a property named `id` of type `String`. The default serialization mechanism used in `CassandraTemplate` (which backs the repository support) regards properties named `id` as being the row ID.

The following example shows a repository definition to persist `Person` entities:

Example 70. Basic repository interface to persist `Person` entities

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

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

Right now, the interface in the preceding example serves only typing purposes, but we add additional methods to it later.

Next, in your Spring configuration, add the following (if you use XML for configuration):

Example 71. General Cassandra repository Spring 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: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 causes the base packages to be scanned for interfaces that extend `CrudRepository` and create Spring beans for each one found. By default, the repositories are 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 want to use Java configuration, use the `@EnableCassandraRepositories` annotation. The annotation carries the same attributes as the namespace element. If no base package is configured, the infrastructure scans the package of the annotated configuration class. The following example shows how to use the `@EnableCassandraRepositories` annotation:

Example 72. Java configuration for repositories

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

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

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

Because our domain repository extends `CrudRepository`, it provides you with basic CRUD operations. Working with the repository instance is a matter of injecting the repository as a dependency into a client, as the following example does by autowiring `PersonRepository`:

Example 73. Basic access to Person entities

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

    @Autowired PersonRepository repository;

    @Test
    public void readsPersonTableCorrectly() {

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

Cassandra repositories support paging and sorting for paginated and sorted access to the entities. Cassandra paging requires a paging state to forward-only navigate through pages. A `Slice` keeps track of the current paging state and allows for creation of a `Pageable` to request the next page. The following example shows how to set up paging access to `Person` entities:

Example 74. Paging access to `Person` entities

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

    @Autowired PersonRepository repository;

    @Test
    public void readsPagesCorrectly() {

      Slice<Person> firstBatch = repository.findAll(CassandraPageRequest.first(10));

      assertThat(firstBatch).hasSize(10);

      Page<Person> nextBatch = repository.findAll(firstBatch.nextPageable());

      // …
    }
}
```

Cassandra repositories do not extend `PagingAndSortingRepository`, because classic paging patterns using limit/offset are not applicable to Cassandra.

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

11.2. 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 a matter of declaring a method on the repository interface. The following example shows a number of such method declarations:

Example 75. PersonRepository with query methods

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

    List<Person> findByLastname(String lastname);                    1
```

```
    Slice<Person> findByFirstname(String firstname, Pageable pageRequest);   2

    List<Person> findByFirstname(String firstname, QueryOptions opts);       3

    List<Person> findByFirstname(String firstname, Sort sort);               4

    Person findByShippingAddress(Address address);                           5

    Person findFirstByShippingAddress(Address address);                      6

    Stream<Person> findAllBy();                                              7

    @AllowFiltering
    List<Person> findAllByAge(int age);                                      8
}
```

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

2 Applies pagination to a query. You can equip your method signature with a `Pageable` parameter and let the method return a `Slice` instance, and we automatically page the query accordingly.

3 Passing a `QueryOptions` object applies the query options to the resulting query before its execution.

4 Applies dynamic sorting to a query. You can add a `Sort` parameter to your method signature, and Spring Data automatically applies ordering to the query.

5 Shows that you can query based on properties that are not a primitive type by using `Converter` instances registered in `CustomConversions`. Throws `IncorrectResultSizeDataAccessException` if more than one match is found.

6 Uses the `First` keyword to restrict the query to only the first result. Unlike the preceding method, this method does not throw an exception if more than one match is found.

7 Uses a Java 8 `Stream` to read and convert individual elements while iterating the stream.

8 Shows a query method annotated with `@AllowFiltering`, to allow server-side filtering.

Querying non-primary key properties requires secondary indexes.

The following table shows short examples of the keywords that you can use in query methods:

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)`
`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`

11.2.1. 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 create projections based on certain attributes of those types. Spring Data allows modeling dedicated return types, to more selectively retrieve partial views of the managed aggregates.

Imagine a repository and aggregate root type such as the following example:

Example 76. 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 that we want to retrieve the person's name attributes only. What means does Spring Data offer to achieve this? The rest of this chapter answers that question.

Interface-based Projections

The easiest way to limit the result of the queries to only the name attributes is by declaring an interface that exposes accessor methods for the properties to be read, as shown in the following example:

Example 77. 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. Doing so lets a query method be added as follows:

Example 78. 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 creates proxy instances of that interface at runtime for each element returned and forwards calls to the exposed methods to the target object.

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

Example 79. 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 is obtained and wrapped into a projecting proxy in turn.

Closed Projections

A projection interface whose accessor methods all match properties of the target aggregate is considered to be a closed projection. The following example (which we used earlier in this chapter, too) is a closed projection:

Example 80. A closed projection

```
interface NamesOnly {

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

If you use a closed projection, Spring Data can optimize the query execution, because we know about all the attributes that are needed to back the projection proxy. For more details on that, see 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, as shown in the following example:

Example 81. An Open Projection

```
interface NamesOnly {

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

The aggregate root backing the projection is available in the `target` variable. A projection interface using `@Value` is an open projection. Spring Data cannot apply query execution optimizations in this case, because the SpEL expression could use any attribute of the aggregate root.

The expressions used in `@Value` should not be too complex — you want to avoid programming in `String` variables. For very simple expressions, one option might be to resort to default methods (introduced in Java 8), as shown in the following example:

Example 82. 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 invoke that from the SpEL expression, as shown in the following example:

Example 83. Sample Person object

```
@Component
class MyBean {

  String getFullName(Person person) {
    …
  }
}

interface NamesOnly {

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

Notice how the SpEL expression refers to `myBean` and invokes the `getFullName(…)` method and forwards the projection target as a 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 through an `Object` array named `args`. The following example shows how to get a method parameter from the `args` array:

Example 84. Sample Person object

```
interface NamesOnly {

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

Again, for more complex expressions, you should use a Spring bean and let the expression invoke a method, as described earlier.

Class-based Projections (DTOs)

Another way of defining projections is by using value type DTOs (Data Transfer Objects) that hold properties for the fields that are supposed to be retrieved. These DTO types can be used in exactly the same way projection interfaces are used, except that no proxying happens and no nested projections can be applied.

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

The following example shows a projecting DTO:

Example 85. 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
}
```

Avoid boilerplate code for projection DTOs

You can dramatically simplify the code for a DTO by using Project Lombok, which provides an `@Value` annotation (not to be confused with Spring's `@Value` annotation shown in the earlier interface examples). If you use Project

Lombok's `@Value` annotation, the sample DTO shown earlier would become the following:

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

Fields are `private final` by default, and the class exposes a constructor that takes 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, you might want to select the type to be used at invocation time (which makes it dynamic). To apply dynamic projections, use a query method such as the one shown in the following example:

Example 86. A repository using a dynamic projection parameter

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

  <T> 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, as shown in the following example:

Example 87. 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);
}
```

11.2.2. Query Options

You can specify query options for query methods by passing a `QueryOptions` object. The options apply to the query before the actual query execution. `QueryOptions` is treated as a non-query parameter and is not considered to be a query parameter value. Query options apply to derived and string `@Query` repository methods.

To statically set the consistency level, use the `@Consistency` annotation on query methods. The declared consistency level is applied to the query each time it is executed. The following example sets the consistency level to `ConsistencyLevel.LOCAL_ONE`:

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

    @Consistency(ConsistencyLevel.LOCAL_ONE)
    List<Person> findByLastname(String lastname);

    List<Person> findByFirstname(String firstname, QueryOptions options);
}
```

The DataStax Cassandra documentation includes a good discussion of the available consistency levels.

You can control fetch size, consistency level, and retry policy defaults by configuring the following parameters on the CQL API instances: `CqlTemplate`, `AsyncCqlTemplate`, and `ReactiveCqlTemplate`. Defaults apply if the particular query option is not set.

11.2.3. 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. To activate it, drop 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`, as the following examlpe shows:

```
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 picks up `CassandraOperations` as a CDI bean and creates a proxy for a Spring Data repository whenever a bean of a repository type is requested by the container. Thus, obtaining an instance of a Spring Data repository is a matter of declaring an injected property, as the following example shows:

```
class RepositoryClient {

  @Inject
  PersonRepository repository;

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

12. Reactive Cassandra Repositories

This chapter outlines the specialties handled by the reactive repository support for Apache Cassandra. It builds on the core repository infrastructure explained in Cassandra Repositories, so you should have a good understanding of the basic concepts explained there.

Reactive usage is broken up into two phases: Composition and Execution.

Calling repository methods lets you compose a reactive sequence by obtaining `Publisher` instances and applying operators. No I/O happens until you subscribe. Passing the reactive sequence to a reactive execution infrastructure, such as Spring WebFlux or Vert.x), subscribes to the publisher and initiate the actual execution. See the Project reactor documentation for more detail.

12.1. Reactive Composition Libraries

The reactive space offers various reactive composition libraries. The most common libraries are RxJava and Project Reactor.

Spring Data for Apache Cassandra is built on top of the DataStax Cassandra Driver. The driver is not reactive but the asynchronous capabilities allow us to adopt and expose the `Publisher` APIs to provide maximum interoperability by relying on the Reactive Streams initiative. Static APIs, such as `ReactiveCassandraOperations` , are provided by using Project Reactor's `Flux` and `Mono` types. Project Reactor offers various adapters to convert reactive wrapper types (`Flux` to `Observable` and back), but conversion can easily clutter your code.

Spring Data's repository abstraction is a dynamic API that is mostly defined by you and your requirements as you declare query methods. Reactive Cassandra repositories can be

implemented by using either RxJava or Project Reactor wrapper types by extending from one of the library-specific repository interfaces:

- `ReactiveCrudRepository`
- `ReactiveSortingRepository`
- `RxJava2CrudRepository`
- `RxJava2SortingRepository`

Spring Data converts reactive wrapper types behind the scenes so that you can stick to your favorite composition library.

12.2. Usage

To access domain entities stored in Apache Cassandra, you can use Spring Data's sophisticated repository support, which significantly eases implementing DAOs. To do so, create an interface for your repository, as the following example shows:

Example 88. Sample Person entity

```
@Table
public class Person {

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

  // … getters and setters omitted
}
```

Note that the entity has a property named `id` of type `String`. The default serialization mechanism used in `CassandraTemplate` (which backs the repository support) regards properties named `id` as being the row ID.

The following example shows a repository definition to persist `Person` entities:

Example 89. Basic repository interface to persist `Person` entities

```
public interface ReactivePersonRepository extends ReactiveSortingRepository<Person, Long>
{
```

```
  Flux<Person> findByFirstname(String firstname);                                  1

  Flux<Person> findByFirstname(Publisher<String> firstname);                       2

  Mono<Person> findByFirstnameAndLastname(String firstname, String lastname);      3

  Mono<Person> findFirstByFirstname(String firstname);                             4

  @AllowFiltering
  Flux<Person> findByAge(int age);                                                 5
}
```

1. A query for all people with the given `firstname`. The query is derived by parsing the method name for constraints, which can be concatenated with `And` and `Or`. Thus, the method name results in a query expression of `SELECT * FROM person WHERE firstname = :firstname`.
2. A query for all people with the given `firstname` once the `firstname` is emitted from the given `Publisher`.
3. Find a single entity for the given criteria. Completes with `IncorrectResultSizeDataAccessException` on non-unique results.
4. Unlike the preceding query, the first entity is always emitted even if the query yields more result rows.
5. A query method annotated with `@AllowFiltering`, which allows server-side filtering.

For Java configuration, use the `@EnableReactiveCassandraRepositories` annotation. The annotation carries the same attributes as the corresponding XML namespace element. If no base package is configured, the infrastructure scans the package of the annotated configuration class. The following example uses the `@EnableReactiveCassandraRepositories` annotation:

Example 90. Java configuration for repositories

```
@Configuration
@EnableReactiveCassandraRepositories
class ApplicationConfig extends AbstractReactiveCassandraConfiguration {

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

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

Since our domain repository extends `ReactiveSortingRepository`, it provides you with CRUD operations as well as methods for sorted access to the entities. Working with the repository instance is a matter of dependency injecting it into a client, as the following example shows:

Example 91. Sorted access to Person entities

```
public class PersonRepositoryTests {

    @Autowired ReactivePersonRepository repository;

    @Test
    public void sortsElementsCorrectly() {
        Flux<Person> people = repository.findAll(Sort.by(new Order(ASC, "lastname")));
    }
}
```

Cassandra repositories support paging and sorting for paginated and sorted access to the entities. Cassandra paging requires a paging state to forward-only navigate through pages. A `Slice` keeps track of the current paging state and allows for creation of a `Pageable` to request the next page. The following example shows how to set up paging access to `Person` entities:

Example 92. Paging access to `Person` entities

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

    @Autowired PersonRepository repository;

    @Test
    public void readsPagesCorrectly() {

      Mono<Slice<Person>> firstBatch =
repository.findAll(CassandraPageRequest.first(10));

      Mono<Slice<Person>> nextBatch = firstBatch.flatMap(it ->
repository.findAll(it.nextPageable()));
```

```
        // …
    }
}
```

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

12.3. Features

Spring Data's Reactive Cassandra support comes with the same set of features as the support for imperative repositories.

It supports the following features:

- Query Methods that use String queries and Query Derivation
- Projections

Query methods must return a reactive type. Resolved types (`User` versus `Mono<User>`) are not supported.

13. Mapping

Rich object 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 by using annotations on your domain objects. However, the infrastructure is not limited to using annotations as the only source of metadata. The `MappingCassandraConverter` also lets you map domain objects to tables without providing any additional metadata, by following a set of conventions.

In this chapter, we 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.

13.1. Object Mapping Fundamentals

This section covers the fundamentals of Spring Data object mapping, object creation, field and property access, mutability and immutability. Note, that this section only applies to Spring Data modules that do not use the object mapping of the underlying data store (like JPA). Also be sure to consult the store-specific sections for store-specific object mapping, like indexes, customizing column or field names or the like.

Core responsibility of the Spring Data object mapping is to create instances of domain objects and map the store-native data structures onto those. This means we need two fundamental steps:

1. Instance creation by using one of the constructors exposed.
2. Instance population to materialize all exposed properties.

13.1.1. Object creation

Spring Data automatically tries to detect a persistent entity's constructor to be used to materialize objects of that type. The resolution algorithm works as follows:

1. If there's a no-argument constructor, it will be used. Other constructors will be ignored.
2. If there's a single constructor taking arguments, it will be used.
3. If there are multiple constructors taking arguments, the one to be used by Spring Data will have to be annotated with `@PersistenceConstructor`.

The value resolution assumes constructor argument names to match the property names of the entity, i.e. the resolution will be performed as if the property was to be populated, including all customizations in mapping (different datastore column or field name etc.). This also requires either parameter names information available in the class file or an `@ConstructorProperties` annotation being present on the constructor.

The value resolution can be customized by using Spring Framework's `@Value` value annotation using a store-specific SpEL expression. Please consult the section on store specific mappings for further details.

Object creation internals

To avoid the overhead of reflection, Spring Data object creation uses a factory class generated at runtime by default, which will call the domain classes constructor directly. I.e. for this example type:

```
class Person {
  Person(String firstname, String lastname) { … }
}
```

we will create a factory class semantically equivalent to this one at runtime:

```
class PersonObjectInstantiator implements ObjectInstantiator {

  Object newInstance(Object... args) {
    return new Person((String) args[0], (String) args[1]);
  }
}
```

This gives us a roundabout 10% performance boost over reflection. For the domain class to be eligible for such optimization, it needs to adhere to a set of constraints:

- it must not be a private class
- it must not be a non-static inner class
- it must not be a CGLib proxy class
- the constructor to be used by Spring Data must not be private

If any of these criteria match, Spring Data will fall back to entity instantiation via reflection.

13.1.2. Property population

Once an instance of the entity has been created, Spring Data populates all remaining persistent properties of that class. Unless already populated by the entity's constructor (i.e. consumed through its constructor argument list), the identifier property will be populated first to allow the resolution of cyclic object references. After that, all non-transient properties that have not already been populated by the constructor are set on the entity instance. For that we use the following algorithm:

1. If the property is immutable but exposes a wither method (see below), we use the wither to create a new entity instance with the new property value.

2. If property access (i.e. access through getters and setters) is defined, we're invoking the setter method.
3. By default, we set the field value directly.

Property population internals

Similarly to our optimizations in object construction we also use Spring Data runtime generated accessor classes to interact with the entity instance.

```
class Person {

  private final Long id;
  private String firstname;
  private @AccessType(Type.PROPERTY) String lastname;

  Person() {
    this.id = null;
  }

  Person(Long id, String firstname, String lastname) {
    // Field assignments
  }

  Person withId(Long id) {
    return new Person(id, this.firstname, this.lastame);
  }

  void setLastname(String lastname) {
    this.lastname = lastname;
  }
}
```

Example 93. A generated Property Accessor

```
class PersonPropertyAccessor implements PersistentPropertyAccessor {

  private static final MethodHandle firstname;              2

  private Person person;                                    1

  public void setProperty(PersistentProperty property, Object value) {

    String name = property.getName();

    if ("firstname".equals(name)) {
      firstname.invoke(person, (String) value);             2
    } else if ("id".equals(name)) {
```

```
      this.person = person.withId((Long) value);              3
    } else if ("lastname".equals(name)) {
      this.person.setLastname((String) value);                4
    }
  }
}
```

1 PropertyAccessor's hold a mutable instance of the underlying object. This is, to enable mutations of otherwise immutable properties.

2 By default, Spring Data uses field-access to read and write property values. As per visibility rules of `private` fields, `MethodHandles` are used to interact with fields.

3 The class exposes a `withId(…)` method that's used to set the identifier, e.g. when an instance is inserted into the datastore and an identifier has been generated. Calling `withId(…)` creates a new `Person` object. All subsequent mutations will take place in the new instance leaving the previous untouched.

4 Using property-access allows direct method invocations without using `MethodHandles`.

This gives us a roundabout 25% performance boost over reflection. For the domain class to be eligible for such optimization, it needs to adhere to a set of constraints:

- Types must not reside in the default or under the `java` package.
- Types and their constructors must be `public`
- Types that are inner classes must be `static`.
- The used Java Runtime must allow for declaring classes in the originating `ClassLoader`. Java 9 and newer impose certain limitations.

By default, Spring Data attempts to use generated property accessors and falls back to reflection-based ones if a limitation is detected.

Let's have a look at the following entity:

Example 94. A sample entity

```
class Person {

  private final @Id Long id;                                                  1
  private final String firstname, lastname;                                   2
  private final LocalDate birthday;
  private final int age;  3

  private String comment;                                                     4
  private @AccessType(Type.PROPERTY) String remarks;                          5

  static Person of(String firstname, String lastname, LocalDate birthday) {  6

    return new Person(null, firstname, lastname, birthday,
      Period.between(birthday, LocalDate.now()).getYears());
  }

  Person(Long id, String firstname, String lastname, LocalDate birthday, int age) {  6

    this.id = id;
    this.firstname = firstname;
    this.lastname = lastname;
    this.birthday = birthday;
    this.age = age;
  }

  Person withId(Long id) {                                                    1
    return new Person(id, this.firstname, this.lastname, this.birthday);
  }

  void setRemarks(String remarks) {                                           5
    this.remarks = remarks;
  }
}
```

1 The identifier property is final but set to `null` in the constructor. The class exposes a `withId(…)` method that's used to set the identifier, e.g. when an instance is inserted into the datastore and an identifier has been generated. The original `Person` instance stays unchanged as a new one is created. The same pattern is usually applied for other properties that are store managed but might have to be changed for persistence operations.

2 The `firstname` and `lastname` properties are ordinary immutable properties potentially exposed through getters.

3 The `age` property is an immutable but derived one from the `birthday` property. With the design shown, the database value will trump the defaulting as Spring Data uses the

only declared constructor. Even if the intent is that the calculation should be preferred, it's important that this constructor also takes `age` as parameter (to potentially ignore it) as otherwise the property population step will attempt to set the age field and fail due to it being immutable and no wither being present.

4 The `comment` property is mutable is populated by setting its field directly.

5 The `remarks` properties are mutable and populated by setting the `comment` field directly or by invoking the setter method for

6 The class exposes a factory method and a constructor for object creation. The core idea here is to use factory methods instead of additional constructors to avoid the need for constructor disambiguation through `@PersistenceConstructor`. Instead, defaulting of properties is handled within the factory method.

13.1.3. General recommendations

- *Try to stick to immutable objects* — Immutable objects are straightforward to create as materializing an object is then a matter of calling its constructor only. Also, this avoids your domain objects to be littered with setter methods that allow client code to manipulate the objects state. If you need those, prefer to make them package protected so that they can only be invoked by a limited amount of co-located types. Constructor-only materialization is up to 30% faster than properties population.
- *Provide an all-args constructor* — Even if you cannot or don't want to model your entities as immutable values, there's still value in providing a constructor that takes all properties of the entity as arguments, including the mutable ones, as this allows the object mapping to skip the property population for optimal performance.
- *Use factory methods instead of overloaded constructors to avoid* `@PersistenceConstructor` — With an all-argument constructor needed for optimal performance, we usually want to expose more application use case specific constructors that omit things like auto-generated identifiers etc. It's an established pattern to rather use static factory methods to expose these variants of the all-args constructor.
- *Make sure you adhere to the constraints that allow the generated instantiator and property accessor classes to be used* —
- *For identifiers to be generated, still use a final field in combination with a wither method* —
- *Use Lombok to avoid boilerplate code* — As persistence operations usually require a constructor taking all arguments, their declaration becomes a tedious repetition of boilerplate parameter to field assignments that can best be avoided by using Lombok's `@AllArgsConstructor`.

13.1.4. Kotlin support

Spring Data adapts specifics of Kotlin to allow object creation and mutation.

Kotlin object creation

Kotlin classes are supported to be instantiated , all classes are immutable by default and require explicit property declarations to define mutable properties. Consider the following `data class Person` :

```
data class Person(val id: String, val name: String)
```

The class above compiles to a typical class with an explicit constructor. We can customize this class by adding another constructor and annotate it with `@PersistenceConstructor` to indicate a constructor preference:

```
data class Person(var id: String, val name: String) {

    @PersistenceConstructor
    constructor(id: String) : this(id, "unknown")
}
```

Kotlin supports parameter optionality by allowing default values to be used if a parameter is not provided. When Spring Data detects a constructor with parameter defaulting, then it leaves these parameters absent if the data store does not provide a value (or simply returns `null`) so Kotlin can apply parameter defaulting. Consider the following class that applies parameter defaulting for `name`

```
data class Person(var id: String, val name: String = "unknown")
```

Every time the `name` parameter is either not part of the result or its value is `null` , then the `name` defaults to `unknown` .

Property population of Kotlin data classes

In Kotlin, all classes are immutable by default and require explicit property declarations to define mutable properties. Consider the following `data class Person`:

```
data class Person(val id: String, val name: String)
```

This class is effectively immutable. It allows to create new instances as Kotlin generates a `copy(…)` method that creates new object instances copying all property values from the existing object and applying property values provided as arguments to the method.

13.2. Data Mapping and Type Conversion

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

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 custom converters to adjust type conversion. See "Overriding Mapping with Explicit Converters" for further details. The following table maps Spring Data types to Cassandra types:

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`

Type	Cassandra types
`BigInteger`	`varint`
`BigDecimal`	`decimal`
`java.util.Date`	`timestamp`
`com.datastax.driver.core.LocalDate`	`date`
`InetAddress`	`inet`
`ByteBuffer`	`blob`
`java.util.UUID`	`timeuuid`
`TupleValue`, mapped Tuple Types	`tuple<…>`
`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`
`LocalTime`+ (Joda, Java 8, JSR310-BackPort)	`time`
`LocalDateTime`, `LocalTime`, `Instant` (Joda, Java 8, JSR310-BackPort)	`timestamp`
`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`, as the following example shows:

Example 95. Enum mapping to numeric types

```
@Table
public class EnumToOrdinalMapping {

  @PrimaryKey String id;

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

public enum Condition {
  NEW, USED
}
```

13.3. 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 simple (short) Java class name is mapped to the table name by being changed to lower case. For example, `com.bigbank.SavingsAccount` maps to a table named `savingsaccount`.
- The converter uses any registered Spring `Converter` instances 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.

13.3.1. 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` to tell it where to scan the classpath at startup for your domain classes to extract metadata and construct indexes.

Also, by creating your own instance, you can register Spring `Converter` instances to use for mapping specific classes to and from the database. The following example configuration class sets up Cassandra mapping support:

Example 96. @Configuration class to configure Cassandra mapping support

```
@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 named `getEntityBasePackages(…)`. You can override it to tell the converter where to scan for classes annotated with the `@Table` annotation.

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

`AbstractCassandraConfiguration` creates a `CassandraTemplate` instance and registers it with the container under the name of `cassandraTemplate`.

13.4. 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 domain objects with the `@Table` annotation. Doing so lets the classpath scanner find and pre-process your domain objects to extract the necessary metadata. Only annotated entities are used to perform schema actions.

In the worst case, a `SchemaAction.RECREATE_DROP_UNUSED` operation drops your tables and you lose your data. The following example shows a simple domain object:

Example 97. 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;
}
```

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.

13.4.1. Working with Primary Keys

Cassandra requires at least one partition key field for a CQL table. A table can additionally declare 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 Keys

A simple primary key consists of one partition key field within an entity class. Since it is one field only, we safely can assume it is a partition key. The following listing shows a CQL table defined in Cassandra with a primary key of `user_id`:

Example 98. CQL Table defined in Cassandra

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

The following example shows a Java class annotated such that it corresponds to the Cassandra defined in the previous listing:

Example 99. 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

}
```

Composite Keys

Composite primary keys (or compound keys) consist of more than one primary key field. 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:

- Embedded in an entity.
- By using `@PrimaryKeyClass`.

The simplest form of a composite key is a key with one partition key and one clustering key.

The following example shows a CQL statement to represent the table and its composite key:

Example 100. 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 Keys

Flat composite primary keys are embedded inside the entity as flat fields. Primary key fields are annotated with `@PrimaryKeyColumn`. Selection requires either a query to contain predicates for the individual fields or the use of `MapId`. The following example shows a class with a flat composite primary key:

Example 101. 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
}
```

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 is annotated with `@PrimaryKeyClass` and must define `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 entity's identity in a single complex object. The following example shows a composite primary key class:

Example 102. 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
}
```

The following example shows how to use a composite primary key:

Example 103. 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
}
```

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

13.4.2. Mapping Annotation Overview

The `MappingCassandraConverter` can use metadata to drive the mapping of objects to rows in a Cassandra table. An overview of the annotations follows:

- `@Id` : Applied at the field or property level to mark the property used for identity purposes.
- `@Table` : Applied at the class level to indicate that this class is a candidate for mapping to the database. You can specify the name of the table where the object is stored.
- `@PrimaryKey` : Similar to `@Id` but lets you specify the column name.
- `@PrimaryKeyColumn` : Cassandra-specific annotation for primary key columns that lets you specify primary key column attributes, such as for clustered or partitioned. Can be used on single and multiple attributes to indicate either a single or a composite (compound) primary key.
- `@PrimaryKeyClass` : Applied at the class level to indicate that this class is a compound primary key class. Must be referenced with `@PrimaryKey` in the entity class.
- `@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 is represented in the Cassandra table, thus letting the name differ from the field name of the class.
- `@Indexed` : Applied at the field level. Describes the index to be created at session initialization.
- `@SASI` : Applied at the field level. Allows SASI index creation during session initialization.
- `@CassandraType` : Applied at the field level to specify a Cassandra data type. Types are derived from the property 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.
- `@Tuple` : Applied at the type level to use a type as a mapped tuple.
- `@Element` : Applied at the field level to specify element or field ordinals within a mapped tuple. Types are derived from the property declaration by default.

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

The following example shows a more complex mapping:

Example 104. 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)
  @Indexed
  private String lastName;

  private Address address;

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

  private Coordinates coordinates;

  @Transient
  private Integer accountTotal;

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

  private Map<@Indexed 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
}
```

The following example shows how to map a UDT `Address`:

Example 105. Mapped User-Defined Type `Address`

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

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

  private String city;

  private Set<String> zipcodes;

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

  // other getters/setters ommitted
}
```

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

The following example shows how map a tuple:

Example 106. Mapped Tuple

```
@Tuple
public class Coordinates {

  @Element(0)
  @CassandraType(type = Name.VARCHAR)
  private String description;

  @Element(1)
  private long longitude;

  @Element(2)
  private long latitude;

  // other getters/setters ommitted
}
```

Index Creation

You can annotate particular entity properties with `@Indexed` or `@SASI` if you wish to create secondary indexes on application startup. Index creation creates simple secondary indexes for scalar types, user-defined types, and collection types.

You can configure a SASI Index to apply an analyzer, such as `StandardAnalyzer` or `NonTokenizingAnalyzer` (by using `@StandardAnalyzed` and `@NonTokenizingAnalyzed`, respectively).

Map types distinguish between `ENTRY`, `KEYS`, and `VALUES` indexes. Index creation derives the index type from the annotated element. The following example shows a number of ways to create an index:

Example 107. Variants of map indexing

```
@Table
public class Person {

  @Id
  private String key;

  @SASI @StandardAnalyzed
  private String names;

  @Indexed("indexed_map")
  private Map<String, String> entries;

  private Map<@Indexed String, String> keys;
```

```
  private Map<String, @Indexed String> values;

  // …
}
```

Index creation on session initialization may have a severe performance impact on application startup.

13.4.3. Overriding Mapping with Explicit Converters

When storing and querying objects, it is often 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 let you selectively handle the conversion for a particular type. Other times, you may want to optimize performance.

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

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

The following example of a Spring `Converter` implementation converts from a row to a `Person` POJO:

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

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

13.5. Lifecycle Events

The Cassandra mapping framework has several built-in `org.springframework.context.ApplicationEvent` events that your application can respond to by registering special beans in the `ApplicationContext`. Being based on Spring's application context event infrastructure lets other products, such as Spring Integration, easily receive these events as they are a well known eventing mechanism in Spring-based applications.

To intercept an object before it goes into the database, you can register a subclass of `org.springframework.data.cassandra.core.mapping.event.AbstractCassandraEventListener` that overrides the `onBeforeSave(…)` method. When the event is dispatched, your listener is called and passed the domain object (which is a Java entity). The following example uses the `onBeforeSave` method:

```
public class BeforeSaveListener extends AbstractCassandraEventListener<Person> {
  @Override
  public void onBeforeSave(BeforeSaveEvent<Person> event) {
    … change values, delete them, whatever …
  }
}
```

Declaring these beans in your Spring `ApplicationContext` will cause them to be invoked whenever the event is dispatched.

The `AbstractCassandraEventListener` has the following callback methods:

- `onBeforeSave`: Called in `CassandraTemplate.insert(…)` and `.update(…)` operations before inserting or updating a row in the database.
- `onAfterSave`: Called in `CassandraTemplate…insert(…)` and `.update(…)` operations after inserting or updating a row in the database.
- `onBeforeDelete`: Called in `CassandraTemplate.delete(…)` operations before deleting row from the database.
- `onAfterDelete`: Called in `CassandraTemplate.delete(…)` operations after deleting row from the database.
- `onAfterLoad`: Called in the `CassandraTemplate.select(…)`, `.slice(…)`, and `.stream(…)` methods after each row is retrieved from the database.

- `onAfterConvert` : Called in the `CassandraTemplate.select(…)` , `.slice(…)` , and `.stream(…)` methods after converting a row retrieved from the database to a POJO.

Lifecycle events are emitted only for root-level types. Complex types used as properties within an aggregate root are not subject to event publication.

Appendix

Appendix A: Namespace reference

The `<repositories />` Element

The `<repositories />` element triggers the setup of the Spring Data repository infrastructure. The most important attribute is `base-package` , which defines the package to scan for Spring Data repository interfaces. See "XML configuration". The following table describes the attributes of the `<repositories />` element:

Table 5. Attributes

Name	Description
`base-package`	Defines the package to be scanned for repository interfaces that extend `*Repository` (the actual interface is determined by the specific Spring Data module) in auto-detection mode. All packages below the configured package are scanned, too. Wildcards are allowed.
`repository-impl-postfix`	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix are 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` .

Name	Description
`named-queries-location`	Defines the location to search for a Properties file containing externally defined queries.
`consider-nested-repositories`	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.[1]

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 keywords listed here might not be supported in a particular store.

Table 7. Query keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`

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

Logical keyword	Keyword expressions
`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 types listed here might not be supported in a particular store.

Geospatial types (such as `GeoResult`, `GeoResults`, and `GeoPage`) are available only 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. If no result is found, `null` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.

Return type	Description
`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. If no result is found, `Optional.empty()` or `Optional.absent()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Option<T>`	Either a Scala or Javaslang `Option` type. Semantically the same behavior as Java 8's `Optional`, described earlier.
`Stream<T>`	A Java 8 `Stream`.
`Future<T>`	A `Future`. Expects a method to be annotated with `@Async` and requires Spring's asynchronous method execution capability to be enabled.
`CompletableFuture<T>`	A Java 8 `CompletableFuture`. Expects a method to be annotated with `@Async` and requires Spring's asynchronous method execution capability to be enabled.
`ListenableFuture`	A `org.springframework.util.concurrent.ListenableFuture`. Expects a method to be annotated with `@Async` and requires Spring's asynchronous method execution capability to be enabled.
`Slice`	A sized chunk of data with an indication of whether there is more data available. Requires a `Pageable` method parameter.
`Page<T>`	A `Slice` with additional information, such as the total number of results. Requires a `Pageable` method parameter.
`GeoResult<T>`	A result entry with additional information, such as the distance to a reference location.

Return type	Description
`GeoResults<T>`	A list of `GeoResult<T>` with additional information, such as the average distance to a reference location.
`GeoPage<T>`	A `Page` with `GeoResult<T>`, such as the average distance to a reference location.
`Mono<T>`	A Project Reactor `Mono` emitting zero or one element using reactive repositories. Expects the query method to return one result at most. If no result is found, `Mono.empty()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Flux<T>`	A Project Reactor `Flux` emitting zero, one, or many elements using reactive repositories. Queries returning `Flux` can emit also an infinite number of elements.
`Single<T>`	A RxJava `Single` emitting a single element using reactive repositories. Expects the query method to return one result at most. If no result is found, `Mono.empty()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Maybe<T>`	A RxJava `Maybe` emitting zero or one element using reactive repositories. Expects the query method to return one result at most. If no result is found, `Mono.empty()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Flowable<T>`	A RxJava `Flowable` emitting zero, one, or many elements using reactive repositories. Queries returning `Flowable` can emit also an infinite number of elements.

Appendix E: Migration Guides

Migration Guide from Spring Data Cassandra 1.x to 2.x

Spring Data for Apache Cassandra 2.0 introduces a set of breaking changes when upgrading from earlier versions:

- Merged the `spring-cql` and `spring-data-cassandra` modules into a single module.
- Separated asynchronous and synchronous operations in `CqlOperations` and `CassandraOperations` into dedicated interfaces and templates.
- Revised the `CqlTemplate` API to align with `JdbcTemplate`.
- Removed the `CassandraOperations.selectBySimpleIds` method.
- Used better names for `CassandraRepository`.
- Removed SD Cassandra `ConsistencyLevel` and `RetryPolicy` types in favor of DataStax `ConsistencyLevel` and `RetryPolicy` types.
- Refactored CQL specifications to value objects and configurators.
- Refactored `QueryOptions` to be immutable objects.
- Refactored `CassandraPersistentProperty` to single-column.

Deprecations

- Deprecated `QueryOptionsBuilder.readTimeout(long, TimeUnit)` in favor of `QueryOptionsBuilder.readTimeout(Duration)`.
- Deprecated `CustomConversions` in favor of `CassandraCustomConversions`.
- Deprecated `BasicCassandraMappingContext` in favor of `CassandraMappingContext`.
- Deprecated `o.s.d.c.core.cql.CachedPreparedStatementCreator` in favor of `o.s.d.c.core.cql.support.CachedPreparedStatementCreator`.
- Deprecated `CqlTemplate.getSession()` in favor of `getSessionFactory()`.
- Deprecated `CqlIdentifier.cqlId(…)` and `KeyspaceIdentifier.ksId(…)` in favor of the `.of(…)` methods.
- Deprecated constructors of `QueryOptions` in favor of their builders.
- Deprecated `TypedIdCassandraRepository` in favor of `CassandraRepository`

Merged Spring CQL and Spring Data Cassandra Modules

Spring CQL and Spring Data Cassandra are now merged into a single module. The standalone `spring-cql` module is no longer available. You can find all types merged into `spring-data-cassandra`. The following listing shows how to include `spring-data-cassandra` in your maven dependencies:

```
<dependencies>

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-cassandra</artifactId>
    <version>2.1.1.RELEASE</version>
  </dependency>

</dependencies>
```

With the merge, we merged all CQL packages into Spring Data Cassandra:

- Moved `o.s.d.cql` into `o.s.d.cassandra.core.cql`.
- Merged `o.s.d.cql` with `o.s.d.cassandra.config` and flattened the XML and Java subpackages.
- Moved `CassandraExceptionTranslator` and `CqlExceptionTranslator` to `o.s.d.c.core.cql`.
- Moved Cassandra exceptions `o.s.d.c.support.exception` to `o.s.d.cassandra`.
- Moved `o.s.d.c.convert` to `o.s.d.c.core.convert` (affects converters).
- Moved `o.s.d.c.mapping` to `o.s.d.c.core.mapping` (affects mapping annotations).
- Moved `MapId` from `o.s.d.c.repository` to `o.s.d.c.core.mapping`.

Revised `CqlTemplate` / `CassandraTemplate`

We split `CqlTemplate` and `CassandraTemplate` in three ways:

- `CassandraTemplate` is no longer a `CqlTemplate` but uses an instance that allows reuse and fine-grained control over fetch size, consistency levels, and retry policies. You can obtain the `CqlOperations` through `CassandraTemplate.getCqlOperations()`. Because of the change, dependency injection of `CqlTemplate` requires additional bean setup.
- `CqlTemplate` now reflects basic CQL operations instead of mixing high-level and low-level API calls (such as `count(…)` versus `execute(…)`) and the reduced method set is aligned with Spring Frameworks's `JdbcTemplate` with its convenient callback interfaces.
- Asynchronous methods are re-implemented on `AsyncCqlTemplate` and `AsyncCassandraTemplate` by using `ListenableFuture`. We removed `Cancellable` and the various async callback listeners. `ListenableFuture` is a flexible approach and allows transition into a `CompletableFuture`.

Removed `CassandraOperations.selectBySimpleIds()`

The method was removed because it did not support complex IDs. The newly introduced query DSL allows mapped and complex id's for single column Id's, as the following example shows:

```
cassandraTemplate.select(Query.query(Criteria.where("id").in(…)), Person.class)
```

Better names for `CassandraRepository`

We renamed `CassandraRepository` and `TypedIdCassandraRepository` to align Spring Data Cassandra naming with other Spring Data modules:

- Renamed `CassandraRepository` to `MapIdCassandraRepository`
- Renamed `TypedIdCassandraRepository` to `CassandraRepository`
- Introduced `TypedIdCassandraRepository`, extending `CassandraRepository` as a deprecated type to ease migration

Removed SD Cassandra `ConsistencyLevel` and `RetryPolicy` types in favor of DataStax `ConsistencyLevel` and `RetryPolicy` types

Spring Data Cassandra `ConsistencyLevel` and `RetryPolicy` have been removed. Please use the types provided by the DataStax driver.

The Spring Data Cassandra types restricted usage of available features provided in and allowed by the Cassandra native driver. As a result, the Spring Data Cassandra's types required an update each time newer functionality was introduced by the driver.

Refactored CQL Specifications to Value Objects and Configurators

As much as possible, CQL specification types are now value types (such as `FieldSpecification`, `AlterColumnSpecification`), and objects are constructed by static factory methods. This allows immutability for simple value objects. Configurator objects (such as `AlterTableSpecification`) that operate on mandatory properties (such as a table name or keyspace name) are initially constructed through a a static factory method and allow further configuration until the desired state is created.

Refactored `QueryOptions` to be Immutable Objects

`QueryOptions` and `WriteOptions` are now immutable and can be created through builders. Methods accepting `QueryOptions` enforce non-null objects, which are available from static `empty()` factory methods. The following example shows how to use `QueryOptions.builder()`:

```
QueryOptions queryOptions = QueryOptions.builder()
        .consistencyLevel(ConsistencyLevel.ANY)
        .retryPolicy(FallthroughRetryPolicy.INSTANCE)
        .readTimeout(Duration.ofSeconds(10))
        .fetchSize(10)
        .tracing(true)
        .build();
```

Refactored `CassandraPersistentProperty` to Single-column

This change affects You only if you operate directly on the mapping model.

`CassandraPersistentProperty` allowed previously multiple column names to be bound for composite primary key use. Columns of a `CassandraPersistentProperty` are now reduced to a single column. Resolved composite primary keys map to a class through `MappingContext.getRequiredPersistentEntity(…)`.

---

1. see XML configuration

Version 2.1.1.RELEASE
Last updated 2018-10-15 10:39:18 MESZ