Spring Data Cassandra - Reference Documentation

David Webb, Matthew Adams

Version 1.1.4.RELEASE
2015-10-14

Table of Contents

Preface

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

Chapter 1. Project Metadata

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

Introduction

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

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

Knowing Spring

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

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

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

Knowing NoSQL and Cassandra

NoSQL stores have taken the storage world by storm. It is a vast domain with a plethora of solutions, terms and patterns (to make things worth even the term itself has multiple meanings). While some of the principles are common, it is crucial that the user is familiar to some degree with the Cassandra Columnar NoSQL Datastore supported by DATACASS. The best way to get acquainted to this solutions is to read their documentation and follow their examples - it usually doesn't take more then 5-10 minutes

to go through them and if you are coming from an RDMBS-only background many times these exercises can be an eye opener.

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

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

The DataStax site offers commercial support and many resources.

Chapter 2. Requirements

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

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

Chapter 3. Additional Help Resources

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

3.1. Support

There are a few support options available:

3.1.1. Questions & Answers

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

- spring-data
- spring-data-cassandra

3.1.2. Professional Support

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

3.2. Following Development

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

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

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

Reference Documentation

Document Structure

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

Cassandra support introduces the Cassandra module feature set.

Cassandra repositories introduces the repository support for Cassandra.

Chapter 4. Cassandra support

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

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

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

4.1. Getting Started

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

First you need to set up a running Cassandra server.

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

Then add the following to pom.xml dependencies section.

```
<dependencies>

  <!-- other dependency elements omitted -->

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

</dependencies>
```

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

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

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

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

The repository is also browseable here.

Create a simple Employee class to persist.

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

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

@Table
public class Person {

 @PrimaryKey
 private String id;

 private String name;
 private int age;

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

 public String getId() {
  return id;
 }

 public String getName() {
  return name;
 }

 public int getAge() {
  return age;
 }

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

}
```

And a main application to run

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

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

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

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

public class CassandraApp {

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

 private static Cluster cluster;
 private static Session session;

 public static void main(String[] args) {

  try {

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

   session = cluster.connect("mykeyspace");

   CassandraOperations cassandraOps = new CassandraTemplate(session);

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

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

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

   cassandraOps.truncate("person");

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

 }
}
```

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

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

4.2. Examples Repository

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

4.3. Connecting to Cassandra with Spring

4.3.1. Externalize Connection Properties

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

We will call this cassandra.properties

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

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

4.3.2. XML Configuration

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

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

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

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

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

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

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

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

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

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

</beans>
```

4.3.3. Java Configuration

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

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

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

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

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

  @Autowired
  private Environment env;

  @Bean
  public CassandraClusterFactoryBean cluster() {

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

    return cluster;
  }

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

  @Bean
  public CassandraConverter converter() {
```

```
    return new MappingCassandraConverter(mappingContext());
  }

  @Bean
  public CassandraSessionFactoryBean session() throws Exception {

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

    return session;
  }

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

4.4. General auditing configuration

Auditing support is not available in the current version.

4.5. Introduction to CassandraTemplate

4.5.1. Instantiating CassandraTemplate

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

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

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

AutoWiring

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

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

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

Bean Lookup with ApplicationContext

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

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

4.6. Saving, Updating, and Removing Rows

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

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

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

Simplest Composite Key

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

CQL Table defined in Cassandra

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

Class defining the **Composite Primary Key**.

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

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

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

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

@PrimaryKeyClass
public class LoginEventKey implements Serializable {

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

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

  public String getPersonId() {
    return personId;
  }

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

  public Date getEventTime() {
    return eventTime;
  }

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

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

  @Override
  public boolean equals(Object obj) {
```

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

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

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

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

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

  @PrimaryKey
  private LoginEventKey pk;

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

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

  public LoginEventKey getPk() {
    return pk;
  }

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

  public int getEventCode() {
    return eventCode;
  }

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

  public String getIpAddress() {
    return ipAddress;
  }

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

}
```

Complex Composite Primary Key

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

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

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

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

@PrimaryKeyClass
public class DetailedLoginEventKey implements Serializable {

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

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

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

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

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

  ...

}
```

4.6.2. Type mapping

Spring Data Cassandra relies on the DataStax Java Driver type mapping component. This approach ensures that as types are added or changed, the Spring Data Cassandra module will continue to

function without requiring changes. For more information on the DataStax CQL3 to Java Type mappings, please see their Documentation here.

4.6.3. Methods for saving and inserting rows

Single records inserts

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

Insert a record with an annotated POJO.

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

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

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

cassandraOperations.execute(insert);
```

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

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

cassandraOperations.execute(cql);
```

Multiple inserts for high speed ingestion

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

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

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

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

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

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

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

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

4.6.4. Updating rows in a CQL table

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

Update a record with an annotated POJO.

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

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

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

cassandraOperations.execute(update);
```

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

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

cassandraOperations.execute(cql);
```

4.6.5. Methods for removing rows

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

Delete a record with an annotated POJO.

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

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

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

cassandraOperations.execute(delete);
```

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

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

cassandraOperations.execute(cql);
```

4.6.6. Methods for truncating tables

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

Truncate a table using the truncate() method.

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

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

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

cassandraOperations.execute(truncate);
```

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

```
String cql = "truncate person";

cassandraOperations.execute(cql);
```

4.7. Querying CQL Tables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.8. Overriding default mapping with custom converters

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

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

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

4.8.1. Saving using a registered Spring Converter

Coming Soon!

4.8.2. Reading using a Spring Converter

Coming Soon!

4.8.3. Registering Spring Converters with the CassandraConverter

Coming Soon!

4.8.4. Converter disambiguation

Coming Soon!

4.9. Executing Commands

4.9.1. Methods for executing commands

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

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

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

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

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

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

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

4.10. Exception Translation

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

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

Chapter 5. Cassandra repositories

5.1. Introduction

This chapter will point out the specialties for repository support for Cassandra. This builds on the core repository support explained in [repositories]. So make sure you've got a sound understanding of the basic concepts explained there.

5.2. Usage

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

TODO

5.3. Query methods

5.3.1. Repository delete queries

5.4. Miscellaneous

5.4.1. CDI Integration

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

```
class RepositoryClient {

  @Inject
  PersonRepository repository;

  public void businessMethod() {

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

Chapter 6. Mapping

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

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

6.1. Convention based Mapping

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

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

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

TODO

6.1.2. Mapping Configuration

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

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

Example 1. @Configuration class to configure Cassandra mapping support

```
TODO
```

Example 2. XML schema to configure Cassandra mapping support

```
TODO
```

Appendix