Oracle® Database
Spring Data SDK Developers Guide

Release 22.2

F58555-04

October 2022

ORACLE®

Contents

List of Figures

List of Tables

1
Oracle NoSQL Database SDK for Spring Data

Learn about how to access the Oracle NoSQL Database from the Spring Data Framework (Spring-based programming model for data).

Prerequisites:

This chapter assumes that the user has a good understanding of the following:

* Maven
* Spring Data Framework

About the Oracle NoSQL Database SDK for Spring Data

Connect to the Oracle NoSQL Database with applications using the Spring Data Framework (Spring-based programming model for data) and the Oracle NoSQL Database SDK for Spring Data. The Spring Data Framework provides a familiar and consistent, Spring-based programming model for data access. For more information on Spring Data Framework, see Spring Data.

The Oracle NoSQL Database SDK for Spring Data provides POJO (Plain Old Java Object) centric modeling and integration between the Oracle NoSQL Database and the Spring Data Framework. One of the key benefits available to the Java programmer is the ability to write your code as a repository style data access layer, while the Spring Data Framework maps those repository style data access operations to Oracle NoSQL Database API calls.

The Oracle NoSQL Database SDK for Spring Data is available in Maven Central repository, details available here. The main location of the project is in GitHub.

You can get all the required files for running the Spring Data Framework with the following POM file dependencies.

```
<dependencies>
    <dependency>
        <groupId>com.oracle.nosql.sdk</groupId>
        <artifactId>spring-data-oracle-nosql</artifactId>
    </dependency>
</dependencies>
```

> **✎ Note:**
>
> The Oracle NoSQL Database SDK for Spring Data requires an Oracle NoSQL Database Proxy to connect to an Oracle NoSQL Database cluster. For more information on setting up an Oracle NoSQL Database Proxy, see Oracle NoSQL Database Proxy in the *Administrator's Guide*.

Supported Features

The following features are currently supported by the Oracle NoSQL Database SDK for Spring Data.

- Generic CRUD operations on a repository using methods in the `CrudRepository` interface. For more information on `CrudRepository` interface, see CrudRepository.

- Pagination and sorting operations using methods in the `PagingAndSortingRepository` interface. For more information on `PagingAndSortingRepository` interface, see PagingAndSortingRepository.

- Derived Queries.

- Native Queries.

Example: Accessing Oracle NoSQL Database Using Spring Data Framework

The following example demonstrates how to access Oracle NoSQL Database from Spring using Oracle NoSQL Database SDK for Spring Data. In this example, using the Spring Data Framework, you set up a connection with Oracle NoSQL Database non-secure store, insert a row in the `Student` table, and then retrieve the data from the `Student` table.

In this example, you set up a Maven Project and then add the following classes/interfaces:

- `Student` class

- `StudentRepository` interface

- `AppConfig` class

- `App` class

After that, you will run the Spring application to get the desired output. The following steps discuss this in detail.

1. Set up a Maven project with the following POM file dependencies.

```
<dependencies>
    <dependency>
        <groupId>com.oracle.nosql.sdk</groupId>
        <artifactId>spring-data-oracle-nosql</artifactId>
    </dependency>
</dependencies>
```

2. Create a new package and add the following `Student` entity class to persist. This entity class represents a table in the Oracle NoSQL Database and an instance of this entity corresponds to a row in that table.

```
import com.oracle.nosql.spring.data.core.mapping.NosqlId;
import com.oracle.nosql.spring.data.core.mapping.NosqlTable;

/*The @NosqlTable annotation specifies that
```

```
        this class will be mapped to an Oracle NoSQL Database table.*/
@NosqlTable
public class Student {
    /*The @NosqlId annotation specifies that this field will act
      as the ID field. And the generated=true attribute specifies
      that this ID will be auto-generated by a sequence.*/
    @NosqlId(generated = true)
    long id;
    String firstName;
    String lastName;
    /* public or package protected constructor required when retrieving
from database */
    public Student() {
    }
    /*This method overrides the toString() method, and then
      concatenates id, firstname, and lastname, and then returns a
String*/
    @Override
    public String toString() {
        return "Student{" +
            "id=" + id + ", " +
            "firstName=" + firstName + ", " +
            "lastName=" + lastName +
            '}';
    }
}
```

3. Create the following `StudentRepository` interface. This interface must extend the `NosqlRepository` interface and provide the entity class and the data type of the primary key in that class as sub-typing to the `NosqlRepository` interface. This `NosqlRepository` interface provides methods that could be used to retrieve data from the database.

```
import com.oracle.nosql.spring.data.repository.NosqlRepository;

/*The Student is the entity class, and Long is the data type of the
  primary key in the Student class. This interface implements a derived
query
  findByLastName and returns an iterable instance of the Student class.*/
public interface StudentRepository extends NosqlRepository<Student, Long>
{
    /*The Student table is searched by lastname and
      returns an iterable instance of the Student class.*/
    Iterable<Student> findByLastName(String lastname);
}
```

4. Create the following `AppConfig` class that extends `AbstractNosqlConfiguration` class to provide the connection details of the database.

```
import oracle.nosql.driver.kv.StoreAccessTokenProvider;

import com.oracle.nosql.spring.data.config.AbstractNosqlConfiguration;
import com.oracle.nosql.spring.data.config.NosqlDbConfig;
import
com.oracle.nosql.spring.data.repository.config.EnableNosqlRepositories;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/*The @Configuration annotation specifies that this class can be
  used by the Spring Data Framework as a source of bean
definitions.*/
@Configuration
//annotation to enable NoSQL repositories.
@EnableNosqlRepositories
public class AppConfig extends AbstractNosqlConfiguration {

    public static NosqlDbConfig nosqlDBConfig =
        new NosqlDbConfig("hostname:port", new
StoreAccessTokenProvider());

/*The @Bean annotation tells the Spring Data Framework that the
returned object
  should be registered as a bean in the Spring application.*/
@Bean
    public NosqlDbConfig nosqlDbConfig() {
        return nosqlDBConfig;
    }
}
```

> **Note:**
>
> See Setting up the Connection section to know more about connecting to an Oracle NoSQL Database secure store.

5. This example uses the `CommandLineRunner` interface to create a runner class that implements the `run` method and has the `main` method. You can code the functionality as per your requirements by implementing any of the various interfaces that the Spring Data Framework provides. For more information on setting up a Spring boot application, see Spring Boot.

   In the following code, the first two `Student` entities are created and saved. You then search for all the rows in the Student table and print the results to the output.

```
import com.oracle.nosql.spring.data.core.NosqlTemplate;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

/*The @SpringBootApplication annotation helps you to build
  an application using Spring Data Framework rapidly.*/
@SpringBootApplication
public class App implements CommandLineRunner {

    /*The annotation enables Spring Data Framework to look up the
```

```java
     configuration file for a matching bean.*/
@Autowired
private StudentRepository repo;

public static void main( String[] args ) {
    ConfigurableApplicationContext ctx =
        SpringApplication.run(App.class, args);
    SpringApplication.exit(ctx, () -> 0);
    ctx.close();
    System.exit(0);
}

@Override
public void run(String... args) throws Exception {

    System.out.println("=== Start of App ====");

    //Delete all the existing rows of data, if any, in the Student
table.
    repo.deleteAll();

    //Create a new Student instance and load values into it.
    Student s1 = new Student();
    s1.firstName = "John";
    s1.lastName = "Doe";

    //Save the Student instance.
    repo.save(s1);

    //Create a new Student instance and load values into it.
    Student s2 = new Student();
    s2.firstName = "John";
    s2.lastName = "Smith";

    //Save the Student instance.
    repo.save(s2);


    System.out.println("\nfindAll:");
    /*Selects all the rows in the Student table
      and load it into an iterable instance.*/
    Iterable<Student> students = repo.findAll();

    //Print the values to the output from the iterable object.
    for (Student s : students) {
        System.out.println("  Student: " + s);
    }

    System.out.println("\nfindByLastName: Smith");
    /*The Student table is searched by lastname
      and an iterable instance of the Student class is returned.*/
    students = repo.findByLastName("Smith");

    //Print the values to the output from the iterable instance.
    for (Student s : students) {
```

```
            System.out.println("  Student: " + s);
        }

        System.out.println("=== End of App ====");
    }
}
```

6. Run the program from the runner class. You will get the following output.

```
=== Start of App ====
findAll:
  Student: Student{id=5, firstName=John, lastName=Doe}
  Student: Student{id=6, firstName=John, lastName=Smith}

findByLastName: Smith
  Student: Student{id=6, firstName=John, lastName=Smith}
=== End of App ====
```

Components of Oracle NoSQL Database SDK for Spring Data

The Oracle NoSQL Database Proxy should be set up to facilitate a connection between Oracle NoSQL Database and Spring Data Framework. To set up the Oracle NoSQL Database Proxy, see Oracle NoSQL Database Proxy in the *Administrator's Guide*. Once set up, you then configure the Oracle NoSQL Database Proxy details in the `NosqlRepository` interface. You provide the Oracle NoSQL Database connection and authentication (if any) details in the `NosqlDBConfig` class. The POJOs (entity) with the `@NosqlTable` annotation are mapped to the Oracle NoSQL Database tables by the Oracle NoSQL Database SDK for Spring Data. The following diagram provides the components of the Oracle NoSQL Database SDK for Spring Data.

Figure 1-1   Components of Oracle NoSQL Database SDK for Spring Data
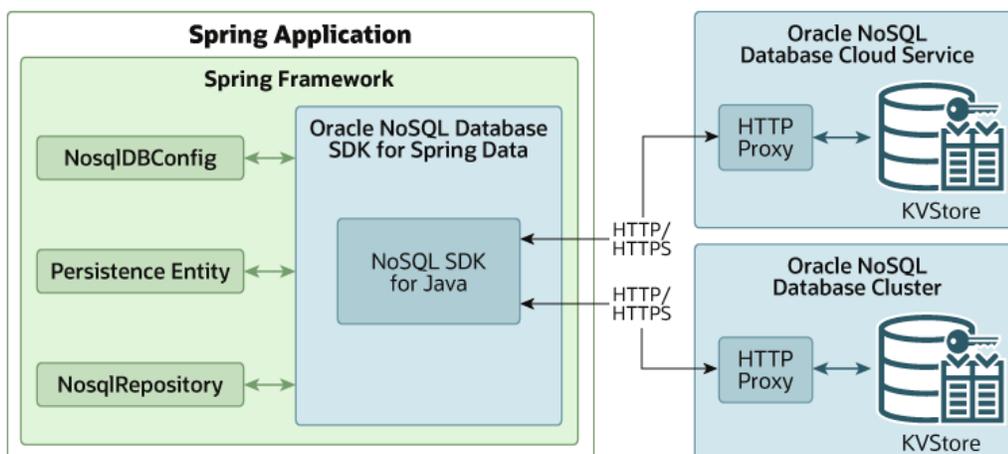
Projections

Use Projections when the required result is a subset of an entity, that is when the required result is just a small part of the entity. You can define an interface or a POJO class with a subset of the properties found in the entity class. Then you use these interfaces or POJO classes as the parametrized type result of the custom repository methods.

Examples

The following examples are shown in the context of Student entity class. See Example: Accessing Oracle NoSQL Database Using Spring Data Framework to get the details on creating the Student entity class and the StudentRepository interface.

1. Define an interface StudentView and a POJO class StudentProjection.

```
public interface StudentView {
    String getLastName();
}


public class StudentProjection {
    private String firstName;
    private String lastName;
    public StudentProjection(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
   public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

2. The new types (StudentView and StudentProjection) can be used as the result of the custom find methods in the StudentRepository class.

```
import java.util.Date;
import com.oracle.nosql.spring.data.repository.NosqlRepository;
public interface StudentRepository
extends NosqlRepository<Student, Long>
{
    Iterable<Student> findByLastName(String lastname);
    Iterable<Student> findByCreatedAtBetween(Date start, Date end);
    Iterable<StudentView> findAllByLastName(String lastName);
```

```
    Iterable<StudentProjection> getAllByLastName(String lastName);
}
```

Since these results contain a subset of the row, if the Id property is not included the returned set could contain duplicates. If these duplicates are not required then you can use the `Distinct` keyword to eliminate them.

Example:

```
List<StudentView>findAllDistinctByLastName(String lastName);
List<StudentProjection> getAllDistinctByLastName(String lastName);
```

These methods will generate the following queries:

```
declare $p_lastName String;
select distinct {'lastName': t.kv_json_.lastName} as kv_json_ from
Student \
                              as t where t.kv_json_.lastName
= $p_lastName


declare $p_lastName String;
select distinct {'firstName': t.kv_json_.firstName, 'lastName':
t.kv_json_.lastName} as kv_json_ \
from Student as t where t.kv_json_.lastName = $p_lastName
```

> **Note:**
>
> Only interface and class based projections that contain a subset of entity properties are supported by NoSQL SDK for Spring Data. Projections using @Value annotations are not supported. Dynamic projections, when return type is parametrized, are also not supported.

3. Modify the `run` method and invoke the custom methods (defined with Projection interface and POJO Class).

```
// Using projection interface
System.out.println("\n With projection findAllByLastName: Smith");
repo.findAllByLastName("Smith")
.forEach(c -> System.out.println("StudentView :" + c));
// using projection POJO class here
System.out.println("\n With projection getAllByLastName: Smith");
repo.getAllByLastName("Smith")
.forEach(c -> System.out.println("StudentProjection.firstName :" +
c.getFirstName()
                              + " StudentProjection.lastName :" +
c.getLastName() ));
```

> **📝 Note:**
>
> See Example: Accessing Oracle NoSQL Database Using Spring Data Framework to get more details on the `AppConfig` class to provide the connection details of the database and the `App` class that implements the `run` method and has the `main` method.

4. Run the program from the runner class. You will get the following output.

```
With projection findAllByLastName: Smith
StudentView :Student{id=0, firstName='null', lastName='Smith',
createdAt='null'}
With projection getAllByLastName: Smith
StudentProjection.firstName :John
StudentProjection.lastName :Smith
```

Persistence Model

An entity is a lightweight persistence domain object. The persistent state of an entity is represented through persistent fields using Java Beans / Plain Old Java Objects (POJOs).

The Spring Data Framework supports the persistence of entities to Oracle NoSQL Database tables. An entity is mapped to a table. The `ID` field in that entity is mapped to the primary key column of that table. All other fields in the entity are mapped to a JSON column of that table. Each instance of the entity will be stored as a single row in that table. The value of the `ID` field in that instance will be stored as the primary key value of that row. The values of all other fields (including other objects) (see JSON Column) in that instance will be serialized and stored as values in the JSON column of that row. Effectively, the table will always have only two columns: a primary key column and a JSON column.

Figure 1-2   Persistence Model
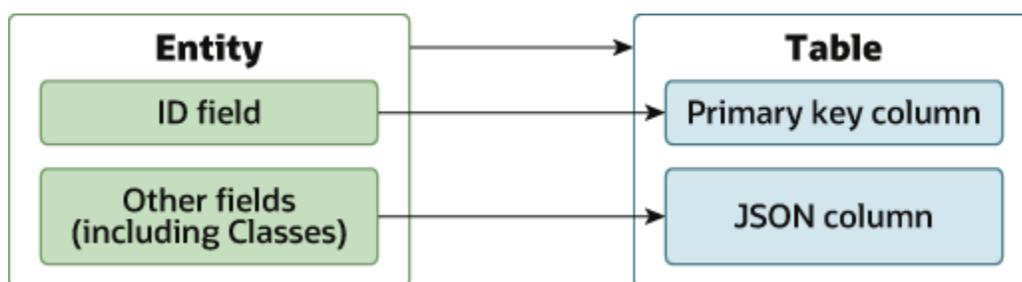
If a persistent POJO has a reference to another persistent POJO (nested objects) that maps to a different table, the Spring Data Framework will not serialize objects to multiple tables. Instead, all the nested objects will be serialized and stored as values in the JSON column. For more information on JSON Column mappings, see JSON Column.

The following is the syntax of an entity with `@NosqlTable` and `@NosqlId` annotations. In the example below, the `Student` class with the `@NosqlTable` annotation will be mapped to a table named `Student` in the Oracle NoSQL Database. The `ID` field with the `@NosqlId` annotation

will be the primary key field in the `Student` table. The `firstName` and `lastName` fields will be mapped to a single JSON field named `kv_json_` in the `Student` table.

When retrieving entries from the repository the driver needs to instantiate the entity classes. These classes need to have a default constructor or an empty constructor that is public or package protected.

> **✎ Note:**
>
> The classes may have other constructors too.

```
/*The @NosqlTable annotation specifies that
this class will be mapped to an Oracle NoSQL Database table.*/
@NosqlTable
public class Student {
    //The @NosqlId annotation specifies that this field will act as
the ID field.
    @NosqlId
    public long ID;

    public String firstName;
    public String lastName;

    public Student() {}
}
```

Table Name

By default, the entity simple class name is used for the table name. You can provide a different table name using the `@NosqlTable` annotation. The `@NosqlTable` annotation enables you to define additional configuration parameters such as table name and timeout.

For example, an entity named `Student` will be persisted in a table named `Student`. If you want to persist an entity named `Student` in a table named `Learner`, you can achieve that using the `@NosqlTable` annotation.

If the `@NosqlTable` annotation is specified, then the following configuration could be provided.

Table 1-1   Attributes in NosqlTable Annotation

Paramete r	Type	Ignored/ Optional/ Required in Oracle NoSQL Database	Ingnored/ Optional/ Required in Oracle NoSQL Database Cloud Service	Default	Description
tableNam e	String	Optional	Optional	empty	Specifies the name of the table, simple or namespace-qualified form. If empty, then the entity class name will be used. For more information on the namespace, see Namespace Management in the *SQL Reference Guide*. In the Oracle NoSQL Database Cloud Service, the namespace part, if provided, is used as the compartment name. For more information on using compartments, see About Compartments in the *Using Oracle NoSQL Database Cloud Service*.

Table 1-1   (Cont.) Attributes in NosqlTable Annotation

Paramete r	Type	Ignored/ Optional/ Required in Oracle NoSQL Database	Ingnored/ Optional/ Required in Oracle NoSQL Database Cloud Service	Default	Description
autoCrea te	boolean	Optional	Optional	true	Specifies if the table should be created if it does not exist.

> **Note:**
>
> The Spring Data Framewor k looks for the repositorie s used in the applicatio n in the `init` phase. If the table does not exist, and if the `@NosqlTa ble` annotation has the `autoCrea te` as `true,` then the table will be created in the `init` phase.

Table 1-1   (Cont.) Attributes in NosqlTable Annotation

Parameter	Type	Ignored/ Optional/ Required in Oracle NoSQL Database	Ingnored/ Optional/ Required in Oracle NoSQL Database Cloud Service	Default	Description
readUnits	int	Ignored	Required	-1	Specifies the maximum read throughput to be used if the table is to be created.
					For more information on readUnits, see Service Metrics in the *Using Oracle NoSQL Database Cloud Service.*
					✎ **Note:** In Oracle NoSQL Database Cloud Service, the readUnits parameter should be set to a value greater than 0 else it will return an error.

Table 1-1   (Cont.) Attributes in NosqlTable Annotation

Parameter	Type	Ignored/ Optional/ Required in Oracle NoSQL Database	Ingnored/ Optional/ Required in Oracle NoSQL Database Cloud Service	Default	Description
writeUnits	int	Ignored	Required	-1	Specifies the maximum write throughput to be used if the table is to be created. For more information on writeUnits, see Service Metrics in the *Using Oracle NoSQL Database Cloud Service*. **✎ Note:** In Oracle NoSQL Database Cloud Service, the writeUnits parameter should be set to a value greater than 0 else it will return an error.

Table 1-1   (Cont.) Attributes in NosqlTable Annotation

Parameter	Type	Ignored/ Optional/ Required in Oracle NoSQL Database	Ingnored/ Optional/ Required in Oracle NoSQL Database Cloud Service	Default	Description
storageGB	int	Ingored	Required	-1	Specifies the maximum amount of storage, in gigabytes, allowed for the table, if the table is to be created. For more information on storageGB, see Service Metrics in the *Using Oracle NoSQL Database Cloud Service*. ✎ **Note:** In Oracle NoSQL Database Cloud Service, the storageGB parameter should be set to a value greater than 0 else it will return an error.

Table 1-1   (Cont.) Attributes in NosqlTable Annotation

Parameter	Type	Ignored/ Optional/ Required in Oracle NoSQL Database	Ingnored/ Optional/ Required in Oracle NoSQL Database Cloud Service	Default	Description
timeout	int	Optional	Optional	0	Specifies the maximum time length, in milliseconds, that the operations are allowed to take before a timeout exception is thrown.
					If the value for `timeout` is not set then the timeout set in `NoSQLHandleConfig` class is used. For information on getting the timeout from `NoSQLHandleConfig` class using `getTableRequestTimeout()` method, see NoSQLHandleConfig in the *Java SDK API Reference*.
					The `timeout` value can also be changed using `NosqlRepository.setTimeout(int)` method. For more information, see setTimeout in the *SDK for Spring Data API Reference*.

Table 1-1   (Cont.) Attributes in NosqlTable Annotation

Parameter	Type	Ignored/ Optional/ Required in Oracle NoSQL Database	Ingnored/ Optional/ Required in Oracle NoSQL Database Cloud Service	Default	Description
`consistency`	`String`	Optional	Optional	`EVENTUAL`	Specifies the consistency used for read operations. Valid values are based on `oracle.nosql.driver.Consistency` are `EVENTUAL` and `ABSOLUTE` . See Consistency in the *Java SDK API Reference*.

> ✎ **Note:**
>
> This is the default for all read operations . It can be overridden by using `NosqlRepository. setConsistency(String)`. For more information, see setConsistency in the *SDK for Spring Data API Reference*.

Primary Key

The table requires a primary key. The field named `ID` in the entity will be used as the primary key. You can select a different field in the entity (a field with a different name other than `ID`) to designate as the primary key using the `@NosqlId` annotation or the `@id` annotation.

When an `ID` field is mapped to a primary key column, the Spring Data Framework will automatically assign the corresponding data type to that field before storing them in the table. The following is a list of data type mappings between a Java type and an Oracle NoSQL Database type for the `ID` field.

The Java types that are provided in the following table are the only valid data types that can be used for a primary key.

Table 1-2   Mapping Between Java and Oracle NoSQL Database Types

Java Type	Oracle NoSQL Database Type
`java.lang.String`	STRING
int `java.lang.Integer`	INTEGER
long `java.lang.Long`	LONG

double
`java.lang.Double`
float
`java.lang.Float`
	DOUBLE

> **Note:**
>
> double, `java.lang.Double`, `float`, and `java.lang.Float` can be a primary key but it's not a valid `generated=true` type

> **Note:**
>
> Since `FLOAT` in Oracle NoSQL Database type is not explicitly used in NoSQL SDK for Java, the Java `float` and `java.lang.Float` are mapped to the `DOUBLE` type.

Java Type	Oracle NoSQL Database Type
java.math.BigDecimal `java.math.BigInteger`	NUMBER
boolean `java.lang.Boolean`	BOOLEAN

java.util.Date
`java.sql.Timestamp`
`java.time.Instant`
	TIMESTAMP (P)

The Spring Data Framework deduces the primary key using the following rules:

- **@NosqlId annotation:** If `@NosqlId` annotation is used on a field with a valid data type for the primary key, then that field is considered as the primary key. If `@NosqlId` is used on a field of a type other than a valid data type for the primary key, an error is raised. For more information, see NosqlId in the *SDK for Spring Data API Reference*.

- **@org.springframework.data.annotation.Id annotation:** If `@org.springframework.data.annotation.Id` field annotation is used on a field with a valid data type for the primary key, then that field is considered as the

primary key. If `@org.springframework.data.annotation.Id` is used on a field of a type other than a valid data type for the primary key, an error is raised.

- **Not specified:** If none of the above two annotations are specified, then the Spring Data Framework will use the field named `ID` as the primary key.

An error is raised if:

- No `@NosqlId` annotation or `@org.springframework.data.annotation.Id` annotation or `ID` field is found in the entity, as no primary key field can be inferred.

- Two or more of `@NosqlId` or `@org.springframework.data.annotation.Id` annotated fields are used in the entity, as multiple primary key fields can be inferred.

> **Note:**
>
> The name of the fields that take the `@NosqlId` or `@org.springframework.data.annotation.Id` annotations must not be named `kv_json_`. It is because the second column of the table created by the Spring Data Framework will be named as `kv_json_` and will be a JSON column where all attributes in the persistent entity that are not listed as primary key attributes will be stored.

The `@NosqlId` field annotation can take the following additional configuration:

Table 1-3   Attributes in NosqlId Annotation

Parameter	Type	Optional/ Required	Default	Description

`generated`	`boolean`	Optional	`false`	Specifies if the `ID` is auto-generated or not.
				
				• If `true`, then it is defined as auto-generated by the program.
				  – If `int/Integer`, `long/Long`, `BigInteger` or `BigDecimal`, then `GENERATED ALWAYS as IDENTITY` is used.
				  – If `String`, then "String as UUID GENERATED BY DEFAULT" is used.
				
				> **Note:**
				> Currently not available in the Oracle NoSQL Database Cloud Service.
				
				• If `false`, then the value must be managed by your application.

> **Note:**
>
> Composite primary keys are not supported.

JSON Column

All other fields in the entity other than the primary key field will be converted into a NoSQL JSON value with the following rules:

- The Java scalar values will be converted to NoSQL JSON atomic values.

- The Java collections and array structures will be converted to a NoSQL JSON array.

- The Java non-scalar values will be recursively converted to NoSQL JSON objects.

- The Java null values will be converted to NoSQL JSON NULL values.

- The complex values will be converted to NoSQL JSON objects according to the following table.

Table 1-4   Mapping Between Java and NoSQL JSON Types

Java Type	Representation within Oracle NoSQL Database JSON Datatype
`java.lang.String`	STRING
int `java.lang.Integer`	INTEGER
long `java.lang.Long`	LONG

double
`java.lang.Double`
`float`
`java.lang.Float`
	DOUBLE

> **Note:**
>
> Since `FLOAT` in Oracle NoSQL Database type is not explicitly used in NoSQL SDK for Java, Java `float`, and `java.lang.Float` are mapped to the `DOUBLE` type.

Java Type	Representation within Oracle NoSQL Database JSON Datatype
java.math.BigDecimal `java.math.BigInteger`	NUMBER
boolean `java.lang.Boolean`	BOOLEAN
`byte[]`	STRING - a binary base64-encoded representation.

java.util.Date
`java.sql.Timestamp`
`java.time.Instant`
	STRING - an ISO-8601 UTC timestamp encoded representation.

`org.springframework.data.geo.Point`	GeoJson Point For more information on GeoJson Data, see About GeoJson Data in the *SQL Reference Guide*.

Table 1-4   (Cont.) Mapping Between Java and NoSQL JSON Types

Java Type	Representation within Oracle NoSQL Database JSON Datatype
`org.springframework.d ata.geo.Polygon`	GeoJson Polygon For more information on GeoJson Data, see About GeoJson Data in the *SQL Reference Guide* . **✎ Note:** Polygons must conform to the following rules to be well-formed, otherwise they will be ignored when used in queries. 1. A linear ring is a closed `LineString` with four or more positions. 2. The first and last positions are equivalent, and they must contain identical values. 3. A linear ring is either the boundary of a surface or the boundary of a hole in a surface. 4. A linear ring must follow the right-hand rule for the area it bounds, that is, for exterior rings, their positions must be ordered counterclockwise, and for holes, their position must be ordered clockwise. Before inserting new polygons in the table, the `geo_is_geometry()` function can be used for verification. If polygon data is indexed an error will be raised if for some row the value of the index path is not valid, unless that value is NULL, json null, or EMPTY.

Table 1-4   (Cont.) Mapping Between Java and NoSQL JSON Types

Java Type	Representation within Oracle NoSQL Database JSON Datatype

`java.util.ArrayList`
`java.util.Collection`
`java.util.List`
`java.util.AbstractList`
`java.util.HashSet`
`java.util.Set`
`java.util.AbstractSet`
`java.util.TreeSet`
`java.util.SortedSet`
`java.util.NavigableSet`
`java.util.Array []`
	`ARRAY(JSON)`
	
	Note:
	
	• For fields of type `java.util.Collection`, `java.util.List`, `java.util.AbstractList`, and `java.util.ArrayList`, an `java.util.ArrayList` object is instantiated.
	• For fields of type `java.util.Set`, `java.util.AbstractSet`, and `java.util.HashSet`, a `java.util.HashSet` object is instantiated.
	• And for fields of type `java.util.SortedSet`, `java.util.NavigableSet`, and `java.util.TreeSet`, a `java.util.TreeSet` object is instantiated.

`POJO<f1 T1, f2 T2...>`	`MAP(JSON)`
`java enum types`	`STRING`

> **Note:**
>
> Java data structures that contain cycles are neither supported nor detected. That is, if the entity object is traversed from the root down the fields and encounters the same object twice it becomes a cycle.

Transactional Model

The transaction model for the Oracle NoSQL Database SDK for Spring Data builds on top of the existing transaction model exposed by the Oracle NoSQL Database. That is, ACID transactions are only supported for operations that do not span database shards. From the perspective of your Spring application, you should think about ACID transactions as being supported for those repository methods that operate over single objects. Repository methods like `deleteAll()` are implemented in the Oracle NoSQL Database SDK for Spring Data to make a "best-effort" to complete across all database shards but make no ACID guarantees.

The write operations when using `save()`, `saveAll()`, `delete()`, `deleteById()`, `deleteAll()` or write queries will be done based on the default Java driver durability. For more information on default Java driver durability, see COMMIT_NO_SYNC in the *Java Direct Driver API Reference*.

The read operations when using `findByID()`, `findAllById()`, `findAll()`, `count()` or select queries will be done based on the default eventual consistency or as specified in the `@NosqlTable` annotation. For more information on default eventual consistency, see getDefaultConsistency in the *Java SDK API Reference*.

Setting up the Connection

To expose the connection and security parameters to the Oracle NoSQL Database SDK for Spring Data, you need to create a class that extends the `AbstractNosqlConfiguration` class. You could customize this code as per your requirement. Perform the following steps to set up a connection to the Oracle NoSQL Database.

Step 1: In your application, create the `NosqlDbConfig` class. This class will have the connection details to the Oracle NoSQL Database Proxy. Provide the `@Configuration` and `@EnableNoSQLRepositories` annotations to this `NosqlDbConfig` class. The `@Configuration` annotation tells the Spring Data Framework that the `@Configuration` annotated class is a configuration class that should be loaded before running the program. The `@EnableNoSQLRepositories` annotation tells the Spring Data Framework that it needs to load the program and lookup for the repositories that extends the `NosqlRepository` interface. The `@Bean` annotation is required for the repositories to be instantiated.

Step 2: Create an `@Bean` annotated method to return an instance of the `NosqlDBConfig` class. The `NosqlDBConfig` class will also be used by the Spring Data Framework to authenticate the Oracle NoSQL Database.

Step 3: Instantiate the `NosqlDbConfig` class. Instantiating the `NosqlDbConfig` class will cause the Spring Data Framework to internally instantiate an Oracle NoSQL Database handle by authenticating with the Oracle NoSQL Database.

> **Note:**
>
> You could add an exception code block to catch any connection error that might be thrown upon authentication failure.

> **Note:**
>
> Creating an Oracle NoSQL Database handle using the above-mentioned steps has a limitation. The limitation is that the application will not be able to connect to two or more different clusters at the same time. This is a Spring Data Framework limitation. For more information on Spring Data Framework, see Spring Core.

> **✎ Note:**
>
> If you have trouble connecting to Oracle NoSQL Database from your Spring application, you can add an exception block and print the message for debugging.

As given in the following example, you can use the StoreAccessTokenProvider class to configure the Spring Data Framework to connect and authenticate with an Oracle NoSQL Database. You need to provide the URL of the Oracle NoSQL Database Proxy with non-secure access.

```
/*Annotation to specify that this class can be used by the
  Spring Data Framework as a source of bean definitions.*/
@Configuration
//Annotation to enable NoSQL repositories.
@EnableNosqlRepositories
public class AppConfig extends AbstractNosqlConfiguration {

    /*Annotation to tell the Spring Data Framework that the returned
object
      should be registered as a bean in the Spring application.*/
    @Bean
    public NosqlDbConfig nosqlDbConfig() {
        AuthorizationProvider authorizationProvider;
        authorizationProvider = new StoreAccessTokenProvider();
        //Provide the host name and port number of the NoSQL cluster.
        return new NosqlDbConfig("http://<host:port>",
authorizationProvider);
    }
}
```

The following example modifies the previous example to connect to a secure Oracle NoSQL Database store. For more details on StoreAccessTokenProvider class, see StoreAccessTokenProvider in the *Java SDK API Reference*.

```
/*Annotation to specify that this class can be used by the
  Spring Data Framework as a source of bean definitions.*/
@Configuration
//Annotation to enable NoSQL repositories.
@EnableNosqlRepositories
public class AppConfig extends AbstractNosqlConfiguration {

    /*Annotation to tell the Spring Data Framework that the returned
object
      should be registered as a bean in the Spring application.*/
    @Bean
    public NosqlDbConfig nosqlDbConfig() {
        AuthorizationProvider authorizationProvider;
        //Provide the username and password of the NoSQL cluster.
        authorizationProvider = new StoreAccessTokenProvider(user,
password);
        //Provide the host name and port number of the NoSQL cluster.
```

```
        return new NosqlDbConfig("http://<host:port>",
authorizationProvider);
    }
}
```

For secure access, the `StoreAccessTokenProvider` parameterized constructor takes the following arguments.

- `username` is the username of the kvstore.

- `password` is the password of the kvstore user.

For more details on the security configuration, see Creating NoSQL Handle in the *Administrator's Guide*.

As given in the following example, you can use the `SignatureProvider` class to configure the Spring Data Framework to connect and authenticate with the Oracle NoSQL Database Cloud Service. See SignatureProvider in the *Java SDK API Reference*.

```
/*Annotation to specify that this class can be used by the
  Spring Data Framework as a source of bean definitions.*/
@Configuration
//Annotation to enable NoSQL repositories.
@EnableNosqlRepositories
public class AppConfig extends AbstractNosqlConfiguration {

    /*Annotation to tell the Spring Data Framework that the returned object
      should be registered as a bean in the Spring application.*/
    @Bean
    public NosqlDbConfig nosqlDbConfig() {
        SignatureProvider signatureProvider;

        /*Details that are required to authenticate and authorize access to
          the Oracle NoSQL Database Cloud Service are provided.*/
        signatureProvider = new SignatureProvider(
            <tenantId>,  //The Oracle Cloud Identifier (OCID) of the tenancy.
            <userId>,  //The Oracle Cloud Identifier (OCID) of a user in the
tenancy.
            <fingerprint>,  //The fingerprint of the key pair used for
signing.
            <privateKeyFile>,  //Full path to the key file.
            <passphrase>  //Optional. A pass phrase for the key, if it is
encrypted.
        );
        /*Provide the service URL of the Oracle NoSQL Database Cloud Service
and
          update the 'Region.US_PHOENIX_1' with an appropriate value.*/
        return new NosqlDbConfig(Region.US_PHOENIX_1,signatureProvider);
    }
}
```

Defining a Repository

The entity class that is used for persistence is discoverable by the Spring Data Framework either via annotation or inheritance. The `NosqlRepository` interface allows you to inherit and create an interface for each entity that will use the Oracle NoSQL Database for persistence.

The `NosqlRepository` interface extends Spring's `PagingAndSortingRepository` interface that provides many methods that define queries.

In addition to those methods that are provided by the `NosqlRepository` interface, you can add methods to your repository interface to define derived queries. These interface methods follow a specific naming pattern for Spring derived queries (for more information derived queries, see Query Creation) intercepted by the Spring Data Framework. The Spring Data Framework will use this naming pattern to generate an expression tree, passing this tree to the Oracle NoSQL Database SDK for Spring Data, where this expression tree is converted into an Oracle NoSQL Database query, which is compiled and then executed. These Oracle NoSQL Database queries are executed when you call the repository's respective methods.

If you wish to create your derived queries, this must be done by extending the `NosqlRepository` interface and adding your own Java method signatures that conform to the naming patterns as discussed in the derived queries section.

The following is an example of a code that implements the `NosqlRepository` interface. You must provide the bounded type parameters: the entity type and the data type of the `ID` field. This interface implements a derived query `findByLastName` and returns an iterable instance of the `Student` class.

```
import com.oracle.nosql.spring.data.repository.NosqlRepository;

/*The Student is the entity class, and Long is the data type of the
  primary key in the Student class. This interface implements a
derived query
  findByLastName and returns an iterable instance of the Student
class.*/
public interface StudentRepository extends NosqlRepository<Student,
Long> {

    /*The Student is searched by lastname and
      an iterable instance of the Student class is returned.*/
    Iterable<Student> findByLastName(String lastname);
}
```

Starting the Application

After creating the entity and repository, you should write a program to run the Spring application. You can do that using a Spring boot application or a Spring core application.

Create an `@SpringBootApplication` annotated class to run a Spring boot application. You could override the run method in the `CommandLineRunner` interface to write your code.

The following is an example of a Spring boot application.

```
//The annotation helps to build an application using Spring Data Framework
rapidly.
@SpringBootApplication
public class BootExample implements CommandLineRunner {

    /*The annotation enables Spring Data Framework to
      look up the configuration file for a matching bean.*/
    @Autowired
    private StudentRepository nosqlRepo;

    @Override
    public void run(String... args) throws Exception {
        ...
    }
}
```

The following is an example of a Spring core application.

```
public class CoreExample {
    public static void main(String[] args) {
        ApplicationContext ctx =
            new AnnotationConfigApplicationContext(AppConfig.class);
        NosqlOperations ops = (NosqlOperations)ctx.getBean("nosqlTemplate");
        ...
    }
}
```

> **✎ Note:**
>
> The Spring Data Framework will look in the classpath for a class with the
> `@configuration` annotation and contains a method named "NosqlTemplate" with
> the `@Bean` annotation.

Queries

You can use the queries provided in the repository base classes such as the
`PagingAndSortingRepository` interface, or write your queries. The Spring Data Framework
supports the following types of queries.

1. Generic queries - queries provided by methods in the `PagingAndSortingRepository`
   interface and `CrudRepository` interfaces.

2. Derived queries - queries derived/generated by Spring SDK from the name of the method
   based on the keywords.

3. Native queries - queries provided by user in the SQL for NoSQL Database format.

`PagingAndSortingRepository` Interface

The `NosqlRepository` interface extends the `PagingAndSortingRepository` interface.

The `PagingAndSortingRepository` interface extends the `CrudRepository` interface and provides methods such as:

- Page<T> findAll(Pageable pageable)
- Iterable<T> findAll(Sort sort)
- long count()
- void delete(T entity)
- void deleteAll()
- void deleteAll(Iterable<? extends T> entities)
- void deleteAllById(Iterable<? extends ID> ids)
- void deleteById(ID id)
- boolean existsById(ID id)
- Iterable<T> findAll()
- Iterable<T> findAllById(Iterable<ID> ids)
- Optional<T> findById(ID id)
- <S extends T> S save(S entity)
- <S extends T> Iterable<S> saveAll(Iterable<S> entities)

You can use any of these methods for the required functionality.

For more information on the Spring's `PagingAndSortingRepository` interface, see PagingAndSortingRepository.

Derived Queries

Apart from those query methods that are provided by Spring's `PagingAndSortingRepository` interface, you could also define derived queries. Spring Data Framework has an inbuilt query creation feature. Spring Data Framework creates queries directly from the Java method name alone.

For example, if we have a Java method name with the following construct,

```
List<Customer> findByFirstName(String firstName);
```

then the following derived query will be auto-created by the Spring Data Framework.

```
declare $firstName String;

SELECT * FROM Customer AS c WHERE c.kv_json_.firstName = $firstName;
```

The only requirement for this derived query to work is that this Java method should be defined in the interface that extends the `NosqlRepository` interface. The `NosqlRepository` interface extends the `Repository` interface which is responsible for

the derived queries. The common prefixes from the Java method name are removed and the constraints of the query are parsed from the rest of the Java method name. For more information on Spring derived query creation, see Query Creation.

The Java methods with the prefixes `find…By`, `read…By`, `query…By`, `count…By`, `get…By`, `exists… By`, `delete…By`, and `remove…By` are considered as derived query methods by Spring Data Framework. Apart from these prefixes, the Java method name could also have other keywords. The following section provides the detailed derived query snippets that would be generated if the given keywords are used.

And

If a method name has the word `and` in the following construct,

```
Iterable<Student> findByFirstNameAndLastName(String firstname, String
lastname);
```

then the following derived query will be auto-created by the Spring Data Framework.

```
declare $p_firstName String;
$p_lastName String;

SELECT * FROM Student AS s WHERE (
    s.kv_json_.firstName = $p_firstName AND s.kv_json_.lastName
= $p_lastName)
```

> **Note:**
>
> The Oracle NoSQL Database SDK for Spring Data supports derived queries that use a combination of the logical operators (`and`, `or`). The generated query will follow the rules of operator precedence defined in the Oracle NoSQL Database SQL query language. For more information on the operator precedence in the Oracle NoSQL Database SQL query language, see Operator Precedence in the *SQL Reference Guide*.

Or

If a method name has the word or in the following construct,

```
Iterable<Student> findByFirstNameOrLastName(String firstname, String
lastname);
```

then the following derived query will be auto-created by the Spring Data Framework.

```
declare $p_firstName String;
$p_lastName String;

SELECT * FROM Student AS s WHERE (
    s.kv_json_.firstName = $p_firstName OR s.kv_json_.lastName = $p_lastName)
```

> **Note:**
>
> The Oracle NoSQL Database SDK for Spring Data supports derived queries
> that use a combination of the logical operators (`and`, `or`). The generated
> query will follow the rules of operator precedence defined in the Oracle
> NoSQL Database SQL query language. For more information on the
> operator precedence in the Oracle NoSQL Database SQL query language,
> see Operator Precedence in the *SQL Reference Guide*.

`OrderBy (Asc/Desc)`

If a method name has the word orderby in the following construct,

```
Iterable<Student> findByLastNameOrderByFirstNameAsc(String lastname);
```

then the following derived query will be auto-created by the Spring Data Framework.

```
declare $p_lastName String;

SELECT * FROM Student AS s
    WHERE s.kv_json_.lastName = $p_lastName ORDER BY
s.kv_json_.firstName ASC
```

If a method name has the word orderby in the following construct,

```
Iterable<Student> findByLastNameOrderByFirstNameDesc(String lastname);
```

then the following derived query will be auto-created by the Spring Data Framework.

```
declare $p_lastName String;

SELECT * FROM Student AS s
    WHERE s.kv_json_.lastName = $p_lastName ORDER BY
s.kv_json_.firstName DESC
```

`First`

If a method name has the word first in the following construct,

```
Page<Student> queryFirst5ByLastname(String lastname, Pageable
pageable);
```

then the following derived query will be auto-created by the Spring Data Framework.

For more information on `Page`, see Page. For more information on `Pageable`, see
Pageable.

```
declare $p_lastName String;
$kv_limit_ Long;
$kv_offset_ Long;
```

```
SELECT * FROM Student AS s
    WHERE s.kv_json_.lastName = $p_lastName LIMIT $kv_limit_
OFFSET $kv_offset_
```

Top

If a method name has the word top in the following construct,

```
Slice<Student> findTop10ByLastName(String lastname, Pageable pageable);
```

then the following derived query will be auto-created by the Spring Data Framework.

For more information on `Slice`, see Slice.

```
declare $p_lastName String;
$kv_limit_ Long;
$kv_offset_ Long;

SELECT * FROM Student AS s
    WHERE s.kv_json_.lastName = $p_lastName LIMIT $kv_limit_
OFFSET $kv_offset_
```

For the complete list of supported keywords in query methods in Oracle NoSQL Database SDK for Spring Data, see Supported Keywords in Query Method.

The following is an example of an Oracle NoSQL Database repository. It must extend the `NosqlRepository` interface. The bounded types represent the entity type and the data type of the `ID` field.

```
interface PersonRepository extends NosqlRepository<Person, Long> {
    List<Person> findByFirstNameAndLastName(String firstname, String
lastname);
    List<Person> findByLastNameOrderByFirstNameDesc(String lastname);
}
```

Native Queries

The `@oracle.spring.data.nosql.repository.Query` annotation allows you to execute the native SQL query.

```
public interface AuthorRepository extends NoSQLRepository<Author, Long> {
    @Query(value = "DECLARE $firstName STRING;
        SELECT * FROM author WHERE first_name = $firstName")
    List<Author> findAuthorsByFirstName(@Param("firstName") String
firstName);

    @Query("DECLARE $firstName STRING; $last STRING; " +
        "SELECT * FROM Customer AS c " +
        "WHERE c.kv_json_.firstName = $firstName AND " +
        "c.kv_json_.lastName = $last")
    List<Customer> findCustomersWithLastAndFirstNosqlValues(
        @Param("$last") StringValue paramLast,
        @Param("$firstName") StringValue firstName
```

```
        );
}
```

Parameters are matched by name using the
`@org.springframework.data.repository.query.Param` annotation. The `@Param`
annotation value field must match exactly, including the '$' char, the name of the
declared bind variable. If `@Param` annotation is not used an exception is thrown. All the
parameters will get mapped according to the mapping rules mentioned in the
Persistence Model section.

> **✎ Note:**
>
> The second method `findAuthorsWithLastAndFirstNosqlValues` works with
> `oracle.nosql.driver.values.StringValue`. All `FieldValue` sub-classes are
> supported for query parameters. `FieldValue` is the base class of all data
> items in the NoSQL SDK for Java. Each data item is an instance of
> `FieldValue` allowing access to its type and its value as well as additional
> utility methods that operate on `FieldValue`. On top of that, parameters of
> type `FieldValue` are also supported. For more information on `FieldValue`,
> see FieldValue.

For details on full query support in the Oracle NoSQL Database, see SQL Reference
Guide.

Supported Keywords in Query Method

The following is the list of supported keywords for prefix in the derived query method
name.

Table 1-5   Supported Keywords for Prefix

Prefix Keyword	Example
findBy	`List<Customer> findByFirstName(String firstName)`
queryBy	`List<Customer> queryByFirstName(String firstName)`
getBy	`List<Customer> getByFirstName(String firstName)`
readBy	`List<Customer> readByFirstName(String firstName)`
countBy	`long countByFirstName(String firstName)` - returns the count of the matching rows
existsBy	`boolean existsByLastName(String lastname)` - returns true if returned rows > 0

The following is the list of supported keywords for body in the derived query method
name.

Table 1-6   Supported Keywords for Body

Body Keyword	No. of Parts	No. of Params	Example
`fieldname`	1	1	`List<Customer> findByLastName(String lastName)`

`fieldnameReferencefieldname`	1	1	`List<Customer> findByAddressCity(String city)`
			`class Customer { Address adress; ...}`
			`class Address { String city; ...}`

`And`	2	0	`List<Customer> findByFirstNameAndLastName(String firstName, String lastName)`
`Or`	2	0	`List<Customer> findByFirstNameOrLastName(String firstName, String lastName`
`GreaterThan`	1	1	`List<Customer> findByAgeGreaterThan(int minAge)`
`GreaterThanEqual`	1	1	`List<Customer> findByAgeGreaterThanEqual(int minAge)`
`LessThan`	1	1	`List<Customer> findByAgeLessThan(int maxAge)`
`LessThanEqual`	1	1	`List<Customer> findByAgeLessThanEqual(int maxAge)`
`IsTrue`	1	0	`List<Customer> findByVanillaIsTrue()`
`Desc`	1	0	`List<Customer> queryByLastNameOrderByFirstNameDesc(String lastname)`
`Asc`	1	0	`List<Customer> getByLastNameOrderByFirstNameAsc(String lastname)`
`In`	1	1	`List<Customer> findByAddressCityIn(List<Object> cities)` - param must be a List
`NotIn`	1	1	`List<Customer> findByAddressCityNotIn(List<String> cities)` - param must be a List
`Between`	2	2	`List<Customer> findByKidsBetween(int min, int max)`
`Regex`	1	1	`List<Customer> findByFirstNameRegex(String regex)`
`Exists`	1	0	`List<Customer> findByAddressCityExists()` - find all that have a city set
`Near`	1	1	`List<Customer> findByAddressGeoJsonPointNear(Circle circle)` - param must be of org.springframework.data.geo.Circle type

Table 1-6   (Cont.) Supported Keywords for Body

Body Keyword	No. of Parts	No. of Params	Example
Within	1	1	List<Customer> findByAddressGeoJsonPointWithin(Polygon point) - param must be of org.springframework.data.geo.Polygon type
IgnoreCase	1	0	List<Customer> findByLastNameAndFirstNameIgnoreCase(String lastname, String firstname); -Enable ignore case only for firstName field
AllIgnoreCase	many	0	List<Customer> findByLastNameAndFirstNameAllIgnoreCase(String lastname, String firstname); - Enable ignore case for all suitable properties
Distinct	0	0	List<CustomerView> findAllDistinctByLastName(String lastName); - Projection to interface CustomerView List<CustomerProjection> getAllDistinctByLastName(String lastName); - Projection to POJO class CustomerProjection

Activating Logging

To enable logging in Oracle NoSQL Database SDK for Spring Data, you must include the following parameter when running the application.

```
-Dlogging.level.com.oracle.nosql.spring.data=DEBUG
```

The following are the logging levels that you could provide:

- **ERROR**: The ERROR level logging includes any unexpected errors.
- **DEBUG**: The DEBUG level logging includes generated SQL statements that the module generates internally.

The following example contains the code to run the application with logging.

```
# To run the application with Nosql module logging at DEBUG level
$ java -cp $CP:target/example-spring-data-oracle-nosql-1.3-
SNAPSHOT.jar
    -Dlogging.level.com.oracle.nosql.spring.data=DEBUG org.example.App
...
020-12-02 11:50:18.426 DEBUG 20325 --- [ main]
    c.o.n.spring.data.core.NosqlTemplate : DDL: CREATE TABLE IF NOT
EXISTS
        StudentTable (id LONG GENERATED ALWAYS as IDENTITY (NO CYCLE),
        kv_json_ JSON, PRIMARY KEY( id ))
2020-12-02 11:50:19.334 INFO 20325 --- [ main]
    org.example.App : Started App in 2.464 seconds (JVM running for
2.782)
=== Start of App ====
```

```
2020-12-02 11:50:19.340 DEBUG 20325 --- [ main]
    c.o.n.spring.data.core.NosqlTemplate : Q: DELETE FROM StudentTable
Saving s1: Student{id=0, firstName='John', lastName='Doe'}
2020-12-02 11:50:19.362 DEBUG 20325 --- [ main]
    c.o.n.spring.data.core.NosqlTemplate : execute insert in table
StudentTable
Saving s2: Student{id=0, firstName='John', lastName='Smith'}
2020-12-02 11:50:19.387 DEBUG 20325 --- [ main]
    c.o.n.spring.data.core.NosqlTemplate : execute insert in table
StudentTable

findAll:
2020-12-02 11:50:19.392 DEBUG 20325 --- [ main]
    c.o.n.spring.data.core.NosqlTemplate : Q: SELECT * FROM StudentTable t
Student: Student{id=1, firstName='John', lastName='Doe'}
Student: Student{id=2, firstName='John', lastName='Smith'}

findByLastName: Smith
2020-12-02 11:50:19.412 DEBUG 20325 --- [ main]
    c.o.n.spring.data.core.NosqlTemplate : Q: declare $p_lastName String;
        select * from StudentTable as t where t.kv_json_.lastName
= $p_lastName
Student: Student{id=2, firstName='John', lastName='Smith'}
2020-12-02 11:50:19.426 DEBUG 20325 --- [ main]
    c.o.n.spring.data.core.NosqlTemplate : DDL: DROP TABLE IF EXISTS
StudentTable
=== End of App ====

# To enable Nosql module logging when running tests
$ mvn test -Dlogging.level.com.oracle.nosql.spring.data=DEBUG
...
```

Glossary

Index