Oracle® NoSQL Database

Spring Data SDK Developers Guide

2.1.0
F58555-20
September 2024

Oracle NoSQL Database Spring Data SDK Developers Guide, 2.1.0

F58555-20

Primary Author: Vandana Rajamani

Contents

List of Examples

List of Figures

List of Tables

1

Introduction to Oracle NoSQL Database SDK for Spring Data

Learn about the Spring Data Framework and Oracle NoSQL Database SDK for Spring Data.

Oracle NoSQL Database SDK for Spring Data provides a Spring Data implementation module to connect to an Oracle NoSQL Database cluster or to Oracle NoSQL Database Cloud Service.

In the following sections, you will learn about the Spring Data Framework (Spring-based programming model for data) and how to access the Oracle NoSQL Database using the Oracle NoSQL Database SDK for Spring 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 about 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 the Maven Central repository, details are available here. The main location of the project is is the `oracle-spring-sdk` project on GitHub.

You can get all the required files for running the Spring Data Framework with the following POM file dependencies. The version changes with each release. Ensure that you install the latest supported version as suggested in the GitHub.

```
<dependencies>
    <dependency>
        <groupId>com.oracle.nosql.sdk</groupId>
        <artifactId>spring-data-oracle-nosql</artifactId>
        <version>2.1.0</version>
    </dependency>
</dependencies>
```

Add the additional dependency to use the Spring Data Framework:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>3.3.4</version>
</dependency>
```

The Oracle NoSQL Database SDK for Spring Data provides you with all the Spring Data classes, methods, interfaces, and examples. Documentation is available as nosql-spring-sdk in GitHub or from SDK for Spring Data API Reference.

> **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 about 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.

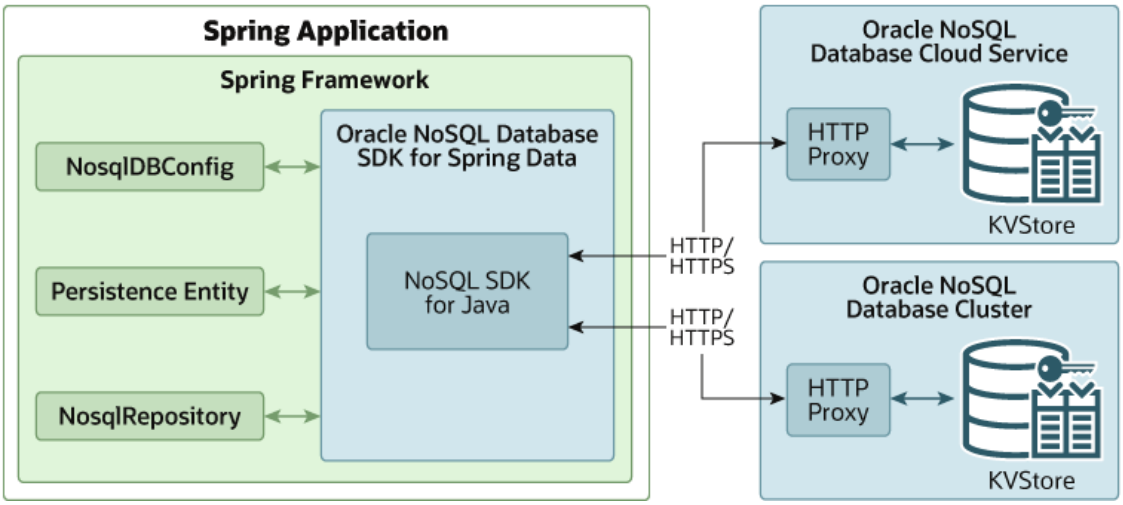
- Generic CRUD operations on a repository using methods in the `CrudRepository` interface. For more information about the `CrudRepository` interface, see CrudRepository.
- Pagination and sorting operations using methods in the `PagingAndSortingRepository` interface. For more information about the `PagingAndSortingRepository` interface, see PagingAndSortingRepository.
- Derived Queries.
- Native Queries.

Components of Oracle NoSQL Database SDK for Spring Data

Learn about the modules of Oracle NoSQL Database SDK for Spring Data.

The Oracle NoSQL Database Proxy must 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*. After setting up the proxy, you 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



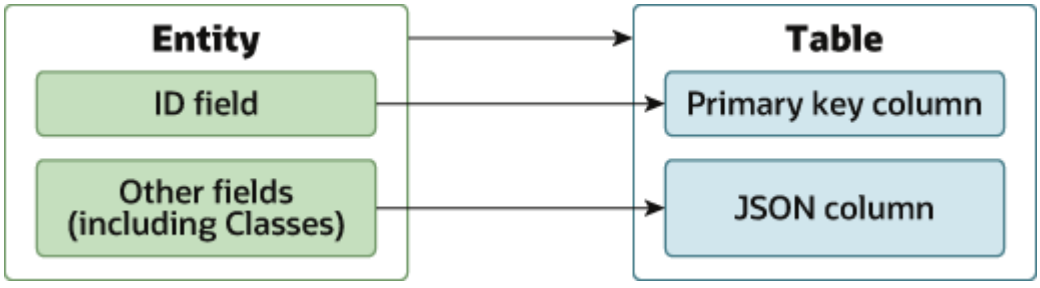
Persistence Model

Learn about the schema of the persistence table and Spring Data Framework annotations.

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 about JSON Column mappings, see JSON Column.

The following is the syntax of an entity with `@NosqlTable` and `@NosqlId` annotations. In the following example, 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 must instantiate the entity classes. These classes must 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 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 can be provided.

Table 1-1 Attributes in NosqlTable Annotation

Parameter	Type	Ignored/ Optional/ Required in Oracle NoSQL Database	Ingnored/ Optional/ Required in Oracle NoSQL Database Cloud Service	Default	Description
`tableName`	`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 about 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 about using compartments, see Creating a Compartment in the *Oracle NoSQL Database Cloud Service* Guide.

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
`autoCreateTable`	`boolean`	Optional	Optional	`true`	Specifies if the table must be created if it does not exist.

Note:

The Spring Data Framework looks for the repositories used in the application in the `init` phase. If the table does not exist, and if the `@NosqlTable` annotation has the `autoCreateTable` 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 about `readUnits`, see Plan your service in the *Oracle NoSQL Database Cloud Service*. **Note:** In Oracle NoSQL Database Cloud Service, you must set the `readUnits` parameter to a value greater than `0`. If you do not set the value, the Oracle NoSQL Database Cloud Service 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 about `writeUnits`, see Plan your service in the *Oracle NoSQL Database Cloud Service*. **Note:** In Oracle NoSQL Database Cloud Service, you must set the `writeUnits` parameter to a value greater than `0`. If you do not set the value, the Oracle NoSQL Database Cloud Service 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, permitted for the table, if the table is to be created. For more information about `storageGB`, see Plan your service in the *Oracle NoSQL Database Cloud Service*. **Note:** In Oracle NoSQL Database Cloud Service, you must set the `storageGB` parameter to a value greater than `0`. If you do not set the value, the Oracle NoSQL Database Cloud Service 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 permitted 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 about getting the timeout from `NoSQLHandleConfig` class using the `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*.
durability	String	Optional	Optional	`COMMIT_NO_SYNC`	Sets the default durability for all the write operations applied to this table. Valid values based on `oracle.nosql.driver.Durability` are `COMMIT_NO_SYNC`, `COMMIT_SYNC`, and `COMMIT_WRITE_NO_SYNC`. See Durability in the *Java SDK 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
capacityMode	NosqlCapacityMode For more information, see NosqlCapacityMode.	Optional	Optional	`NosqlCapacityMode.PROVISIONED`	Sets the capacity mode when the table is created. This applies only in cloud or cloud sim scenarios. A table is created with either `Provisioned Capacity` or `On-Demand Capacity`. For more details, see Cloud Concepts in the *Oracle NoSQL Database Cloud Service*. Set the values for the TableLimits instance based on the capacity mode as follows: • Set `capacityMode` to PROVISIONED and all three `TableLimits: readUnits, writeUnits,` and `storageGB` to values greater than 0. • Set `capacityMode` to ON_DEMAND and `storageGB` to a value greater than 0.
ttl	int	Optional	Optional	`0`	Sets the default table level Time to Live (TTL) when the table is created. The TTL enables the automatic expiration of table rows after the elapse of the specified duration. If the value is not set, the value `Constants.NOTSET_TABLE_TTL` is used, that is, table-level TTL is not applicable. See NOTSET_TABLE_TTL in the SDK for Spring Data API Reference. This parameter is applicable only when autoCreateTable is set to true.
ttlUnit	TtlUnit	Optional	Optional	`NosqlTable.TtlUnit.DAYS`	Sets the unit of TTL value. The valid values are: `NosqlTable.TtlUnit.DAYS` and `NosqlTable.TtlUnit.HOURS`. If the value is not set, the default value of days is used. This parameter is applicable only when autoCreateTable is set to true.

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 the ID field before storing it 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.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 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 preceding 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 the `@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_`. This is because the data column of the table created by the Spring Data Framework will be named `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 autogenerated or not. • If `true`, then it is defined as autogenerated 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. • If `false`, then the value must be managed by your application.

> **Note:**
>
> You can't autogenerate composite keys. Setting `@NosqlId.autoGenerated=true` leads to an error. You must manage the key values for all read/write calls when using the composite keys. If the key values are not set, the Oracle NoSQL Database generates an error.

Composite Primary Keys

Composite primary keys contain more than one primary key field. You can define a composite key class type to represent the composite keys.

A composite key class is a type that is mapped to multiple primary key fields of the entity class. A composite key class must be serializable and must define equals and hashcode methods. This class must consist of fields that are primitive data types.

> **Note:**
>
> The equality checks for the user-defined methods in the composite key class must be consistent with the equality checks performed in the Oracle NoSQL Database between the database types and their mapped keys.

You can use `@NosqlKey` annotation to specify the components of a composite primary key in the composite key class.

Table 1-4 Attributes in the Nosqlkey Annotation

Parameter	Type	Optional/Required	Default	Description
`shardKey`	`boolean`	`Optional`	`true`	Identifies if a primary key field is also a Shard Key. Shard keys affect the distribution of rows across shards. • If `true`, the Spring Data Framework considers the primary key field as a part of the shard key. • If `false`, the primary key field is not a part of the shard key. If you do not supply the `shardKey` parameter in the `Nosqlkey` annotation, the Spring Data Framework creates the primary key field as a shard key.

Table 1-4 (Cont.) Attributes in the Nosqlkey Annotation

Parameter	Type	Optional/Required	Default	Description
`order`	`int`	`Optional`	System determined	Specifies the ordering of the shard keys and non-shard keys within the primary key in a composite key class. You can set the `order` value based on the following rules, otherwise, the Spring Data Framework generates an error. • The order of the shard keys must be less than the order of the non-shard primary key fields. • The order must be specified for all the primary key fields or none. The Spring Data Framework does not support specifying the order for a partial list of primary key fields. • The `order` value of each primary key field must be unique. If you do not specify the `order` parameter in the `Nosqlkey` annotation, the Spring Data Framework orders shard keys and non-shard keys individually in the alphabetic order of the field names. See *Ordering the composite keys* example.

For more details on `@NosqlKey` annotation, see NosqlKey in the *SDK for Spring Data API Reference* document.

Example: Ordering the composite keys

Consider primary key fields *universityId*, *academicYear*, and *studentId* defined in a composite key class.

You can define the `universityId` and `academicYear` fields to be a part of the shard key. The order values of these shard keys must be lesser than the `studentId` field, which is a non-shard key. You can use the following sample code to create a composite class.

```
/* Define a composite Key class */

public class StudentKey implements Serializable {

    @NosqlKey(shardKey = true, order = 1)
    long universityId;

    @NosqlKey(shardKey = true, order = 0)
    int academicYear;

    @NosqlKey(shardKey = false, order = 2)
    long studentId;

    /* public or package protected constructor required when retrieving from
database */
    public StudentKey() {
    }
}
```

In the preceding example, the `academicYear` field is considered as the first primary key field during the creation of the table.

The Spring Data Framework creates the table with the following DDL:

```
/* Table DDL */

CREATE TABLE IF NOT EXISTS Students (
   academicYear INTEGER,
   universityId LONG,
   studentId LONG,
   kv_json_ JSON,
   PRIMARY KEY(SHARD(academicYear, universityId), studentId)
)
```

Consider a composite key class without specifying the `order` field.

```
/* Define a composite Key class */

public class StudentKey implements Serializable {

    @NosqlKey(shardKey = true)
    long universityId;

    @NosqlKey(shardKey = true)
```

```
    int academicYear;

    @NosqlKey(shardKey = false)
    long studentId;

    @NosqlKey(shardKey = false)
    long branchId;

    /* public or package protected constructor required when retrieving from
database */
    public StudentKey() {
    }
}
```

In the preceding example, the Spring Data Framework creates the shard keys and non-shard keys in the alphabetic order of the field names within the primary key. The table DDL is as follows:

```
/* Table DDL */

CREATE TABLE IF NOT EXISTS Students (
   academicYear INTEGER,
   universityId LONG,
   branchId LONG,
   studentId LONG,
   kv_json_ JSON,
   PRIMARY KEY(SHARD(academicYear, universityId), branchId, studentId)
)
```

In the following cases, the Spring Data Framework considers all the primary key fields as shard keys and uses alphabetical ordering:

- If you declare the primary key fields in the composite key class without using the `@NosqlKey` annotation.
- If you declare the primary key fields in the composite key class without specifying the `shardKey` and the `order` values in the `@NosqlKey` annotation.

Note the following properties of the composite key class.

- You must have at least one field with `shardKey=true` in the composite key class, otherwise, the Spring Data Framework will generate an error.
- You can use a composite key class with repositories (as the `ID` type) and to represent an entity's identity in a single object.
- You can annotate the fields as `@transient` to designate the nonpersistent state of the field.
- You can't nest composite key classes. This will generate an error.
- You can't autogenerate composite primary key fields. Setting `@NosqlId.autoGenerated=true` leads to an error. You must manage the key values for all read/write calls when using the composite keys. If the key values are not set, the Oracle NoSQL Database generates an error.

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 nonscalar 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 1-5 Mapping Between Java and NoSQL JSON Types

Java Type	Representation within Oracle NoSQL Database JSON data 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:
	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.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 time stamp encoded representation.

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

Table 1-5 (Cont.) Mapping Between Java and NoSQL JSON Types

Java Type	Representation within Oracle NoSQL Database JSON data type
`org.springframework.data.geo.Polygon`	GeoJson Polygon For more information about 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 1-5 (Cont.) Mapping Between Java and NoSQL JSON Types

Java Type	Representation within Oracle NoSQL Database JSON data type

`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:
	• A `java.util.ArrayList` object is instantiated for fields of type `java.util.Collection`, `java.util.List`, `java.util.AbstractList`, and `java.util.ArrayList`.
	• A `java.util.HashSet` object is instantiated for fields of type `java.util.Set`, `java.util.AbstractSet`, and `java.util.HashSet`.
	• A `java.util.TreeSet` object is instantiated for fields of type `java.util.SortedSet`, `java.util.NavigableSet`, and `java.util.TreeSet`.

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

`java.util.Map`
`java.util.NavigableMap`
`java.util.SortedMap`
`java.util.HashMap`
`java.util.LinkedHashMap`
`java.util.Hashtable`
`java.util.TreeMap`
	MAP(JSON)
	
	Note:
	• A `java.util.HashMap` is instantiated for fields of type `java.util.HashMap`
	• A `java.util.LinkedHashMap` is instantiated for fields of type `java.util.Map` and `java.util.LinkedHasMap`.
	• A `java.util.TreeMap` is instantiated for fields of type `java.util.NavigableMap`, `java.util.SortedMap`, and `java.util.TreeMap`.

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

Learn about how the Oracle NoSQL Database SDK for Spring Data handles transactions on Oracle NoSQL Database.

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 must think about ACID transactions as being supported for those repository methods that operate over single objects. Repository methods such as `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 performed based on the default Java driver durability. For more information about 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 performed based on the default eventual consistency or as specified in the `@NosqlTable` annotation. For more information about default eventual consistency, see getDefaultConsistency in the *Java SDK API Reference*.

Setting up the Connection

Learn how to set up a connection from Oracle NoSQL Database SDK for Spring Data to the Oracle NoSQL Database.

To expose the connection and security parameters to the Oracle NoSQL Database SDK for Spring Data, you must create a class that extends the `AbstractNosqlConfiguration` class. You can customize this code as required. 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 must be loaded before running the program. The `@EnableNoSQLRepositories` annotation tells the Spring Data Framework that it must 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 can 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 previously-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 about 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.

Example 1-1 Setting up the connection in a nonsecure data store

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 must provide the URL of the Oracle NoSQL Database Proxy with nonsecure 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
      must 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);
    }
}
```

Example 1-2 Setting up the connection in a secure data store

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
      must be registered as a bean in the Spring application.*/
    @Bean
    public NosqlDbConfig nosqlDbConfig() {
        AuthorizationProvider authorizationProvider;
        /* Provide the user name 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.

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

For more details on the security configuration, see Obtaining a NoSQL Handle.

Example 1-3 Setting up the connection in Oracle NoSQL Database Cloud Service

You can use different methods to connect to the Oracle NoSQL Database Cloud Service. For more details, see Connecting your Application to NDCS.

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*.

You require tenancy id, user id, and fingerprint information which can be found on the user profile page of the cloud account under the `User Information` tab on `View Configuration File`. You can also add the passphrase to your private key. For more details, see Authentication to connect to Oracle NoSQL Database.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import oracle.nosql.driver.kv.StoreAccessTokenProvider;
import oracle.nosql.driver.Region;
import oracle.nosql.driver.iam.SignatureProvider;
import java.io.File;

public class AppConfig extends AbstractNosqlConfiguration

{
    /* Annotation to tell that the returned object must be registered as a
bean in the Spring application.*/
    @Bean
    public NosqlDbConfig nosqlDbConfig()
    {
        SignatureProvider signatureProvider;
        char passphrase[] = < Pass phrase > ; // Optional. A passphrase for
the key, if it is encrypted.
```

```
        /* Details that are required to authenticate and authorize access to
the Oracle NDCS 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.
            new File( < privateKeyFile > ), // Full path to the key file.
            passphrase //Optional.
        );

        /* Provide the service URL of the Oracle NoSQL Database Cloud Service
*/
        /* Update the Region.<Region name>.endpoint() with the appropriate
value. */
        /* For example, Region.US_ASHBURN_1.endpoint() .*/

        return new NosqlDbConfig(Region. < Region name > .endpoint(),
signatureProvider);
    }
```

Defining a Repository

Learn about Oracle NoSQL Database SDK for Spring Data's `NosqlRepository` interface, which includes methods to save, delete, update individual entities and also findAll, deleteAll on sets of entities.

The entity class that is used for persistence is discoverable by the Spring Data Framework either via annotation or inheritance. The `NosqlRepository` interface enables 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 want to create your derived queries, this must be performed 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

Learn how to create a program to run the Spring boot application.

After creating the entity and repository, you must 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 can 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 class path for a class with the `@configuration` annotation and contains a method named `NosqlTemplate` with the `@Bean` annotation.

Queries

Learn about the types of queries supported by the Spring Data Framework.

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

Learn about the `PagingAndSortingRepository` interface supported by the Spring Data Framework.

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 about the Spring's `PagingAndSortingRepository` interface, see PagingAndSortingRepository.

Derived Queries

Learn about the customized query creation feature supported by the Spring Data Framework.

Apart from those query methods that are provided by Spring's `PagingAndSortingRepository` interface, you can 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 created automatically 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 must 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 about 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 can also have other keywords. The following section provides the detailed derived query snippets that will 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 about 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 about 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 created automatically 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 created automatically 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 created automatically by the Spring Data Framework.

For more information about `Page`, see Page. For more information about `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 created automatically by the Spring Data Framework.

For more information about `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);
}
```

Supported Keywords in Derived Queries

Learn about the keywords supported by the Spring Data Framework for prefixing the method names in derived queries.

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

Table 1-6 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 1-7 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()`

Table 1-7 (Cont.) Supported Keywords for Body

Body Keyword	No. of Parts	No. of Params	Example
`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
`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

Native Queries

Learn to run the native SQL queries using the `@oracle.spring.data.nosql.repository.Query` annotation.

The `@oracle.spring.data.nosql.repository.Query` annotation enables 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` subclasses 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 about `FieldValue`, see FieldValue.

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

Activating Logging

Learn to enable logs to capture the exceptions from Oracle NoSQL Database module.

The Spring Data errors are thrown as exceptions when you build or run your application.

For example, the Oracle NoSQL Database SDK for Spring Data uses `IllegalArgumentException` for invalid parameters and passes through Java SDK and Spring

Data Framework exceptions. The Spring Data Framework throws exceptions directly for some classes, such as `BeansException`.

You can add an `exception` code block to catch any error that might be thrown such as authentication failure during connection setup. Additionally, 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 can 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
...
```

You can enable additional logging and client statistics at the NoSQL Java SDK level. For more details, see *Logging in the SDK* and *Logging internal SDK statistics* in the oracle.nosql.driver package.

2

Develop Applications Using Oracle NoSQL Database SDK for Spring Data

Learn to create applications using Oracle NoSQL Database SDK for Spring Data.

The Oracle NoSQL Database SDK for Spring Data supports applications to access the NoSQL Database and perform database operations such as updating and deleting records, reading, index creation and removal, as well as queries. This section provides an overview of these capabilities.

Accessing Oracle NoSQL Database Using Spring Data Framework

Learn to access Oracle NoSQL Database from Spring using Oracle NoSQL Database SDK for Spring Data.

Using the Spring Data Framework, you can set up a connection with Oracle NoSQL Database nonsecure store, insert a row in a table, and then retrieve the data from the table.

Example 2-1 Accessing NoSQL Database using Spring Data Framework

The following example shows how to set up a Maven Project and then add the following classes/interfaces:

- `AppConfig` class
- `Student` class
- `StudentRepository` interface
- `App` class

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

1. **Setting up a Maven project:**
   Set up a Maven project with the required POM file dependencies. For details, see About the Oracle NoSQL Database SDK for Spring Data.
2. **Setting up an Appconfig class:**
   Set up the `AppConfig` class that extends the `AbstractNosqlConfiguration` class to provide a `NosqlDbConfig` Spring bean. The `NosqlDbConfig` Spring bean describes how to connect to the Oracle NoSQL 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
  must 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.

3. **Defining an entity class:**
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.

   Supply the `@NosqlId` annotation to indicate that the `id` field will act as the ID and be the primary key of the underlying storage table and `generated=true` attribute to specify that this ID will be autogenerated by a sequence.

   If the ID field type is a String, a UUID will be used. If the ID field type is integer or long, a "GENERATED ALWAYS as IDENTITY (NO CYCLE)" sequence is used.

   For details on all the Spring Data classes, methods, interfaces, and examples see SDK for Spring Data API Reference.

```
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 autogenerated 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 +
            '}';
    }
}
```

When a table is created through the Spring Data application, a schema is created automatically, which includes two columns - the primary key column (types String, integer, long, or timestamp) and a JSON column called `kv_json_`.

If a table exists already, it must comply with the generated schema.

> **Note:**
>
> - You can set the table level TTL by supplying the `ttl()` and `ttlUnit()` parameters in the `@NosqlTable` annotation of the entity class. For more details, see Setting TTL values.
> - You can set the default `TableLimits` for Oracle NoSQL Database Cloud Service tables in the `@NosqlDbConfig` instance using `NosqlDbConfig.getDefaultCapacityMode()`, `NosqlDbConfig.getDefaultStorageGB()`, `NosqlDbConfig.getDefaultReadUnits()`, and `NosqlDbConfig.getDefaultWriteUnits()` methods. The `TableLimits` can also be specified per table if the `@NosqlTable` annotation is used, through `capacityMode`, `readUnits`, `writeUnits`, and `storageGB` fields as shown in the following code sample.
>
>   ```
>   /* Set the TableLimits and TTL values. */
>   @NosqlTable(readUnits = 50, writeUnits = 50, storageGB = 25)
>   ```

4. **Declaring a repository that extends NosqlRepository:**
   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 parameterized types to the `NosqlRepository` interface. This `NosqlRepository` interface provides methods that can 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);
}
```

5. **Creating an application class:**
Code the functionality as required by implementing any of the various interfaces provided by the Spring Data Framework. This example uses the `CommandLineRunner` interface to show the application code that implements the run method and has the main method. For more information about setting up a Spring boot application, see Spring Boot.

   In Spring data applications, the tables are automatically created at the beginning of the application when the entities are initialized unless `@NosqlTable.autoCreateTable` is set to `false`.
   You can create the application code to insert data to the table, read the rows from the table, and also run the queries as follows:

   - **Adding and deleting data:** Use one of these methods to add rows to the table - `NosqlRepository.save(entity_object)`, `saveAll(Iterable<T> iterable)`, or `NosqlTemplate.insert(entity)`. To delete any exisiting rows from the table, you can use one of these methods - `NosqlRepository.deleteById()`, `delete()`, `deleteAll(Iterable<? extends T> entities)`, `deleteAll()` or use `NosqlTemplate.delete()`, `deleteAll()`, `deleteById()`, `deleteInShard()`.
     In the following code sample, you first delete the rows from the `Student` table using the `NosqlRepository.deleteAll()` method. This ensures the deletion of all the rows from the table if the table already preexists in the database. You then use the `NosqlRepository.save(entity_object)` method to add the rows to the table. You create and save two student entities.

     ```
     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
           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 {

        /* 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);
   }
}
```

The `@NosqlId` annotation in the `Student` entity class specifies that the `id` field will act as the ID and be the primary key of the underlying storage table. The rest of the entity fields, that is, the `firstName` and `lastName` fields are stored in the JSON column.

- **Reading data:** Use one of these methods to read the data from the table - `NosqlRepository.findById()`, `findAllById()`, `findAll()` or using `NosqlTemplate.find()`, `findAll()`, `findAllById()`.
  In the following code sample, you use the `NosqlRepository.findAll()` method to read all the rows from the table. You select all the rows from the `Student` table and supply them to an iterable instance. Print the values to the output from the iterable object.

```
   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);
     }
```

- **Using queries:** Use one of these methods to run your query - The `NosqlRepository` derived queries, native queries, or using `NosqlTemplate.runQuery()`, `runQueryJavaParams()`, `runQueryNosqlParams()`.

In following code sample, you use the derived queries to select a row from the `Student` table with the required last name and print the values to the output from the object. For more details on the derived queries, see Derived Queries.

```
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);
}
```

6. **Running the program:** Execute the application code samples from step 5. The Spring Data Framework adds rows to the `Student` table, searches for all the rows and prints the results to the output. It also fetches an individual row from the table.

```
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}
```

Modifying the table: To modify a table, you can use the `NosqlTemplate.runTableRequest()` method.

> **Note:**
>
> While the Oracle NoSQL Database SDK for Spring Data provides an option to modify the tables, it is not recommended to alter the schemas as the Spring Data Framework expects tables to comply with the default schema (two columns - the primary key column of types string, integer, long, or timestamp and a JSON column called kv_json_).

Setting TTL values

Learn to set table level Time To Live (TTL) from Oracle NoSQL Database SDK for Spring Data.

You can set the table level TTL by setting the following parameters in the `@NosqlTable` annotation of an entity class:

- `ttl()`: Sets the table level TTL value in either DAYS or HOURS. If not specified, the default value is set to 0, which means the TTL value is not set.
- `ttlUnit()`: Sets the TTL unit to either DAYS or HOURS. If not specified, the default value is set to DAYS.

Example 2-2 Setting table level TTL value using Spring Data Framework

The following example shows how to create the `Student` entity class and set the TTL value to 10 days.

When the `ttl()` value is provided in the `@NosqlTable` annotation, the Spring Data driver creates the `Student` table with the specified TTL value.

```
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. */

/* Sets the table level TTL to 10 Days. */
@NosqlTable(ttl =  10, ttlUnit = NosqlTable.TtlUnit.DAYS)

public class Student {
    /* The @NosqlId annotation specifies that this field will act as the ID
field.

    The generated=true attribute specifies that this ID will be autogenerated
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, lastname,
       and then returns a String. */
    @Override
    public String toString() {
        return "Student{" +
            "id=" + id + ", " +
            "firstName=" + firstName + ", " +
            "lastName=" + lastName +
            '}';
    }
}
```

Using SpEl expressions in `NosqlTable` annotation

Learn about using Spring Expression Language (SpEl) expressions in `@NosqlTable.tableName` annotation.

You can specify the name of the table by setting the `tableName` parameter in the `@NosqlTable` annotation. In the `Student` class example discussed in previous topics, since the `tableName` is not explicitly provided, by default an empty value is set and the entity class name is used as the name of the table by the Spring driver.

SpEl is a way to evaluate complex expressions at runtime. For more details, see Spring Expression Language.

The `@NosqlTable.tableName` parameter supports evaluating (SpEl) expressions. You can use the SpEL expressions while setting the `tableName` parameter in the `@NosqlTable` annotation as shown in the following examples. The expressions are evaluated dynamically at runtime.

Table 2-1 Using SpEL Expressions

SpEL expression in the `tableName` parameter	Description
@NosqlTable(tableName = "#{ systemProperties['sys_ns']}:Customer")	The `Customer` table is created in the namespace defined by JVM system property `sys_ns`. If the system property doesn't exist, the SpEl expression evaluates to empty string, in which case the table is created in the default namespace, `sysdefault`. The *systemProperties* attribute is a predefined variable. To run with the JVM system property use: `java -Dsys_ns=myCustomNamespace ...`
@NosqlTable(tableName = "#{ @environment.getProperty('ENV_NS')}:Customer")	The `Customer` table is created in the namespace defined by the environment property `ENV_NS`. If the environment variable doesn't exist the table is created in the default namespace, `sysdefault`. To run by setting environment property use: `ENV_NS=myCustomNamespace; java ...`
@NosqlTable(tableName = "${app.ns}:Customer")	The `Customer` table is created in the namespace defined by the `app.ns` property in `application.properties` resource file. An error is thrown if the property does not exist.
@NosqlTable(tableName = "${app.ns}:Customer")	The `Customer` table is created in the namespace defined by the `app.ns` property in `application.properties` resource file. If the property does not exist, the table is created in the namespace `ns2`.
@NosqlTable(tableName = "#{ systemProperties['sys_ns'] != null ? systemProperties['sys_ns'] : @environment.getProperty('ENV_NS') != null ? @environment.getProperty('ENV_NS') : '${app.ns:srcNs}' }:Customer")	In this example, the namespace is evaluated in the following order: 1. If the namespace defined by the JVM system property `sys_ns`. 2. If `sys_ns` is not available, then environment variable `ENV_NS` is tried. 3. If `ENV_NS` is not available, then the namespace defined by the `app.ns` property in `application.properties` resource file is tried. 4. If none of the previously mentioned namespaces are available, the `Customer` table is created in the `srcNs` namespace.
@NosqlTable(tableName = ":Customer")	The starting colon ':' is automatically ignored when SpEl expressions '#' and '$' are used and result is an "" empty string namespace. In this example, an error is returned since neither of them are present.

For more details on namespace management, see Introducing Namespaces in the *Java Direct Driver Developer's Guide*.

Example 2-3 Using SpEl expressions in the table name

The following example shows how to create the `Student` entity class and provide the table name as `Customer` in the namespace (JVM system property `sys_ns`) using the `@NosqlTable` annotation.

The Spring Data driver evaluates the SpEL expressions and the `Customer` table is created in `sys_ns` namespace. If the namespace does not exist, the table is created in the `sysdefault` namespace.

```
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. */

/* Sets the table name. */
@NosqlTable(tableName = "#{ systemProperties['sys_ns']}:Customer")

public class Student {
    /* The @NosqlId annotation specifies that this field will act as the ID
field.
       The generated=true attribute specifies that this ID will be
autogenerated 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, lastname,
       and then returns a String. */
    @Override
    public String toString() {
        return "Student{" +
            "id=" + id + ", " +
            "firstName=" + firstName + ", " +
            "lastName=" + lastName +
            '}';
    }
}
```

Creating Tables with Composite Keys

Learn to create a table with composite primary key fields using Oracle NoSQL Database SDK for Spring Data.

You use the `@NosqlKey` annotation to identify the annotated field as a component of the composite primary key.

Example 2-4 Creating a table with composite primary key fields

The following examples shows how to model `Student` as an entity and use *universityId*, *academicYear*, and *studentId* fields as composite keys.

A Composite key is helpful when you want to use more than one primary key field conjointly to identify a unique row. Within the composite key, you can identify the primary key fields that can be a part of the Shard Key and also specify the ordering of the fields.

The following example shows how to create a composite key with the key fields from student data. You define a class named `StudentKey` to represent the composite key class and then use that in the `Student` entity as described in the following example:

Create a composite key class with the `@NosqlKey` annotation to identify the composite keys. Set the `shardKey` value to `true` if the field is a part of the shard key. Set the `order` value for all the fields in the order of primary key field generation in the table. For more details on the `shardKey` and `order` elements, see Table 1-4.

In the following code sample, the `universityId`, `academicYear`, and `studentId` fields represent the key fields for identifying a student's data and are declared as the primary key fields using the `@NosqlKey` annotation. For an illustration of ordering within the primary key fields, consider two of the primary key fields as shard keys and the third as a non-shard key.

Set the `shardKey` value of the `universityId` and `academicYear` fields to `true`, and the `studentId` field to false. Set the order value for the `universityId` field to 0 to create the `universityId` field as the first primary key field and `academicYear` to 1 to create as second primary key field. As the `studentId` field is a non-shard key, its order value must be higher than the shard keys. Set the `order` value of the `studentId` field to 2.

```
import com.oracle.nosql.spring.data.core.mapping.NosqlKey;
import java.io.Serializable;
import java.util.Objects;

/* Define a composite Key class */

public class StudentKey implements Serializable {

    @NosqlKey(shardKey = true, order = 0)
    long universityId;

    @NosqlKey(shardKey = true, order = 1)
    int academicYear;

    @NosqlKey(shardKey = false, order = 2)
    long studentId;

    /* public or package protected constructor required when retrieving from
database */
    public StudentKey() {
    }

    public StudentKey(long universityId, int academicYear, long studentId) {
        this.universityId = universityId;
        this.academicYear = academicYear;
        this.studentId = studentId;
    }
```

```
    public long getUniversityId() {
        return universityId;
    }

    public void setUniversityId(long universityId) {
        this.universityId = universityId;
    }

    public int getAcademicYear() {
        return academicYear;
    }

    public void setAcademicYear(int academicYear) {
        this.academicYear = academicYear;
    }

    public long getStudentId() {
        return studentId;
    }

    public void setStudentId(long studentId) {
        this.studentId = studentId;
    }

    /* Define equals method */
    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof StudentKey)) {
            return false;
        }
        StudentKey studentKey = (StudentKey) o;
        return Objects.equals(universityId, studentKey.universityId) &&
                Objects.equals(academicYear, studentKey.academicYear) &&
                Objects.equals(studentId, studentKey.studentId);
    }

    /* Define hashcode method */
    @Override
    public int hashCode() {
        return Objects.hash(universityId, academicYear, studentId);
    }
}
```

Create the `Student` entity class with `StudentKey` as a composite primary key. The `StudentKey` is annotated with `@NosqlId` in the entity class to indicate the primary key.

You can declare any non-key fields in the entity class. The non-key fields will be included as JSON data in the `kv_json_` column.

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

```
import java.io.Serializable;
import java.util.Objects;

/*The @NosqlTable annotation specifies that
  this class will be mapped to an Oracle NoSQL Database table.*/

@NosqlTable

public class Student {
    @NosqlId
    StudentKey studentKey;
    String firstName;
    String lastName;
    String resident;

    /* public or package protected constructor required when retrieving from
database */
    public Student() {
        studentKey = new StudentKey();
    }


    /*This method overrides the toString() method, and then concatenates id
and name, and then returns a String*/
    @Override
    public String toString() {
        return "Student{" +
                "universityId=" + studentKey.universityId + ", " +
                "academicYear=" + studentKey.academicYear + ", " +
                "studentId=" + studentKey.studentId + ", " +
                "firstName=" + firstName + ", " +
                "lastName=" + lastName + ", " +
                "resident=" + resident+
                '}';
    }
}
```

> **Note:**
>
> You must set up the `AppConfig` class that provides a `NosqlDbConfig` Spring bean. The `NosqlDbConfig` Spring bean describes how to connect to the Oracle NoSQL Database. You must also create an interface that extends the `NosqlRepository` interface to retrieve the data from the Oracle NoSQL Database. For details, see the section Accessing Oracle NoSQL Database Using Spring Data Framework.

The Spring Data Framework creates the `Student` table with the following DDL:

```
/* Student table DDL */

CREATE TABLE IF NOT EXISTS Student (
   universityId LONG,
   academicYear INTEGER,
   studentId LONG,
```

```
   kv_json_ JSON,
   PRIMARY KEY(SHARD(universityId, academicYear), studentId)
)
```

The primary key fields `universityId` and `academicYear` are also shard keys and `studentId` is a non-shard primary key field.

Creating an Index

Learn to create an index on a field in a Oracle NoSQL Database table using Oracle NoSQL Database SDK for Spring Data.

To create an index on a field in an Oracle NoSQL Database table from the Spring Data Framework, you use `NosqlTemplate.runTableRequest()` method.

In the application, you instantiate the `NosqlTemplate` class by providing the `NosqlTemplate.create(NosqlDbConfig nosqlDBConfig)` method with the instance of the `AppConfig` class. You then modify the table using the `NosqlTemplate.runTableRequest()` method. You provide the NoSQL statement for the index creation in the `NosqlTemplate.runTableRequest()` method.

Example 2-5 Creating an Index on a table using Spring Data Framework

The following example shows how to create an index on the `lastName` field in the `Student` table.

```
/* Create an Index on the lastName field of the Users Table. */

try {
    AppConfig config = new AppConfig();
    NosqlTemplate idx = NosqlTemplate.create(config.nosqlDbConfig());
    idx.runTableRequest("CREATE INDEX IF NOT EXISTS nameIdx ON
Student(kv_json_.lastName AS STRING)");
    System.out.println("Index created successfully");
} catch (Exception e) {
    System.out.println("Exception creating index" + e);
}
```

For details on table creation, see Accessing Oracle NoSQL Database Using Spring Data Framework.

Projections

Learn to use Projections to customize a part of the entity class.

Use Projections when the required result is a subset of an entity, that is when the required result is 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.

Example 2-6 Using Projections

The following example defines projections in the context of `Student` entity class. See 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. You can use the new types (`StudentView` and `StudentProjection`) 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 must contain duplicates. If these duplicates are not required then you can use the `Distinct` keyword to eliminate them as follows:

```
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 Oracle NoSQL Database 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 call 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 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
```

Dropping Tables and Indexes

Learn to drop Oracle NoSQL Database tables and indexes using Oracle NoSQL Database SDK for Spring Data.

To drop the tables and indexes on the fields in tables, you use `NosqlTemplate.runTableRequest()` or `NosqlTemplate.dropTableIfExists()` methods.

Create the `AppConfig` class that extends `AbstractNosqlConfiguration` class to provide the connection details of the database. For details, see Accessing Oracle NoSQL Database Using Spring Data Framework.

In the application, you instantiate the `NosqlTemplate` class by providing the `NosqlTemplate.create(NosqlDbConfig nosqlDBConfig)` method with the instance of the `AppConfig` class. You then drop the table using the `NosqlTemplate.dropTableIfExists()` method. The `NosqlTemplate.dropTableIfExists()` method drops the table and returns `true` if the result indicates a change of the table's state to DROPPED or DROPPING.

Example 2-7 Dropping Tables and Indexes using Spring Data Framework

The following code sample shows how to drop the `Student` table.

```
try {
    AppConfig config = new AppConfig();
    NosqlTemplate tabledrop = NosqlTemplate.create(config.nosqlDbConfig());
    Boolean result = tabledrop.dropTableIfExists("Student");
    if (result == true) {
        System.out.println("Table dropped successfully");
    } else {
        System.out.println("Failed to drop table");
    }
} catch (Exception e) {
    System.out.println("Exception creating index" + e);
}
```

Glossary

Index