Spring Data Key Value - Reference Documentation

Oliver Gierke, Thomas Darimont, Christoph Strobl

Version 1.0.0.RELEASE, 2015-09-01

Table of Contents

Preface

Chapter 1. Project metadata

- Version control - http://github.com/spring-projects/spring-data-keyvalue
- Bugtracker - https://jira.spring.io/browse/DATAKV
- Release repository - https://repo.spring.io/libs-release
- Milestone repository - https://repo.spring.io/libs-milestone
- Snapshot repository - https://repo.spring.io/libs-snapshot

Reference Documentation

Chapter 2. Key Value Repositories

This chapter explains concepts and usage patterns when working with the key value abstraction and the `java.util.Map` based implementation provided by Spring Data Commons.

2.1. Core Concepts

The Key/Value abstraction within Spring Data Commons requires an `Adapter` shielding the native store implementation freeing up `KeyValueTemplate` to work on top of any key/value pair like structure. Keys are distributed across Keyspaces. Unless otherwise specified the class name is used as the default keyspace for an entity.

```
interface KeyValueOperations {

    <T> T insert(T objectToInsert);                               ①

    void update(Object objectToUpdate);                           ②

    void delete(Class<?> type);                                   ③

    <T> T findById(Serializable id, Class<T> type);               ④

    <T> Iterable<T> findAllOf(Class<T> type);                     ⑤

    <T> Iterable<T> find(KeyValueQuery<?> query, Class<T> type);  ⑥

    //... more functionality omitted.

}
```

① Inserts the given entity and assigns id if required.

② Updates the given entity.

③ Removes all entities of matching type.

④ Returns the entity of given type with matching id.

⑤ Returns all entities of matching type.

⑥ Returns a List of all entities of given type matching the criteria of the query.

2.2. Configuring The KeyValueTemplate

In its very basic shape the `KeyValueTemplate` uses a `MapAdaper` wrapping a `ConcurrentHashMap` using Spring Expression Language to perform queries and sorting.

NOTE The used `KeyValueAdapter` does the heavy lifting when it comes to storing and retrieving data. The data structure used will influence performance and/or multi threading behavior.

One may choose to use a different type or preinitialize the adapter with some values, and can do so using various constructors on `MapKeyValueAdapter`.

```
@Bean
public KeyValueOperations keyValueTemplate() {
  return new KeyValueTemplate(keyValueAdapter());
}

@Bean
public KeyValueAdapter keyValueAdapter() {
  return new MapKeyValueAdapter(ConcurrentHashMap.class);
}
```

2.3. Keyspaces

Keyspaces define in which part of the data structure the entity should be kept. So this is a rather similar concept as collections in MongoDB and Elasticsearch, Cores in Solr, Tables in JPA. By default the keyspace of an entity is extracted form its type, but one can also choose to store entities of different types within one keyspace. In that case any find operation will type check results.

```
@KeySpace("persons")
class Person {

  @Id String id;
  String firstname;
  String lastname;
}

class User extends Person {
  String username;
}

template.findAllOf(Person.class); ①
template.findAllOf(User.class);   ②
```

① Returns all entities for keyspace "persons".

② Returns only elements of type `User` stored in keyspace "persons".

2.3.1. Custom KeySpace Annotation

It is possible to compose own `KeySpace` annotations for a more domain centric usage by annotating one of the attibutes with `@KeySpace`.

NOTE The composed annotation needs to inherit `@Persistent`.

```
@Persistent
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE })
static @interface CacheCentricAnnotation {

  @KeySpace String cacheRegion() default "";
}

@CacheCentricAnnotation(cacheRegion = "customers")
class Customer {
  //...
}
```

2.4. Querying

Query execution is managed by the `QueryEngine`. As mentioned before it is possible to instruct the `KeyValueAdapter` to use an implementation specific `QueryEngine` that allows access to native functionality. When used without further customization queries are be executed using a `SpELQueryEngine`.

NOTE For performance reasons, we highly recommend to have at least Spring 4.1.2 or better to make use of compiled SpEL Expressions. Please use the `-Dspring.expression.compiler.mode=IMMEDIATE` switch to turn it on.

```
KeyValueQuery<String> query = new KeyValueQuery<String>("lastname == 'targaryen'");
List<Person> targaryens = template.find(query, Person.class);
```

WARNING Please note that you need to have getters/setters present to query properties using SpEL.

2.5. Sorting

Depending on the store implementation provided by the adapter entities might already be stored in some sorted way but do not necessarily have to be. Again the underlying `QueryEngine` is capable of performing sort operations. When used without further customization sorting is done using a

`SpelPropertyComperator` extracted from the `Sort` clause provided

```
KeyValueQuery<String> query = new KeyValueQuery<String>("lastname == 'baratheon'");
query.setSort(new Sort(DESC, "age"));
List<Person> targaryens = template.find(query, Person.class);
```

WARNING Please note that you need to have getters/setters present to sort using SpEL.

2.6. Map Repositories

Map repositories reside on top of the `KeyValaueTemplate`. Using the default `SpelQueryCreator` allows deriving query and sort expressions from the given methodname.

```
@Configuration
@EnableMapRepositories
class KeyValueConfig {

}

interface PersonRepository implements CrudRepository<Person, String> {
    List<Person> findByLastname(String lastname);
}
```

Appendix