# Spring Data Redis

Costin Leau, Jennifer Hickey, Christoph Strobl, Thomas Darimont, Mark Paluch

Version 2.1.0.BUILD-SNAPSHOT, 2017-12-01

# Table of Contents

# Preface

The Spring Data Redis project applies core Spring concepts to the development of solutions using a key-value style data store. We provide a "template" as a high-level abstraction for sending and receiving messages. You will notice similarities to the JDBC support in the Spring Framework.

# Chapter 1. New Features

New and noteworthy in the latest releases.

## 1.1. New in Spring Data Redis 2.1

- Unix domain socket connections using Lettuce.
- Write to Master read from Slave support using Lettuce.

## 1.2. New in Spring Data Redis 2.0

- Upgrade to Java 8.
- Upgrade to Lettuce 5.0.
- Removed support for SRP and JRedis drivers.
- Reactive connection support using Lettuce.
- Introduce Redis feature-specific interfaces for `RedisConnection`.
- Improved `RedisConnectionFactory` configuration via `JedisClientConfiguration` and `LettuceClientConfiguration`.
- Revised `RedisCache` implementation.
- Add `SPOP` with count command for Redis 3.2.

## 1.3. New in Spring Data Redis 1.8

- Upgrade to Jedis 2.9.
- Upgrade to `Lettuce` 4.2 (Note: Lettuce 4.2 requires Java 8).
- Support for Redis GEO commands.
- Support for Geospatial Indexes using Spring Data Repository abstractions (see Geospatial Index).
- `MappingRedisConverter` based `HashMapper` implementation (see Hash mapping).
- Support for `PartialUpdate` in repository support (see Persisting Partial Updates).
- SSL support for connections to Redis cluster.
- Support for `client name` via `ConnectionFactory` when using Jedis.

## 1.4. New in Spring Data Redis 1.7

- Support for RedisCluster.
- Support for Spring Data Repository abstractions (see Redis Repositories).

## 1.5. New in Spring Data Redis 1.6

- The `Lettuce` Redis driver switched from wg/lettuce to mp911de/lettuce.
- Support for `ZRANGEBYLEX`.
- Enhanced range operations for `ZSET` s including `+inf` / `-inf`.
- Performance improvements in `RedisCache` now releasing connections earlier.
- Generic Jackson2 `RedisSerializer` making use of Jackson's polymorphic deserialization.

## 1.6. New in Spring Data Redis 1.5

- Add support for Redis HyperLogLog commands `PFADD`, `PFCOUNT` and `PFMERGE`.
- Configurable `JavaType` lookup for Jackson based `RedisSerializers`.
- `PropertySource` based configuration for connecting to Redis Sentinel (see: Redis Sentinel Support).

# Introduction

This document is the reference guide for Spring Data Redis (SDR) Support. It explains Key Value module concepts and semantics and the syntax for various stores namespaces.

For an introduction to key value stores or Spring, or Spring Data examples, please refer to Getting Started - this documentation refers only to Spring Data Redis Support and assumes the user is familiar with the key value storages and Spring concepts.

# Chapter 2. Why Spring Data Redis?

The Spring Framework is the leading full-stack Java/JEE application framework. It provides a lightweight container and a non-invasive programming model enabled by the use of dependency injection, AOP, and portable service abstractions.

NoSQL storages provide an alternative to classical RDBMS for horizontal scalability and speed. In terms of implementation, Key Value stores represent one of the largest (and oldest) members in the NoSQL space.

The Spring Data Redis (or SDR) framework makes it easy to write Spring applications that use the Redis key value store by eliminating the redundant tasks and boiler plate code required for interacting with the store through Spring's excellent infrastructure support.

# Chapter 3. Requirements

Spring Data Redis 1.x binaries requires JDK level 6.0 and above, and Spring Framework 5.0.2.RELEASE and above.

In terms of key value stores, Redis 2.6.x or higher is required. Spring Data Redis is currently tested against the latest 3.2 release.

# Chapter 4. Getting Started

Learning a new framework is not always straight forward. In this section, we (the Spring Data team) tried to provide, what we think is, an easy to follow guide for starting with the Spring Data Redis module. Of course, feel free to create your own learning 'path' as you see fit and, if possible, please report back any improvements to the documentation that can help others.

## 4.1. First Steps

As explained in Why Spring Data Redis?, Spring Data Redis (SDR) provides integration between Spring framework and the Redis key value store. Thus, it is important to become acquainted with both of these frameworks (storages or environments depending on how you want to name them). Throughout the SDR documentation, each section provides links to resources relevant however, it is best to become familiar with these topics beforehand.

### 4.1.1. Knowing Spring

Spring Data uses heavily Spring framework's core functionality, such as the IoC container, resource abstract or AOP infrastructure. While it is not important to know the Spring APIs, understanding the concepts behind them is. At a minimum, the idea behind IoC should be familiar. That being said, the more knowledge one has about the Spring, the faster she will pick up Spring Data Redis. Besides the very comprehensive (and sometimes disarming) documentation that explains in detail the Spring Framework, there are a lot of articles, blog entries and books on the matter - take a look at the Spring Guides home page for more information. In general, this should be the starting point for developers wanting to try Spring DR.

### 4.1.2. Knowing NoSQL and Key Value stores

NoSQL stores have taken the storage world by storm. It is a vast domain with a plethora of solutions, terms and patterns (to make things worse even the term itself has multiple meanings). While some of the principles are common, it is crucial that the user is familiar to some degree with the stores supported by SDR. The best way to get acquainted with these solutions is to read their documentation and follow their examples - it usually doesn't take more then 5-10 minutes to go through them and if you are coming from an RDMBS-only background many times these exercises can be an eye opener.

### 4.1.3. Trying Out The Samples

One can find various samples for key value stores in the dedicated example repo, at http://github.com/spring-projects/spring-data-keyvalue-examples. For Spring Data Redis, of interest is the `retwisj` sample, a Twitter-clone built on top of Redis which can be run locally or be deployed into the cloud. See its documentation, the following blog entry or the live instance for more information.

## 4.2. Need Help?

If you encounter issues or you are just looking for advice, feel free to use one of the links below:

### 4.2.1. Community Support

The Spring Data tag on Stackoverflow is a message board for all Spring Data (not just Redis) users to share information and help each other. Note that registration is needed **only** for posting.

### 4.2.2. Professional Support

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

## 4.3. Following Development

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

You can help make Spring Data best serve the needs of the Spring community by interacting with developers on Stackoverflow at either spring-data or spring-data-redis.

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

To stay up to date with the latest news and announcements in the Spring eco system, subscribe to the Spring Community Portal.

Lastly, you can follow the Spring blog or the project team (@SpringData) on Twitter.

# Reference Documentation

## Document structure

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

Redis support introduces the Redis module feature set.

# Chapter 5. Redis support

One of the key value stores supported by Spring Data is Redis. To quote the project home page:

> Redis is an advanced key-value store. It is similar to memcached but the dataset is not volatile, and values can be strings, exactly like in memcached, but also lists, sets, and ordered sets. All this data types can be manipulated with atomic operations to push/pop elements, add/remove elements, perform server side union, intersection, difference between sets, and so forth. Redis supports different kind of sorting abilities.

Spring Data Redis provides easy configuration and access to Redis from Spring applications. It offers both low-level and high-level abstractions for interacting with the store, freeing the user from infrastructural concerns.

## 5.1. Redis Requirements

Spring Redis requires Redis 2.6 or above and Java SE 8.0 or above. In terms of language bindings (or connectors), Spring Redis integrates with Jedis and Lettuce, two popular open source Java libraries for Redis.

## 5.2. Redis Support High Level View

The Redis support provides several components (in order of dependencies):

For most tasks, the high-level abstractions and support services are the best choice. Note that at any point, one can move between layers - for example, it's very easy to get a hold of the low level connection (or even the native library) to communicate directly with Redis.

## 5.3. Connecting to Redis

One of the first tasks when using Redis and Spring is to connect to the store through the IoC container. To do that, a Java connector (or binding) is required. No matter the library one chooses, there is only one set of Spring Data Redis API that one needs to use that behaves consistently across all connectors, namely the `org.springframework.data.redis.connection` package and its `RedisConnection` and `RedisConnectionFactory` interfaces for working with and retrieving active `connections` to Redis.

### 5.3.1. RedisConnection and RedisConnectionFactory

`RedisConnection` provides the building block for Redis communication as it handles the communication with the Redis back-end. It also automatically translates the underlying connecting library exceptions to Spring's consistent DAO exception hierarchy so one can switch the connectors without any code changes as the operation semantics remain the same.

| | |
|---|---|
| **NOTE** | For the corner cases where the native library API is required, `RedisConnection` provides a dedicated method `getNativeConnection` which returns the raw, underlying object used for communication. |

Active `RedisConnection` s are created through `RedisConnectionFactory`. In addition, the factories act as `PersistenceExceptionTranslator` s, meaning once declared, they allow one to do transparent exception translation. For example, exception translation through the use of the `@Repository` annotation and AOP. For more information see the dedicated section in Spring Framework documentation.

| | |
|---|---|
| **NOTE** | Depending on the underlying configuration, the factory can return a new connection or an existing connection (in case a pool or shared native connection is used). |

The easiest way to work with a `RedisConnectionFactory` is to configure the appropriate connector through the IoC container and inject it into the using class.

| | |
|---|---|
| **IMPORTANT** | Unfortunately, currently, not all connectors support all Redis features. When invoking a method on the Connection API that is unsupported by the underlying library, an `UnsupportedOperationException` is thrown. |

### 5.3.2. Configuring Lettuce connector

Lettuce is a netty-based open-source connector supported by Spring Data Redis through the `org.springframework.data.redis.connection.lettuce` package.

```
@Configuration
class AppConfig {

  @Bean
  public LettuceConnectionFactory redisConnectionFactory() {

    return new LettuceConnectionFactory(new RedisStandaloneConfiguration("server",
6379));
  }
}
```

There are also a few Lettuce-specific connection parameters that can be tweaked. By default, all `LettuceConnection` s created by the `LettuceConnectionFactory` share the same thread-safe native connection for all non-blocking and non-transactional operations. Set `shareNativeConnection` to false to use a dedicated connection each time. `LettuceConnectionFactory` can also be configured with a `LettucePool` to use for pooling blocking and transactional connections, or all connections if `shareNativeConnection` is set to false.

Lettuce integrates with netty's native transports allowing to use unix domain sockets to communicate with Redis. Make sure to include the appropriate native transport dependencies that match your runtime environment.

```
@Configuration
class AppConfig {

  @Bean
  public LettuceConnectionFactory redisConnectionFactory() {

    return new LettuceConnectionFactory(new RedisSocketConfiguration(
"/var/run/redis.sock"));
  }
}
```

| NOTE | Netty currently supports epoll (Linux) and kqueue (BSD/macOS) interfaces for OS-native transport. |
|---|---|

### 5.3.3. Configuring Jedis connector

Jedis is a community-driven connector supported by the Spring Data Redis module through the `org.springframework.data.redis.connection.jedis` package. In its simplest form, the Jedis configuration looks as follow:

```
@Configuration
class AppConfig {

  @Bean
  public JedisConnectionFactory redisConnectionFactory() {
    return new JedisConnectionFactory();
  }
}
```

For production use however, one might want to tweak the settings such as the host or password:

```
@Configuration
class RedisConfiguration {

  @Bean
  public JedisConnectionFactory redisConnectionFactory() {

    RedisStandaloneConfiguration config = new RedisStandaloneConfiguration("server",
6379);
    return new JedisConnectionFactory(config);
  }
}
```

### 5.3.4. Write to Master read from Slave

Redis Master/Slave setup, without automatic failover (for automatic failover see: Sentinel), not only

allows data to be savely stored at more nodes. It also allows, using Lettuce, reading data from slaves while pushing writes to the master. Set the read/write strategy to be used via `LettuceClientConfiguration`.

```
@Configuration
class WriteToMasterReadFromSlaveConfiguration {

  @Bean
  public LettuceConnectionFactory redisConnectionFactory() {

    LettuceClientConfiguration clientConfig = LettuceClientConfiguration.builder()
      .readFrom(SLAVE_PREFERRED)
      .build();

    RedisStandaloneConfiguration serverConfig = new RedisStandaloneConfiguration(
"server", 6379);

    return new LettuceConnectionFactory(serverConfig, clientConfig);
  }
}
```

## 5.4. Redis Sentinel Support

For dealing with high available Redis there is support for Redis Sentinel using `RedisSentinelConfiguration`.

```
/**
 * Jedis
 */
@Bean
public RedisConnectionFactory jedisConnectionFactory() {
  RedisSentinelConfiguration sentinelConfig = new RedisSentinelConfiguration()
  .master("mymaster")
  .sentinel("127.0.0.1", 26379)
  .sentinel("127.0.0.1", 26380);
  return new JedisConnectionFactory(sentinelConfig);
}

/**
 * Lettuce
 */
@Bean
public RedisConnectionFactory lettuceConnectionFactory() {
  RedisSentinelConfiguration sentinelConfig = new RedisSentinelConfiguration()
  .master("mymaster")
  .sentinel("127.0.0.1", 26379)
  .sentinel("127.0.0.1", 26380);
  return new LettuceConnectionFactory(sentinelConfig);
}
```

**TIP**

`RedisSentinelConfiguration` can also be defined via `PropertySource`.

*Configuration Properties*

- `spring.redis.sentinel.master`: name of the master node.
- `spring.redis.sentinel.nodes`: Comma delimited list of host:port pairs.

Sometimes direct interaction with the one of the Sentinels is required. Using `RedisConnectionFactory.getSentinelConnection()` or `RedisConnection.getSentinelCommands()` gives you access to the first active Sentinel configured.

## 5.5. Working with Objects through RedisTemplate

Most users are likely to use `RedisTemplate` and its corresponding package `org.springframework.data.redis.core` - the template is in fact the central class of the Redis module due to its rich feature set. The template offers a high-level abstraction for Redis interactions. While `RedisConnection` offers low level methods that accept and return binary values (`byte` arrays), the template takes care of serialization and connection management, freeing the user from dealing with such details.

Moreover, the template provides operations views (following the grouping from Redis command reference) that offer rich, generified interfaces for working against a certain type or certain key (through the `KeyBound` interfaces) as described below:

*Table 1. Operational views*

| Interface | Description |
|---|---|
| *Key Type Operations* | |
| GeoOperations | Redis geospatial operations like `GEOADD`, `GEORADIUS`,…) |
| HashOperations | Redis hash operations |
| HyperLogLogOperations | Redis HyperLogLog operations like (`PFADD`, `PFCOUNT`,…) |
| ListOperations | Redis list operations |
| SetOperations | Redis set operations |
| ValueOperations | Redis string (or value) operations |
| ZSetOperations | Redis zset (or sorted set) operations |
| *Key Bound Operations* | |
| BoundGeoOperations | Redis key bound geospatial operations. |
| BoundHashOperations | Redis hash key bound operations |
| BoundKeyOperations | Redis key bound operations |
| BoundListOperations | Redis list key bound operations |
| BoundSetOperations | Redis set key bound operations |
| BoundValueOperations | Redis string (or value) key bound operations |
| BoundZSetOperations | Redis zset (or sorted set) key bound operations |

Once configured, the template is thread-safe and can be reused across multiple instances.

Out of the box, `RedisTemplate` uses a Java-based serializer for most of its operations. This means that any object written or read by the template will be serialized/deserialized through Java. The serialization mechanism can be easily changed on the template, and the Redis module offers several implementations available in the `org.springframework.data.redis.serializer` package - see Serializers for more information. You can also set any of the serializers to null and use RedisTemplate with raw `byte` arrays by setting the `enableDefaultSerializer` property to false. Note that the template requires all keys to be non-null - values can be null as long as the underlying serializer accepts them; read the javadoc of each serializer for more information.

For cases where a certain template **view** is needed, declare the view as a dependency and inject the template: the container will automatically perform the conversion eliminating the `opsFor[X]` calls:

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

  <bean id="jedisConnectionFactory" class=
"org.springframework.data.redis.connection.jedis.JedisConnectionFactory" p:use-pool=
"true"/>
  <!-- redis template definition -->
  <bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate"
p:connection-factory-ref="jedisConnectionFactory"/>
  ...

</beans>
```

```
public class Example {

  // inject the actual template
  @Autowired
  private RedisTemplate<String, String> template;

  // inject the template as ListOperations
  @Resource(name="redisTemplate")
  private ListOperations<String, String> listOps;

  public void addLink(String userId, URL url) {
    listOps.leftPush(userId, url.toExternalForm());
  }
}
```

## 5.6. String-focused convenience classes

Since it's quite common for the keys and values stored in Redis to be `java.lang.String`, the Redis modules provides two extensions to `RedisConnection` and `RedisTemplate`, respectively the `StringRedisConnection` (and its `DefaultStringRedisConnection` implementation) and `StringRedisTemplate` as a convenient one-stop solution for intensive String operations. In addition to being bound to `String` keys, the template and the connection use the `StringRedisSerializer` underneath which means the stored keys and values are human readable (assuming the same encoding is used both in Redis and your code). For example:

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

  <bean id="jedisConnectionFactory" class=
"org.springframework.data.redis.connection.jedis.JedisConnectionFactory" p:use-pool=
"true"/>

  <bean id="stringRedisTemplate" class=
"org.springframework.data.redis.core.StringRedisTemplate" p:connection-factory-ref=
"jedisConnectionFactory"/>
  ...
</beans>
```

```
public class Example {

  @Autowired
  private StringRedisTemplate redisTemplate;

  public void addLink(String userId, URL url) {
    redisTemplate.opsForList().leftPush(userId, url.toExternalForm());
  }
}
```

As with the other Spring templates, `RedisTemplate` and `StringRedisTemplate` allow the developer to talk directly to Redis through the `RedisCallback` interface. This gives complete control to the developer as it talks directly to the `RedisConnection`. Note that the callback receives an instance of `StringRedisConnection` when a `StringRedisTemplate` is used.

```
public void useCallback() {

  redisTemplate.execute(new RedisCallback<Object>() {
    public Object doInRedis(RedisConnection connection) throws DataAccessException {
      Long size = connection.dbSize();
      // Can cast to StringRedisConnection if using a StringRedisTemplate
      ((StringRedisConnection)connection).set("key", "value");
    }
   });
}
```

## 5.7. Serializers

From the framework perspective, the data stored in Redis is just bytes. While Redis itself supports

various types, for the most part these refer to the way the data is stored rather than what it represents. It is up to the user to decide whether the information gets translated into Strings or any other objects.

The conversion between the user (custom) types and raw data (and vice-versa) is handled in Spring Data Redis in the `org.springframework.data.redis.serializer` package.

This package contains two types of serializers which as the name implies, takes care of the serialization process:

- Two-way serializers based on `RedisSerializer`.
- Element readers and writers using `RedisElementReader` and `RedisElementWriter`.

The main difference between these variants is that `RedisSerializer` primarily serializes to `byte[]` while readers and writers use `ByteBuffer`.

Multiple implementations are available out of the box, two of which have been already mentioned before in this documentation:

- the `StringRedisSerializer`
- `JdkSerializationRedisSerializer`

However one can use `OxmSerializer` for Object/XML mapping through Spring OXM support or either `Jackson2JsonRedisSerializer` or `GenericJackson2JsonRedisSerializer` for storing data in JSON format.

Do note that the storage format is not limited only to values - it can be used for keys, values or hashes without any restrictions.

# 5.8. Hash mapping

Data can be stored using various data structures within Redis. You already learned about `Jackson2JsonRedisSerializer` which can convert objects in JSON format. JSON can be ideally stored as value using plain keys. A more sophisticated mapping of structured objects can be achieved using Redis Hashes. Spring Data Redis offers various strategies for mapping data to hashes depending on the use case.

1. Direct mapping using `HashOperations` and a serializer
2. Using Redis Repositories
3. Using `HashMapper` and `HashOperations`

## 5.8.1. Hash mappers

Hash mappers are converters to map objects to a `Map<K, V>` and back. `HashMapper` is intended for using with Redis Hashes.

Multiple implementations are available out of the box:

1. `BeanUtilsHashMapper` using Spring's BeanUtils.

2. `ObjectHashMapper` using Object to Hash Mapping.
3. `Jackson2HashMapper` using FasterXML Jackson.

```
public class Person {
  String firstname;
  String lastname;

  // ...
}

public class HashMapping {

  @Autowired
  HashOperations<String, byte[], byte[]> hashOperations;

  HashMapper<Object, byte[], byte[]> mapper = new ObjectHashMapper();

  public void writeHash(String key, Person person) {

    Map<byte[], byte[]> mappedHash = mapper.toHash(person);
    hashOperations.putAll(key, mappedHash);
  }

  public Person loadHash(String key) {

    Map<byte[], byte[]> loadedHash = hashOperations.entries("key");
    return (Person) mapper.fromHash(loadedHash);
  }
}
```

### 5.8.2. Jackson2HashMapper

`Jackson2HashMapper` provides Redis Hash mapping for domain objects using FasterXML Jackson. `Jackson2HashMapper` can map data map top-level properties as Hash field names and optionally flatten the structure. Simple types map to simple values. Complex types (nested objects, collections, maps) are represented as nested JSON.

Flattening creates individual hash entries for all nested properties and resolves complex types into simple types, as far as possible.

```
public class Person {
  String firstname;
  String lastname;
  Address address;
}

public class Address {
  String city;
  String country;
}
```

*Table 2. Normal Mapping*

| Hash Field | Value |
|---|---|
| firstname | `Jon` |
| lastname | `Snow` |
| address | `{ "city" : "Castle Black", "country" : "The North" }` |

*Table 3. Flat Mapping*

| Hash Field | Value |
|---|---|
| firstname | `Jon` |
| lastname | `Snow` |
| address.city | `Castle Black` |
| address.country | `The North` |

**NOTE** Flattening requires all property names to not interfere with the JSON path. Using dots or brackets in map keys or as property names is not supported using flattening. The resulting hash cannot be mapped back into an Object.

## 5.9. Redis Messaging/PubSub

Spring Data provides dedicated messaging integration for Redis, very similar in functionality and naming to the JMS integration in Spring Framework; in fact, users familiar with the JMS support in Spring should feel right at home.

Redis messaging can be roughly divided into two areas of functionality, namely the production or publication and consumption or subscription of messages, hence the shortcut pubsub (Publish/Subscribe). The `RedisTemplate` class is used for message production. For asynchronous reception similar to Java EE's message-driven bean style, Spring Data provides a dedicated message listener container that is used to create Message-Driven POJOs (MDPs) and for synchronous reception, the `RedisConnection` contract.

The package `org.springframework.data.redis.connection` and `org.springframework.data.redis.listener` provide the core functionality for using Redis messaging.

### 5.9.1. Sending/Publishing messages

To publish a message, one can use, as with the other operations, either the low-level `RedisConnection` or the high-level `RedisTemplate`. Both entities offer the `publish` method that accepts as an argument the message that needs to be sent as well as the destination channel. While `RedisConnection` requires raw-data (array of bytes), the `RedisTemplate` allow arbitrary objects to be passed in as messages:

```
// send message through connection RedisConnection con = ...
byte[] msg = ...
byte[] channel = ...
con.publish(msg, channel); // send message through RedisTemplate
RedisTemplate template = ...
template.convertAndSend("hello!", "world");
```

### 5.9.2. Receiving/Subscribing for messages

On the receiving side, one can subscribe to one or multiple channels either by naming them directly or by using pattern matching. The latter approach is quite useful as it not only allows multiple subscriptions to be created with one command but to also listen on channels not yet created at subscription time (as long as they match the pattern).

At the low-level, `RedisConnection` offers `subscribe` and `pSubscribe` methods that map the Redis commands for subscribing by channel respectively by pattern. Note that multiple channels or patterns can be used as arguments. To change the subscription of a connection or simply query whether it is listening or not, `RedisConnection` provides `getSubscription` and `isSubscribed` method.

| | |
|---|---|
| **NOTE** | Subscription commands in Spring Data Redis are blocking. That is, calling subscribe on a connection will cause the current thread to block as it will start waiting for messages - the thread will be released only if the subscription is canceled, that is an additional thread invokes `unsubscribe` or `pUnsubscribe` on the **same** connection. See message listener container below for a solution to this problem. |

As mentioned above, once subscribed a connection starts waiting for messages. No other commands can be invoked on it except for adding new subscriptions or modifying/canceling the existing ones. That is, invoking anything other then `subscribe`, `pSubscribe`, `unsubscribe`, or `pUnsubscribe` is illegal and will throw an exception.

In order to subscribe for messages, one needs to implement the `MessageListener` callback: each time a new message arrives, the callback gets invoked and the user code executed through `onMessage` method. The interface gives access not only to the actual message but to the channel it has been received through and the pattern (if any) used by the subscription to match the channel. This information allows the callee to differentiate between various messages not just by content but also through data.

#### Message Listener Containers

Due to its blocking nature, low-level subscription is not attractive as it requires connection and

thread management for every single listener. To alleviate this problem, Spring Data offers `RedisMessageListenerContainer` which does all the heavy lifting on behalf of the user - users familiar with EJB and JMS should find the concepts familiar as it is designed as close as possible to the support in Spring Framework and its message-driven POJOs (MDPs)

`RedisMessageListenerContainer` acts as a message listener container; it is used to receive messages from a Redis channel and drive the `MessageListener` s that are injected into it. The listener container is responsible for all threading of message reception and dispatches into the listener for processing. A message listener container is the intermediary between an MDP and a messaging provider, and takes care of registering to receive messages, resource acquisition and release, exception conversion and the like. This allows you as an application developer to write the (possibly complex) business logic associated with receiving a message (and reacting to it), and delegates boilerplate Redis infrastructure concerns to the framework.

Furthermore, to minimize the application footprint, `RedisMessageListenerContainer` allows one connection and one thread to be shared by multiple listeners even though they do not share a subscription. Thus no matter how many listeners or channels an application tracks, the runtime cost will remain the same through out its lifetime. Moreover, the container allows runtime configuration changes so one can add or remove listeners while an application is running without the need for restart. Additionally, the container uses a lazy subscription approach, using a `RedisConnection` only when needed - if all the listeners are unsubscribed, cleanup is automatically performed and the used thread released.

To help with the asynch manner of messages, the container requires a `java.util.concurrent.Executor` ( or Spring's `TaskExecutor`) for dispatching the messages. Depending on the load, the number of listeners or the runtime environment, one should change or tweak the executor to better serve her needs - in particular in managed environments (such as app servers), it is highly recommended to pick a a proper `TaskExecutor` to take advantage of its runtime.

**The MessageListenerAdapter**

The `MessageListenerAdapter` class is the final component in Spring's asynchronous messaging support: in a nutshell, it allows you to expose almost **any** class as a MDP (there are of course some constraints).

Consider the following interface definition. Notice that although the interface doesn't extend the `MessageListener` interface, it can still be used as a MDP via the use of the `MessageListenerAdapter` class. Notice also how the various message handling methods are strongly typed according to the **contents** of the various `Message` types that they can receive and handle. In addition, the channel or pattern to which a message is sent can be passed in to the method as the second argument of type String:

```
public interface MessageDelegate {
  void handleMessage(String message);
  void handleMessage(Map message); void handleMessage(byte[] message);
  void handleMessage(Serializable message);
  // pass the channel/pattern as well
  void handleMessage(Serializable message, String channel);
 }
```

```
public class DefaultMessageDelegate implements MessageDelegate {
  // implementation elided for clarity...
}
```

In particular, note how the above implementation of the `MessageDelegate` interface (the above `DefaultMessageDelegate` class) has **no** Redis dependencies at all. It truly is a POJO that we will make into an MDP via the following configuration.

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

<!-- the default ConnectionFactory -->
<redis:listener-container>
  <!-- the method attribute can be skipped as the default method name is
"handleMessage" -->
  <redis:listener ref="listener" method="handleMessage" topic="chatroom" />
</redis:listener-container>

<bean id="listener" class="redisexample.DefaultMessageDelegate"/>
 ...
<beans>
```

| NOTE | The listener topic can be either a channel (e.g. `topic="chatroom"`) or a pattern (e.g. `topic="*room"`) |
|---|---|

The example above uses the Redis namespace to declare the message listener container and automatically register the POJOs as listeners. The full blown, **beans** definition is displayed below:

```
<bean id="messageListener" class=
"org.springframework.data.redis.listener.adapter.MessageListenerAdapter">
  <constructor-arg>
    <bean class="redisexample.DefaultMessageDelegate"/>
  </constructor-arg>
</bean>

<bean id="redisContainer" class=
"org.springframework.data.redis.listener.RedisMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory"/>
  <property name="messageListeners">
    <map>
      <entry key-ref="messageListener">
        <bean class="org.springframework.data.redis.listener.ChannelTopic">
          <constructor-arg value="chatroom">
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

Each time a message is received, the adapter automatically performs translation (using the configured `RedisSerializer`) between the low-level format and the required object type transparently. Any exception caused by the method invocation is caught and handled by the container (by default, being logged).

## 5.10. Redis Transactions

Redis provides support for transactions through the `multi`, `exec`, and `discard` commands. These operations are available on `RedisTemplate`, however `RedisTemplate` is not guaranteed to execute all operations in the transaction using the same connection.

Spring Data Redis provides the `SessionCallback` interface for use when multiple operations need to be performed with the same `connection`, as when using Redis transactions. For example:

```
//execute a transaction
List<Object> txResults = redisTemplate.execute(new SessionCallback<List<Object>>() {
  public List<Object> execute(RedisOperations operations) throws DataAccessException {
    operations.multi();
    operations.opsForSet().add("key", "value1");

    // This will contain the results of all ops in the transaction
    return operations.exec();
  }
});
System.out.println("Number of items added to set: " + txResults.get(0));
```

`RedisTemplate` will use its value, hash key, and hash value serializers to deserialize all results of `exec` before returning. There is an additional `exec` method that allows you to pass a custom serializer for transaction results.

**NOTE**

An important change has been made to the `exec` methods of `RedisConnection` and `RedisTemplate` in version 1.1. Previously these methods returned the results of transactions directly from the connectors. This means that the data types often differed from those returned from the methods of `RedisConnection`. For example, `zAdd` returns a boolean indicating that the element has been added to the sorted set. Most connectors return this value as a long and Spring Data Redis performs the conversion. Another common difference is that most connectors return a status reply (usually the String "OK") for operations like `set`. These replies are typically discarded by Spring Data Redis. Prior to 1.1, these conversions were not performed on the results of `exec`. Also, results were not deserialized in `RedisTemplate`, so they often included raw byte arrays. If this change breaks your application, you can set `convertPipelineAndTxResults` to false on your `RedisConnectionFactory` to disable this behavior.

### 5.10.1. @Transactional Support

Transaction Support is disabled by default and has to be explicitly enabled for each `RedisTemplate` in use by setting `setEnableTransactionSupport(true)`. This will force binding the `RedisConnection` in use to the current `Thread` triggering `MULTI`. If the transaction finishes without errors, `EXEC` is called, otherwise `DISCARD`. Once in `MULTI`, `RedisConnection` would queue write operations, all `readonly` operations, such as `KEYS` are piped to a fresh (non thread bound) `RedisConnection`.

```
/** Sample Configuration **/
@Configuration
public class RedisTxContextConfiguration {
  @Bean
  public StringRedisTemplate redisTemplate() {
    StringRedisTemplate template = new StringRedisTemplate(redisConnectionFactory());
    // explicitly enable transaction support
    template.setEnableTransactionSupport(true);
    return template;
  }

  @Bean
  public PlatformTransactionManager transactionManager() throws SQLException {
    return new DataSourceTransactionManager(dataSource());
  }

  @Bean
  public RedisConnectionFactory redisConnectionFactory( // jedis || lettuce);

  @Bean
  public DataSource dataSource() throws SQLException { // ... }
}
```

```
/** Usage Constrainsts **/

// executed on thread bound connection
template.opsForValue().set("foo", "bar");

// read operation executed on a free (not tx-aware)
connection template.keys("*");

// returns null as values set within transaction are not visible
template.opsForValue().get("foo");
```

## 5.11. Pipelining

Redis provides support for pipelining, which involves sending multiple commands to the server without waiting for the replies and then reading the replies in a single step. Pipelining can improve performance when you need to send several commands in a row, such as adding many elements to the same List.

Spring Data Redis provides several `RedisTemplate` methods for executing commands in a pipeline. If you don't care about the results of the pipelined operations, you can use the standard `execute` method, passing `true` for the `pipeline` argument. The `executePipelined` methods will execute the provided `RedisCallback` or `SessionCallback` in a pipeline and return the results. For example:

```
//pop a specified number of items from a queue
List<Object> results = stringRedisTemplate.executePipelined(
  new RedisCallback<Object>() {
    public Object doInRedis(RedisConnection connection) throws DataAccessException {
      StringRedisConnection stringRedisConn = (StringRedisConnection)connection;
      for(int i=0; i< batchSize; i++) {
        stringRedisConn.rPop("myqueue");
      }
    return null;
  }
});
```

The example above executes a bulk right pop of items from a queue in a pipeline. The `results` List contains all of the popped items. `RedisTemplate` uses its value, hash key, and hash value serializers to deserialize all results before returning, so the returned items in the above example will be Strings. There are additional `executePipelined` methods that allow you to pass a custom serializer for pipelined results.

Note that the value returned from the `RedisCallback` is required to be null, as this value is discarded in favor of returning the results of the pipelined commands.

NOTE

An important change has been made to the `closePipeline` method of `RedisConnection` in version 1.1. Previously this method returned the results of pipelined operations directly from the connectors. This means that the data types often differed from those returned by the methods of `RedisConnection`. For example, `zAdd` returns a boolean indicating that the element has been added to the sorted set. Most connectors return this value as a long and Spring Data Redis performs the conversion. Another common difference is that most connectors return a status reply (usually the String "OK") for operations like `set`. These replies are typically discarded by Spring Data Redis. Prior to 1.1, these conversions were not performed on the results of `closePipeline`. If this change breaks your application, you can set `convertPipelineAndTxResults` to false on your `RedisConnectionFactory` to disable this behavior.

## 5.12. Redis Scripting

Redis versions 2.6 and higher provide support for execution of Lua scripts through the eval and evalsha commands. Spring Data Redis provides a high-level abstraction for script execution that handles serialization and automatically makes use of the Redis script cache.

Scripts can be run through the `execute` methods of `RedisTemplate` and `ReactiveRedisTemplate`. Both use a configurable `ScriptExecutor` / `ReactiveScriptExecutor` to run the provided script. By default, the `ScriptExecutor` takes care of serializing the provided keys and arguments and deserializing the script result. This is done via the key and value serializers of the template. There is an additional overload that allows you to pass custom serializers for the script arguments and result.

The default `ScriptExecutor` optimizes performance by retrieving the SHA1 of the script and attempting first to run `evalsha`, falling back to `eval` if the script is not yet present in the Redis script cache.

Here's an example that executes a common "check-and-set" scenario using a Lua script. This is an ideal use case for a Redis script, as it requires that we execute a set of commands atomically and the behavior of one command is influenced by the result of another.

```
@Bean
public RedisScript<Boolean> script() {

  ScriptSource scriptSource = new ResourceScriptSource(new ClassPathResource("META-
INF/scripts/checkandset.lua");
  return RedisScript.of(scriptSource, Boolean.class);
}
```

```
public class Example {

  @Autowired
  RedisScript<Boolean> script;

  public boolean checkAndSet(String expectedValue, String newValue) {
    return redisTemplate.execute(script, singletonList("key"), asList(expectedValue,
newValue));
  }
}
```

```
-- checkandset.lua local
current = redis.call('GET', KEYS[1])
if current == ARGV[1]
  then redis.call('SET', KEYS[1], ARGV[2])
  return true
end
return false
```

The code above configures a `RedisScript` pointing to a file called `checkandset.lua`, which is expected to return a boolean value. The script `resultType` should be one of `Long`, `Boolean`, `List`, or deserialized value type. It can also be `null` if the script returns a throw-away status (i.e "OK"). It is ideal to configure a single instance of `DefaultRedisScript` in your application context to avoid re-calculation of the script's SHA1 on every script execution.

The checkAndSet method above then executes th Scripts can be executed within a `SessionCallback` as part of a transaction or pipeline. See Redis Transactions and Pipelining for more information.

The scripting support provided by Spring Data Redis also allows you to schedule Redis scripts for periodic execution using the Spring Task and Scheduler abstractions. See the `Spring Framework` documentation for more details.

## 5.13. Support Classes

Package `org.springframework.data.redis.support` offers various reusable components that rely on Redis as a backing store. Currently the package contains various JDK-based interface implementations on top of Redis such as atomic counters and JDK Collections.

The atomic counters make it easy to wrap Redis key incrementation while the collections allow easy management of Redis keys with minimal storage exposure or API leakage: in particular the `RedisSet` and `RedisZSet` interfaces offer easy access to the **set** operations supported by Redis such as `intersection` and `union` while `RedisList` implements the `List`, `Queue` and `Deque` contracts (and their equivalent blocking siblings) on top of Redis, exposing the storage as a *FIFO (First-In-First-Out)*, *LIFO (Last-In-First-Out)* or *capped collection* with minimal configuration:

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

  <bean id="queue" class=
"org.springframework.data.redis.support.collections.DefaultRedisList">
    <constructor-arg ref="redisTemplate"/>
    <constructor-arg value="queue-key"/>
  </bean>

</beans>
```

```
public class AnotherExample {

  // injected
  private Deque<String> queue;

  public void addTag(String tag) {
    queue.push(tag);
  }
}
```

As shown in the example above, the consuming code is decoupled from the actual storage implementation - in fact there is no indication that Redis is used underneath. This makes moving from development to production environments transparent and highly increases testability (the Redis implementation can just as well be replaced with an in-memory one).

### 5.13.1. Support for Spring Cache Abstraction

| **NOTE** | Changed in 2.0 |
|---|---|

Spring Redis provides an implementation for Spring cache abstraction through the `org.springframework.data.redis.cache` package. To use Redis as a backing implementation, simply add `RedisCacheManager` to your configuration:

```
@Bean
public RedisCacheManager cacheManager(RedisConnectionFactory connectionFactory) {
    return RedisCacheManager.create(connectionFactory);
}
```

`RedisCacheManager` behavior can be configured via `RedisCacheManagerBuilder` allowing to set the default `RedisCacheConfiguration`, transaction behaviour and predefined caches.

```
RedisCacheManager cm = RedisCacheManager.builder(connectionFactory)
    .cacheDefaults(defaultCacheConfig())
    .initialCacheConfigurations(singletonMap("predefined", defaultCacheConfig()
.disableCachingNullValues()))
    .transactionAware()
    .build();
```

Behavior of `RedisCache` created via `RedisCacheManager` is defined via `RedisCacheConfiguration`. The configuration allows to set key expiration times, prefixes and `RedisSerializer`s for converting to and from the binary storage format. As shown above `RedisCacheManager` allows definition of configurations on a per cache base.

```
RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig()
    .entryTtl(Duration.ofSeconds(1))
    .disableCachingNullValues();
```

`RedisCacheManager` defaults to a lock-free `RedisCacheWriter` for reading & writing binary values. Lock-free caching improves throughput. The lack of entry locking can lead to overlapping, non atomic commands, for `putIfAbsent` and `clean` methods as those require multiple commands sent to Redis. The locking counterpart prevents command overlap by setting an explicit lock key and checking against presence of this key, which leads to additional requests and potential command wait times.

It is possible to opt in to the locking behavior as follows:

```
RedisCacheManager cm = RedisCacheManager.build(RedisCacheWriter
.lockingRedisCacheWriter())
    .cacheDefaults(defaultCacheConfig())
    ...
```

*Table 4. RedisCacheManager defaults*

| Setting | Value |
|---|---|
| Cache Writer | non locking |
| Cache Configuration | `RedisCacheConfiguration#defaultConfiguration` |
| Initial Caches | none |
| Trasaction Aware | no |

*Table 5. RedisCacheConfiguration defaults*

| Key Expiration | none |
|---|---|
| Cache `null` | yes |
| Prefix Keys | yes |
| Default Prefix | the actual cache name |
| Key Serializer | `StringRedisSerializer` |

| **Key Expiration** | **none** |
| --- | --- |
| Value Serializer | `JdkSerializationRedisSerializer` |
| Conversion Service | `DefaultFormattingConversionService` with default cache key converters |

# Chapter 6. Reactive Redis support

This section covers reactive Redis support and how to get started. You will find certain overlaps with the imperative Redis support.

## 6.1. Redis Requirements

Spring Data Redis requires Redis 2.6 or above and Java SE 8.0 or above. In terms of language bindings (or connectors), Spring Data Redis currently integrates with Lettuce as the only reactive Java connector. Project Reactor is used as reactive composition library.

## 6.2. Connecting to Redis using a reactive driver

One of the first tasks when using Redis and Spring is to connect to the store through the IoC container. To do that, a Java connector (or binding) is required. No matter the library one chooses, there is only one set of Spring Data Redis API that one needs to use that behaves consistently across all connectors, namely the `org.springframework.data.redis.connection` package and its `ReactiveRedisConnection` and `ReactiveRedisConnectionFactory` interfaces for working with and retrieving active `connections` to Redis.

### 6.2.1. Redis Operation Modes

Redis can be run as standalone server, with Redis Sentinel or in Redis Cluster mode. Lettuce supports all above mentioned connection types.

### 6.2.2. ReactiveRedisConnection and ReactiveRedisConnectionFactory

`ReactiveRedisConnection` provides the building block for Redis communication as it handles the communication with the Redis back-end. It also automatically translates the underlying driver exceptions to Spring's consistent DAO exception hierarchy so one can switch the connectors without any code changes as the operation semantics remain the same.

Active `ReactiveRedisConnection`s are created through `ReactiveRedisConnectionFactory`. In addition, the factories act as `PersistenceExceptionTranslator`s, meaning once declared, they allow one to do transparent exception translation. For example, exception translation through the use of the `@Repository` annotation and AOP. For more information see the dedicated section in Spring Framework documentation.

| NOTE | Depending on the underlying configuration, the factory can return a new connection or an existing connection (in case a pool or shared native connection is used). |
|---|---|

The easiest way to work with a `ReactiveRedisConnectionFactory` is to configure the appropriate connector through the IoC container and inject it into the using class.

### 6.2.3. Configuring Lettuce connector

Lettuce is supported by Spring Data Redis through the `org.springframework.data.redis.connection.lettuce` package.

Setting up `ReactiveRedisConnectionFactory` for Lettuce can be done as follows:

```
@Bean
public ReactiveRedisConnectionFactory connectionFactory() {
  return new LettuceConnectionFactory("localhost", 6379);
}
```

A more sophisticated configuration, including SSL and timeouts, using `LettuceClientConfigurationBuilder` might look like below:

```
@Bean
public ReactiveRedisConnectionFactory lettuceConnectionFactory() {

  LettuceClientConfiguration clientConfig = LettuceClientConfiguration.builder()
    .useSsl().and()
    .commandTimeout(Duration.ofSeconds(2))
    .shutdownTimeout(Duration.ZERO)
    .build();

  return new LettuceConnectionFactory(new RedisStandaloneConfiguration("localhost",
6379), clientConfig);
}
```

For more detailed client configuration tweaks have a look at `LettuceClientConfiguration`.

## 6.3. Working with Objects through ReactiveRedisTemplate

Most users are likely to use `ReactiveRedisTemplate` and its corresponding package `org.springframework.data.redis.core` - the template is in fact the central class of the Redis module due to its rich feature set. The template offers a high-level abstraction for Redis interactions. While `ReactiveRedisConnection` offers low level methods that accept and return binary values (`ByteBuffer`), the template takes care of serialization and connection management, freeing the user from dealing with such details.

Moreover, the template provides operation views (following the grouping from Redis command reference) that offer rich, generified interfaces for working against a certain type as described below:

*Table 6. Operational views*

| Interface | Description |
|---|---|
| *Key Type Operations* | |
| ReactiveGeoOperations | Redis geospatial operations like `GEOADD`, `GEORADIUS`,…) |
| ReactiveHashOperations | Redis hash operations |
| ReactiveHyperLogLogOperations | Redis HyperLogLog operations like (`PFADD`, `PFCOUNT`,…) |
| ReactiveListOperations | Redis list operations |
| ReactiveSetOperations | Redis set operations |
| ReactiveValueOperations | Redis string (or value) operations |
| ReactiveZSetOperations | Redis zset (or sorted set) operations |

Once configured, the template is thread-safe and can be reused across multiple instances.

Out of the box, `ReactiveRedisTemplate` uses a Java-based serializer for most of its operations. This means that any object written or read by the template will be serialized/deserialized through `RedisElementWriter` respective `RedisElementReader`. The serialization context is passed to the template upon construction, and the Redis module offers several implementations available in the `org.springframework.data.redis.serializer` package - see Serializers for more information.

```
@Configuration
class RedisConfiguration {

  @Bean
  ReactiveRedisTemplate<String, String> reactiveRedisTemplate
(ReactiveRedisConnectionFactory factory) {
    return new ReactiveRedisTemplate<>(factory, RedisSerializationContext.string());
  }
}
```

```
public class Example {

  @Autowired
  private ReactiveRedisTemplate<String, String> template;

  public Mono<Long> addLink(String userId, URL url) {
    return template.opsForList().leftPush(userId, url.toExternalForm());
  }
}
```

## 6.4. String-focused convenience classes

Since it's quite common for keys and values stored in Redis to be a `java.lang.String`, the Redis module provides a String-based extension to `ReactiveRedisTemplate`: `ReactiveStringRedisTemplate` is a convenient one-stop solution for intensive `String` operations. In addition to being bound to `String`

keys, the template uses the String-based `RedisSerializationContext` underneath which means the stored keys and values are human readable (assuming the same encoding is used both in Redis and your code). For example:

```
@Configuration
class RedisConfiguration {

  @Bean
  ReactiveStringRedisTemplate reactiveRedisTemplate(ReactiveRedisConnectionFactory
factory) {
    return new ReactiveStringRedisTemplate<>(factory);
  }
}
```

```
public class Example {

  @Autowired
  private ReactiveStringRedisTemplate redisTemplate;

  public Mono<Long> addLink(String userId, URL url) {
    return redisTemplate.opsForList().leftPush(userId, url.toExternalForm());
  }
}
```

## 6.5. Reactive Scripting

Executing Redis scripts via the reactive infrastructure can be done using the `ReactiveScriptExecutor` accessed best via `ReactiveRedisTemplate`.

```
public class Example {

  @Autowired
  private ReactiveRedisTemplate<String, String> template;

  public Flux<Long> theAnswerToLife() {

    DefaultRedisScript<Long> script = new DefaultRedisScript<>();
    script.setLocation(new ClassPathResource("META-INF/scripts/42.lua"));
    script.setResultType(Long.class);

    return reactiveTemplate.execute(script);
  }
}
```

Please refer to the scripting section for more details on scripting commands.

# Chapter 7. Redis Cluster

Working with Redis Cluster requires a Redis Server version 3.0+ and provides a very own set of features and capabilities. Please refer to the Cluster Tutorial for more information.

## 7.1. Enabling Redis Cluster

Cluster support is based on the very same building blocks as non clustered communication. `RedisClusterConnection` an extension to `RedisConnection` handles the communication with the Redis Cluster and translates errors into the Spring DAO exception hierarchy. `RedisClusterConnection` 's are created via the `RedisConnectionFactory` which has to be set up with the according `RedisClusterConfiguration`.

*Example 1. Sample RedisConnectionFactory Configuration for Redis Cluster*

```
@Component
@ConfigurationProperties(prefix = "spring.redis.cluster")
public class ClusterConfigurationProperties {

    /*
     * spring.redis.cluster.nodes[0] = 127.0.0.1:7379
     * spring.redis.cluster.nodes[1] = 127.0.0.1:7380
     * ...
     */
    List<String> nodes;

    /**
     * Get initial collection of known cluster nodes in format {@code host:port}.
     *
     * @return
     */
    public List<String> getNodes() {
        return nodes;
    }

    public void setNodes(List<String> nodes) {
        this.nodes = nodes;
    }
}

@Configuration
public class AppConfig {

    /**
     * Type safe representation of application.properties
     */
    @Autowired ClusterConfigurationProperties clusterProperties;

    public @Bean RedisConnectionFactory connectionFactory() {

        return new JedisConnectionFactory(
            new RedisClusterConfiguration(clusterProperties.getNodes()));
    }
}
```

**TIP**

`RedisClusterConfiguration` can also be defined via `PropertySource`.

*Configuration Properties*

- `spring.redis.cluster.nodes`: Comma delimited list of host:port pairs.
- `spring.redis.cluster.max-redirects`: Number of allowed cluster redirections.

| NOTE | The initial configuration points driver libraries to an initial set of cluster nodes. Changes resulting from live cluster reconfiguration will only be kept in the native driver and not be written back to the configuration. |
|---|---|

## 7.2. Working With Redis Cluster Connection

As mentioned above Redis Cluster behaves different from single node Redis or even a Sentinel monitored master slave environment. This is reasoned by the automatic sharding that maps a key to one of 16384 slots which are distributed across the nodes. Therefore commands that involve more than one key must assert that all keys map to the exact same slot in order to avoid cross slot execution errors. Further on, hence a single cluster node, only serves a dedicated set of keys, commands issued against one particular server only return results for those keys served by the server. As a very simple example take the `KEYS` command. When issued to a server in cluster environment it only returns the keys served by the node the request is sent to and not necessarily all keys within the cluster. So to get all keys in cluster environment it is necessary to read the keys from at least all known master nodes.

While redirects for to a specific keys to the corresponding slot serving node are handled by the driver libraries, higher level functions like collecting information across nodes, or sending commands to all nodes in the cluster that are covered by `RedisClusterConnection`. Picking up the keys example from just before, this means, that the `keys(pattern)` method picks up every master node in cluster and simultaneously executes the `KEYS` command on every single one, while picking up the results and returning the cumulated set of keys. To just request the keys of a single node `RedisClusterConnection` provides overloads for those (like `keys(node, pattern)` ).

A `RedisClusterNode` can be obtained from `RedisClusterConnection.clusterGetNodes` or it can be constructed using either host and port or the node Id.

*Example 2. Sample of Running Commands Across the Cluster*

```
redis-cli@127.0.0.1:7379 > cluster nodes

6b38bb... 127.0.0.1:7379 master - 0 0 25 connected 0-5460                    ①
7bb78c... 127.0.0.1:7380 master - 0 1449730618304 2 connected 5461-10922     ②
164888... 127.0.0.1:7381 master - 0 1449730618304 3 connected 10923-16383    ③
b8b5ee... 127.0.0.1:7382 slave 6b38bb... 0 1449730618304 25 connected        ④
```

```
RedisClusterConnection connection = connectionFactory.getClusterConnnection();

connection.set("foo", value);                                                ⑤
connection.set("bar", value);                                                ⑥

connection.keys("*");                                                        ⑦

connection.keys(NODE_7379, "*");                                             ⑧
connection.keys(NODE_7380, "*");                                             ⑨
connection.keys(NODE_7381, "*");                                             ⑩
connection.keys(NODE_7382, "*");                                             ⑪
```

① Master node serving slots 0 to 5460 replicated to slave at 7382

② Master node serving slots 5461 to 10922

③ Master node serving slots 10923 to 16383

④ Slave node holding replicates of master at 7379

⑤ Request routed to node at 7381 serving slot 12182

⑥ Request routed to node at 7379 serving slot 5061

⑦ Request routed to nodes at 7379, 7380, 7381 -> [foo, bar]

⑧ Request routed to node at 7379 -> [bar]

⑨ Request routed to node at 7380 -> []

⑩ Request routed to node at 7381 -> [foo]

⑪ Request routed to node at 7382 -> [bar]

Cross slot requests such as `MGET` are automatically served by the native driver library when all keys map to the same slot. However once this is not the case `RedisClusterConnection` executes multiple parallel `GET` commands against the slot serving nodes and again returns a cumulated result. Obviously this is less performing than the single slot execution and therefore should be used with care. In doubt please consider pinning keys to the same slot by providing a prefix in curly brackets like `{my-prefix}.foo` and `{my-prefix}.bar` which will both map to the same slot number.

*Example 3. Sample of Cross Slot Request Handling*

```
redis-cli@127.0.0.1:7379 > cluster nodes

6b38bb... 127.0.0.1:7379 master - 0 0 25 connected 0-5460                    ①
7bb...
```

```
RedisClusterConnection connection = connectionFactory.getClusterConnnection();

connection.set("foo", value);         // slot: 12182
connection.set("{foo}.bar", value);   // slot: 12182
connection.set("bar", value);         // slot:  5461

connection.mGet("foo", "{foo}.bar");                                         ②

connection.mGet("foo", "bar");                                               ③
```

① Same Configuration as in the sample before.

② Keys map to same slot -> 127.0.0.1:7381 MGET foo {foo}.bar

③ Keys map to different slots and get split up into single slot ones routed to the according nodes
→ 127.0.0.1:7379 GET bar
→ 127.0.0.1:7381 GET foo

**TIP** The above provided simple examples to demonstrate the general strategy followed by Spring Data Redis. Be aware that some operations might require loading huge amounts of data into memory in order to compute the desired command. Additionally not all cross slot requests can safely be ported to multiple single slot requests and will error if misused (eg. `PFCOUNT`).

## 7.3. Working With RedisTemplate and ClusterOperations

Please refer to the section Working with Objects through RedisTemplate to read about general purpose, configuration and usage of `RedisTemplate`.

**WARNING** Please be careful when setting up `RedisTemplate#keySerializer` using any of the JSON `RedisSerializers` as changing json structure has immediate influence on hash slot calculation.

`RedisTemplate` provides access to cluster specific operations via the `ClusterOperations` interface that can be obtained via `RedisTemplate.opsForCluster()`. This allows to execute commands explicitly on a single node within the cluster while retaining de-/serialization features configured for the template and provides administrative commands such as `CLUSTER MEET` or more high level operations for eg.

resharding.

*Example 4. Accessing RedisClusterConnection via RedisTemplate*

```
ClusterOperations clusterOps = redisTemplate.opsForCluster();
clusterOps.shutdown(NODE_7379);                                    ①
```

① Shut down node at 7379 and cross fingers there is a slave in place that can take over.

# Chapter 8. Redis Repositories

Working with Redis Repositories allows to seamlessly convert and store domain objects in Redis Hashes, apply custom mapping strategies and make use of secondary indexes.

| **WARNING** | Redis Repositories requires at least Redis Server version 2.8.0. |
|---|---|

## 8.1. Usage

To access domain entities stored in a Redis you can leverage repository support that eases implementing those quite significantly.

*Example 5. Sample Person Entity*

```
@RedisHash("persons")
public class Person {

  @Id String id;
  String firstname;
  String lastname;
  Address address;
}
```

We have a pretty simple domain object here. Note that it has a property named `id` annotated with `org.springframework.data.annotation.Id` and a `@RedisHash` annotation on its type. Those two are responsible for creating the actual key used to persist the hash.

| **NOTE** | Properties annotated with `@Id` as well as those named `id` are considered as the identifier properties. Those with the annotation are favored over others. |
|---|---|

To now actually have a component responsible for storage and retrieval we need to define a repository interface.

*Example 6. Basic Repository Interface To Persist Person Entities*

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

}
```

As our repository extends `CrudRepository` it provides basic CRUD and finder operations. The thing we need in between to glue things together is the according Spring configuration.

*Example 7. JavaConfig for Redis Repositories*

```
@Configuration
@EnableRedisRepositories
public class ApplicationConfig {

  @Bean
  public RedisConnectionFactory connectionFactory() {
    return new JedisConnectionFactory();
  }

  @Bean
  public RedisTemplate<?, ?> redisTemplate() {

    RedisTemplate<byte[], byte[]> template = new RedisTemplate<byte[], byte[]>();
    return template;
  }
}
```

Given the setup above we can go on and inject `PersonRepository` into our components.

*Example 8. Access to Person Entities*

```
@Autowired PersonRepository repo;

public void basicCrudOperations() {

  Person rand = new Person("rand", "al'thor");
  rand.setAddress(new Address("emond's field", "andor"));

  repo.save(rand);                                          ①

  repo.findOne(rand.getId());                               ②

  repo.count();                                             ③

  repo.delete(rand);                                        ④
}
```

① Generates a new id if current value is `null` or reuses an already set id value and stores properties of type `Person` inside the Redis Hash with key with pattern `keyspace:id` in this case eg. `persons:5d67b7e1-8640-4475-beeb-c666fab4c0e5`.

② Uses the provided id to retrieve the object stored at `keyspace:id`.

③ Counts the total number of entities available within the keyspace *persons* defined by `@RedisHash` on `Person`.

④ Removes the key for the given object from Redis.

## 8.2. Object to Hash Mapping

The Redis Repository support persists Objects in Hashes. This requires an Object to Hash conversion which is done by a `RedisConverter`. The default implementation uses `Converter` for mapping property values to and from Redis native `byte[]`.

Given the `Person` type from the previous sections the default mapping looks like the following:

```
_class = org.example.Person                    ①
id = e2c7dcee-b8cd-4424-883e-736ce564363e
firstname = rand                               ②
lastname = al'thor
address.city = emond's field                   ③
address.country = andor
```

① The `_class` attribute is included on root level as well as on any nested interface or abstract types.

② Simple property values are mapped by path.

③ Properties of complex types are mapped by their dot path.

*Table 7. Default Mapping Rules*

| Type | Sample | Mapped Value |
|---|---|---|
| Simple Type (eg. String) | String firstname = "rand"; | firstname = "rand" |
| Complex Type (eg. Address) | Address adress = new Address("emond's field"); | address.city = "emond's field" |
| List of Simple Type | List<String> nicknames = asList("dragon reborn", "lews therin"); | nicknames.[0] = "dragon reborn", nicknames.[1] = "lews therin" |
| Map of Simple Type | Map<String, String> atts = asMap({"eye-color", "grey"}, {"… | atts.[eye-color] = "grey", atts.[hair-color] = "… |
| List of Complex Type | List<Address> addresses = asList(new Address("em… | addresses.[0].city = "emond's field", addresses.[1].city = "… |
| Map of Complex Type | Map<String, Address> addresses = asMap({"home", new Address("em… | addresses.[home].city = "emond's field", addresses.[work].city = "… |

Mapping behavior can be customized by registering the according `Converter` in `RedisCustomConversions`. Those converters can take care of converting from/to a single `byte[]` as well as `Map<String,byte[]>` whereas the first one is suitable for eg. converting one complex type to eg. a binary JSON representation that still uses the default mappings hash structure. The second option offers full control over the resulting hash. Writing objects to a Redis hash will delete the content from the hash and re-create the whole hash, so not mapped data will be lost.

*Example 9. Sample byte[] Converters*

```
@WritingConverter
public class AddressToBytesConverter implements Converter<Address, byte[]> {

  private final Jackson2JsonRedisSerializer<Address> serializer;

  public AddressToBytesConverter() {

    serializer = new Jackson2JsonRedisSerializer<Address>(Address.class);
    serializer.setObjectMapper(new ObjectMapper());
  }

  @Override
  public byte[] convert(Address value) {
    return serializer.serialize(value);
  }
}

@ReadingConverter
public class BytesToAddressConverter implements Converter<byte[], Address> {

  private final Jackson2JsonRedisSerializer<Address> serializer;

  public BytesToAddressConverter() {

    serializer = new Jackson2JsonRedisSerializer<Address>(Address.class);
    serializer.setObjectMapper(new ObjectMapper());
  }

  @Override
  public Address convert(byte[] value) {
    return serializer.deserialize(value);
  }
}
```

Using the above byte[] `Converter` produces eg.

```
_class = org.example.Person
id = e2c7dcee-b8cd-4424-883e-736ce564363e
firstname = rand
lastname = al'thor
address = { city : "emond's field", country : "andor" }
```

*Example 10. Sample Map<String,byte[]> Converters*

```
@WritingConverter
public class AddressToMapConverter implements Converter<Address, Map<String,byte
[]>> {

  @Override
  public Map<String,byte[]> convert(Address source) {
    return singletonMap("ciudad", source.getCity().getBytes());
  }
}

@ReadingConverter
public class MapToAddressConverter implements Converter<Address, Map<String,
byte[]>> {

  @Override
  public Address convert(Map<String,byte[]> source) {
    return new Address(new String(source.get("ciudad")));
  }
}
```

Using the above Map `Converter` produces eg.

```
_class = org.example.Person
id = e2c7dcee-b8cd-4424-883e-736ce564363e
firstname = rand
lastname = al'thor
ciudad = "emond's field"
```

**NOTE** Custom conversions have no effect on index resolution. Secondary Indexes will still be created even for custom converted types.

## 8.3. Keyspaces

Keyspaces define prefixes used to create the actual *key* for the Redis Hash. By default the prefix is set to `getClass().getName()`. This default can be altered via `@RedisHash` on aggregate root level or by setting up a programmatic configuration. However, the annotated keyspace supersedes any other configuration.

*Example 11. Keyspace Setup via @EnableRedisRepositories*

```
@Configuration
@EnableRedisRepositories(keyspaceConfiguration = MyKeyspaceConfiguration.class)
public class ApplicationConfig {

  //... RedisConnectionFactory and RedisTemplate Bean definitions omitted

  public static class MyKeyspaceConfiguration extends KeyspaceConfiguration {

    @Override
    protected Iterable<KeyspaceSettings> initialConfiguration() {
      return Collections.singleton(new KeyspaceSettings(Person.class, "persons"));
    }
  }
}
```

*Example 12. Programmatic Keyspace setup*

```
@Configuration
@EnableRedisRepositories
public class ApplicationConfig {

  //... RedisConnectionFactory and RedisTemplate Bean definitions omitted

  @Bean
  public RedisMappingContext keyValueMappingContext() {
    return new RedisMappingContext(
      new MappingConfiguration(
        new MyKeyspaceConfiguration(), new IndexConfiguration()));
  }

  public static class MyKeyspaceConfiguration extends KeyspaceConfiguration {

    @Override
    protected Iterable<KeyspaceSettings> initialConfiguration() {
      return Collections.singleton(new KeyspaceSettings(Person.class, "persons"));
    }
  }
}
```

## 8.4. Secondary Indexes

Secondary indexes are used to enable lookup operations based on native Redis structures. Values are written to the according indexes on every save and are removed when objects are deleted or expire.

### 8.4.1. Simple Property Index

Given the sample `Person` entity we can create an index for *firstname* by annotating the property with `@Indexed`.

*Example 13. Annotation driven indexing*

```
@RedisHash("persons")
public class Person {

  @Id String id;
  @Indexed String firstname;
  String lastname;
  Address address;
}
```

Indexes are built up for actual property values. Saving two Persons eg. "rand" and "aviendha" results in setting up indexes like below.

```
SADD persons:firstname:rand e2c7dcee-b8cd-4424-883e-736ce564363e
SADD persons:firstname:aviendha a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56
```

It is also possible to have indexes on nested elements. Assume `Address` has a *city* property that is annotated with `@Indexed`. In that case, once `person.address.city` is not `null`, we have Sets for each city.

```
SADD persons:address.city:tear e2c7dcee-b8cd-4424-883e-736ce564363e
```

Further more the programmatic setup allows to define indexes on map keys and list properties.

```
@RedisHash("persons")
public class Person {

  // ... other properties omitted

  Map<String,String> attributes;      ①
  Map<String Person> relatives;       ②
  List<Address> addresses;            ③
}
```

① SADD persons:attributes.map-key:map-value e2c7dcee-b8cd-4424-883e-736ce564363e

② SADD persons:relatives.map-key.firstname:tam e2c7dcee-b8cd-4424-883e-736ce564363e

③ SADD persons:addresses.city:tear e2c7dcee-b8cd-4424-883e-736ce564363e

**WARNING** Indexes will not be resolved on References.

Same as with *keyspaces* it is possible to configure indexes without the need of annotating the actual domain type.

*Example 14. Index Setup via @EnableRedisRepositories*

```
@Configuration
@EnableRedisRepositories(indexConfiguration = MyIndexConfiguration.class)
public class ApplicationConfig {

  //... RedisConnectionFactory and RedisTemplate Bean definitions omitted

  public static class MyIndexConfiguration extends IndexConfiguration {

    @Override
    protected Iterable<IndexDefinition> initialConfiguration() {
      return Collections.singleton(new SimpleIndexDefinition("persons", "
firstname"));
    }
  }
}
```

*Example 15. Programmatic Index setup*

```
@Configuration
@EnableRedisRepositories
public class ApplicationConfig {

  //... RedisConnectionFactory and RedisTemplate Bean definitions omitted

  @Bean
  public RedisMappingContext keyValueMappingContext() {
    return new RedisMappingContext(
      new MappingConfiguration(
        new KeyspaceConfiguration(), new MyIndexConfiguration()));
  }

  public static class MyIndexConfiguration extends IndexConfiguration {

    @Override
    protected Iterable<IndexDefinition> initialConfiguration() {
      return Collections.singleton(new SimpleIndexDefinition("persons", "
firstname"));
    }
  }
}
```

### 8.4.2. Geospatial Index

Assume the `Address` type contains a property `location` of type `Point` that holds the geo coordinates of the particular address. By annotating the property with `@GeoIndexed` those values will be added using Redis `GEO` commands.

```
@RedisHash("persons")
public class Person {

  Address address;

  // ... other properties omitted
}

public class Address {

  @GeoIndexed Point location;

  // ... other properties omitted
}

public interface PersonRepository extends CrudRepository<Person, String> {

  List<Person> findByAddressLocationNear(Point point, Distance distance);     ①
  List<Person> findByAddressLocationWithin(Circle circle);                    ②
}

Person rand = new Person("rand", "al'thor");
rand.setAddress(new Address(new Point(13.361389D, 38.115556D)));

repository.save(rand);                                                        ③

repository.findByAddressLocationNear(new Point(15D, 37D), new Distance(200)); ④
```

① Query method declaration on nested property using Point and Distance.

② Query method declaration on nested property using Circle to search within.

③ `GEOADD persons:address:location 13.361389 38.115556 e2c7dcee-b8cd-4424-883e-736ce564363e`

④ `GEORADIUS persons:address:location 15.0 37.0 200.0 km`

In the above example the lon/lat values are stored using `GEOADD` using the objects `id` as the member's name. The finder methods allow usage of `Circle` or `Point, Distance` combinations for querying those values.

**NOTE** It is **not** possible to combine `near`/`within` with other criteria.

## 8.5. Time To Live

Objects stored in Redis may only be valid for a certain amount of time. This is especially useful for persisting short lived objects in Redis without having to remove them manually when they reached their end of life. The expiration time in seconds can be set via `@RedisHash(timeToLive=…)` as well as via `KeyspaceSettings` (see Keyspaces).

More flexible expiration times can be set by using the `@TimeToLive` annotation on either a numeric property or method. However do not apply `@TimeToLive` on both a method and a property within the same class.

*Example 16. Expirations*

```
public class TimeToLiveOnProperty {

  @Id
  private String id;

  @TimeToLive
  private Long expiration;
}

public class TimeToLiveOnMethod {

  @Id
  private String id;

  @TimeToLive
  public long getTimeToLive() {
    return new Random().nextLong();
  }
}
```

**NOTE** Annotating a property explicitly with `@TimeToLive` will read back the actual `TTL` or `PTTL` value from Redis. -1 indicates that the object has no expire associated.

The repository implementation ensures subscription to Redis keyspace notifications via `RedisMessageListenerContainer`.

When the expiration is set to a positive value the according `EXPIRE` command is executed. Additionally to persisting the original, a *phantom* copy is persisted in Redis and set to expire 5 minutes after the original one. This is done to enable the Repository support to publish `RedisKeyExpiredEvent` holding the expired value via Springs `ApplicationEventPublisher` whenever a key expires even though the original values have already been gone. Expiry events will be received on all connected applications using Spring Data Redis repositories.

By default, the key expiry listener is disabled when initializing the application. The startup mode can be adjusted in `@EnableRedisRepositories` or `RedisKeyValueAdapter` to start the listener with the application or upon the first insert of an entity with a TTL. See `EnableKeyspaceEvents` for possible values.

The `RedisKeyExpiredEvent` will hold a copy of the actually expired domain object as well as the key.

| | |
|---|---|
| **NOTE** | Delaying or disabling the expiry event listener startup impacts `RedisKeyExpiredEvent` publishing. A disabled event listener will not publish expiry events. A delayed startup can cause loss of events because the delayed listener initialization. |
| **NOTE** | The keyspace notification message listener will alter `notify-keyspace-events` settings in Redis if those are not already set. Existing settings will not be overridden, so it is left to the user to set those up correctly when not leaving them empty. Please note that `CONFIG` is disabled on AWS ElastiCache and enabling the listener leads to an error. |
| **NOTE** | Redis Pub/Sub messages are not persistent. If a key expires while the application is down the expiry event will not be processed which may lead to secondary indexes containing still references to the expired object. |

## 8.6. Persisting References

Marking properties with `@Reference` allows storing a simple key reference instead of copying values into the hash itself. On loading from Redis, references are resolved automatically and mapped back into the object.

*Example 17. Sample Property Reference*

```
_class = org.example.Person
id = e2c7dcee-b8cd-4424-883e-736ce564363e
firstname = rand
lastname = al'thor
mother = persons:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56      ①
```

① Reference stores the whole key (`keyspace:id`) of the referenced object.

| | |
|---|---|
| **WARNING** | Referenced Objects are not subject of persisting changes when saving the referencing object. Please make sure to persist changes on referenced objects separately, since only the reference will be stored. Indexes set on properties of referenced types will not be resolved. |

## 8.7. Persisting Partial Updates

In some cases it is not necessary to load and rewrite the entire entity just to set a new value within it. A session timestamp for last active time might be such a scenario where you just want to alter one property. `PartialUpdate` allows to define `set` and `delete` actions on existing objects while taking care of updating potential expiration times of the entity itself as well as index structures.

*Example 18. Sample Partial Update*

```
PartialUpdate<Person> update = new PartialUpdate<Person>("e2c7dcee", Person.class)
  .set("firstname", "mat")
①
  .set("address.city", "emond's field")
②
  .del("age");
③

template.update(update);

update = new PartialUpdate<Person>("e2c7dcee", Person.class)
  .set("address", new Address("caemlyn", "andor"))
④
  .set("attributes", singletonMap("eye-color", "grey"));
⑤

template.update(update);

update = new PartialUpdate<Person>("e2c7dcee", Person.class)
  .refreshTtl(true);
⑥
  .set("expiration", 1000);

template.update(update);
```

① Set the simple property *firstname* to *mat*.

② Set the simple property *address.city* to *emond's field* without having to pass in the entire object. This does not work when a custom conversion is registered.

③ Remove the property *age*.

④ Set complex property *address*.

⑤ Set a map/collection of values removes the previously existing map/collection and replaces the values with the given ones.

⑥ Automatically update the server expiration time when altering Time To Live.

**NOTE** Updating complex objects as well as map/collection structures requires further interaction with Redis to determine existing values which means that it might turn out that rewriting the entire entity might be faster.

## 8.8. Queries and Query Methods

Query methods allow automatic derivation of simple finder queries from the method name.

*Example 19. Sample Repository finder Method*

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

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

| NOTE | Please make sure properties used in finder methods are set up for indexing. |
|---|---|

| NOTE | Query methods for Redis repositories support only queries for entities and collections of entities with paging. |
|---|---|

Using derived query methods might not always be sufficient to model the queries to execute. `RedisCallback` offers more control over the actual matching of index structures or even custom added ones. All it takes is providing a `RedisCallback` that returns a single or `Iterable` set of *id* values.

*Example 20. Sample finder using RedisCallback*

```
String user = //...

List<RedisSession> sessionsByUser = template.find(new RedisCallback<Set<byte[]>>()
{

  public Set<byte[]> doInRedis(RedisConnection connection) throws
DataAccessException {
    return connection
      .sMembers("sessions:securityContext.authentication.principal.username:" +
user);
  }}, RedisSession.class);
```

Here's an overview of the keywords supported for Redis and what a method containing that keyword essentially translates to.

*Table 8. Supported keywords inside method names*

| Keyword | Sample | Redis snippet |
|---|---|---|
| `And` | `findByLastnameAndFirstname` | `SINTER …:firstname:rand …:lastname:al'thor` |
| `Or` | `findByLastnameOrFirstname` | `SUNION …:firstname:rand …:lastname:al'thor` |
| `Is,Equals` | `findByFirstname,findByFirstnameIs,findByFirstnameEquals` | `SINTER …:firstname:rand` |
| `Top,First` | `findFirst10ByFirstname,findTop5ByFirstname` | |

## 8.9. Redis Repositories running on Cluster

Using the Redis repository support in a clustered Redis environment is fine. Please see the Redis Cluster section for `ConnectionFactory` configuration details. Still some considerations have to be done as the default key distribution will spread entities and secondary indexes through out the whole cluster and its slots.

| key | type | slot | node |
|---|---|---|---|
| persons:e2c7dcee-b8cd-4424-883e-736ce564363e | id for hash | 15171 | 127.0.0.1:7381 |
| persons:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56 | id for hash | 7373 | 127.0.0.1:7380 |
| persons:firstname:rand | index | 1700 | 127.0.0.1:7379 |

Some commands like `SINTER` and `SUNION` can only be processed on the Server side when all involved keys map to the same slot. Otherwise computation has to be done on client side. Therefore it be useful to pin keyspaces to a single slot which allows to make use of Redis serverside computation right away.

| key | type | slot | node |
|---|---|---|---|
| {persons}:e2c7dcee-b8cd-4424-883e-736ce564363e | id for hash | 2399 | 127.0.0.1:7379 |
| {persons}:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56 | id for hash | 2399 | 127.0.0.1:7379 |
| {persons}:firstname:rand | index | 2399 | 127.0.0.1:7379 |

**TIP** Define and pin keyspaces via `@RedisHash("{yourkeyspace}") to specific slots when using Redis cluster.

## 8.10. CDI integration

Instances of the repository interfaces are usually created by a container, which Spring is the most natural choice when working with Spring Data. There's sophisticated support to easily set up Spring to create bean instances. Spring Data Redis ships with a custom CDI extension that allows using the

repository abstraction in CDI environments. The extension is part of the JAR so all you need to do to activate it is dropping the Spring Data Redis JAR into your classpath.

You can now set up the infrastructure by implementing a CDI Producer for the `RedisConnectionFactory` and `RedisOperations`:

```
class RedisOperationsProducer {


  @Produces
  RedisConnectionFactory redisConnectionFactory() {

    JedisConnectionFactory jedisConnectionFactory = new JedisConnectionFactory(new
RedisStandaloneConfiguration());
    jedisConnectionFactory.afterPropertiesSet();

    return jedisConnectionFactory;
  }

  void disposeRedisConnectionFactory(@Disposes RedisConnectionFactory
redisConnectionFactory) throws Exception {

    if (redisConnectionFactory instanceof DisposableBean) {
      ((DisposableBean) redisConnectionFactory).destroy();
    }
  }

  @Produces
  @ApplicationScoped
  RedisOperations<byte[], byte[]> redisOperationsProducer(RedisConnectionFactory
redisConnectionFactory) {

    RedisTemplate<byte[], byte[]> template = new RedisTemplate<byte[], byte[]>();
    template.setConnectionFactory(redisConnectionFactory);
    template.afterPropertiesSet();

    return template;
  }

}
```

The necessary setup can vary depending on the JavaEE environment you run in.

The Spring Data Redis CDI extension will pick up all Repositories available as CDI beans and create a proxy for a Spring Data repository whenever a bean of a repository type is requested by the container. Thus obtaining an instance of a Spring Data repository is a matter of declaring an `@Injected` property:

```
class RepositoryClient {

  @Inject
  PersonRepository repository;

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

A Redis Repository requires `RedisKeyValueAdapter` and `RedisKeyValueTemplate` instances. These beans are created and managed by the Spring Data CDI extension if no provided beans are found. You can however supply your own beans to configure the specific properties of `RedisKeyValueAdapter` and `RedisKeyValueTemplate`.

# Appendixes

## Appendix Document structure

Various appendixes outside the reference documentation.

Schema defines the schemas provided by Spring Data Redis.

# Appendix A: Schema

## Core schema

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns="http://www.springframework.org/schema/redis"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tool="http://www.springframework.org/schema/tool"
    targetNamespace="http://www.springframework.org/schema/redis"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

  <xsd:import namespace="http://www.springframework.org/schema/tool" schemaLocation=
"http://www.springframework.org/schema/tool/spring-tool.xsd"/>

  <xsd:annotation>
    <xsd:documentation><![CDATA[
Defines the configuration elements for the Spring Data Redis support.
Allows for configuring Redis listener containers in XML 'shortcut' style.
    ]]></xsd:documentation>
  </xsd:annotation>

  <xsd:element name="listener-container">
    <xsd:annotation>
      <xsd:documentation><![CDATA[
Container of Redis listeners. All listeners will be hosted by the same container.
      ]]></xsd:documentation>
      <xsd:appinfo>
        <tool:annotation>
          <tool:exports type=
"org.springframework.data.redis.listener.RedisMessageListenerContainer"/>
        </tool:annotation>
      </xsd:appinfo>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="listener" type="listenerType" minOccurs="0" maxOccurs=
"unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="connection-factory" type="xsd:string" default=
"redisConnectionFactory">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
A reference to the Redis ConnectionFactory bean.
Default is "redisConnectionFactory".
          ]]></xsd:documentation>
          <xsd:appinfo>
```

```
            <tool:annotation kind="ref">
              <tool:expected-type type=
"org.springframework.data.redis.connection.ConnectionFactory"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="task-executor" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
A reference to a Spring TaskExecutor (or standard JDK 1.5 Executor) for executing
Redis listener invokers. Default is a SimpleAsyncTaskExecutor.
          ]]></xsd:documentation>
          <xsd:appinfo>
            <tool:annotation kind="ref">
              <tool:expected-type type="java.util.concurrent.Executor"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="subscription-task-executor" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
A reference to a Spring TaskExecutor (or standard JDK 1.5 Executor) for listening
to Redis messages. By default reuses the 'task-executor' value.
          ]]></xsd:documentation>
          <xsd:appinfo>
            <tool:annotation kind="ref">
              <tool:expected-type type="java.util.concurrent.Executor"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="topic-serializer" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
A reference to the RedisSerializer strategy for converting Redis channels/patterns to
serialized format. Default is a StringRedisSerializer.
          ]]></xsd:documentation>
          <xsd:appinfo>
            <tool:annotation kind="ref">
              <tool:expected-type type=
"org.springframework.data.redis.serializer.RedisSerializer"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="phase" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
The lifecycle phase within which this container should start and stop. The lower
```

```
the value the earlier this container will start and the later it will stop. The
default is Integer.MAX_VALUE meaning the container will start as late as possible
and stop as soon as possible.
          ]]></xsd:documentation>
        </xsd:annotation>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="listenerType">
    <xsd:attribute name="ref" type="xsd:string" use="required">
      <xsd:annotation>
        <xsd:documentation><![CDATA[
  The bean name of the listener object, implementing
  the MessageListener interface or defining the specified listener method.
  Required.
        ]]></xsd:documentation>
        <xsd:appinfo>
          <tool:annotation kind="ref"/>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="topic" type="xsd:string">
      <xsd:annotation>
        <xsd:documentation><![CDATA[
The topics(s) to which the listener is subscribed. Can be (in Redis terminology) a
channel or/and a pattern. Multiple values can be specified by separating them with
spaces. Patterns can be specified by using the '*' character.
        ]]></xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="method" type="xsd:string">
      <xsd:annotation>
        <xsd:documentation><![CDATA[
The name of the listener method to invoke. If not specified,
the target bean is supposed to implement the MessageListener
interface or provide a method named 'handleMessage'.
        ]]></xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="serializer" type="xsd:string">
      <xsd:annotation>
        <xsd:documentation><![CDATA[
A reference to the RedisSerializer strategy for converting Redis Messages to
listener method arguments. Default is a StringRedisSerializer.
        ]]></xsd:documentation>
        <xsd:appinfo>
          <tool:annotation kind="ref">
            <tool:expected-type type=
"org.springframework.data.redis.serializer.RedisSerializer"/>
          </tool:annotation>
```

```
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:attribute>
  </xsd:complexType>

  <xsd:element name="collection">
    <xsd:annotation>
      <xsd:documentation><![CDATA[
Factory creating collections on top of Redis keys.
      ]]></xsd:documentation>
      <xsd:appinfo>
        <tool:annotation>
          <tool:exports type=
"org.springframework.data.redis.support.collections.RedisCollectionFactoryBean"/>
        </tool:annotation>
      </xsd:appinfo>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:attribute name="id" type="xsd:ID">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
The name of the Redis collection.]]></xsd:documentation>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="key" type="xsd:string" use="optional">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
Redis key of the created collection. Defaults to bean id.
          ]]></xsd:documentation>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="template" type="xsd:string" default="redisTemplate">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
A reference to a RedisTemplate bean.Default is "redisTemplate".
          ]]></xsd:documentation>
          <xsd:appinfo>
            <tool:annotation kind="ref">
              <tool:expected-type type=
"org.springframework.data.redis.core.RedisTemplate"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="type" default="LIST" use="optional">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
The collection type (default is list).
If the key exists, its type takes priority. The type is used to disambiguate the
collection type (map vs properties) or
specify one in case the key is missing.]]></xsd:documentation>
```

```
        </xsd:annotation>
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="LIST"/>
                <xsd:enumeration value="SET"/>
                <xsd:enumeration value="ZSET"/>
                <xsd:enumeration value="MAP"/>
                <xsd:enumeration value="PROPERTIES"/>
            </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

# Appendix B: Command Reference

## Supported commands

*Table 9. Redis commands supported by RedisTemplate.*

| Command | Template Support |
|---|---|
| APPEND | X |
| AUTH | X |
| BGREWRITEAOF | X |
| BGSAVE | X |
| BITCOUNT | X |
| BITOP | X |
| BLPOP | X |
| BRPOP | X |
| BRPOPLPUSH | X |
| CLIENT KILL | X |
| CLIENT GETNAME | X |
| CLIENT LIST | X |
| CLIENT SETNAME | X |
| CLUSTER SLOTS | - |
| COMMAND | - |
| COMMAND COUNT | - |
| COMMAND GETKEYS | - |
| COMMAND INFO | - |
| CONFIG GET | X |
| CONFIG RESETSTAT | X |
| CONFIG REWRITE | - |
| CONFIG SET | X |
| DBSIZE | X |
| DEBUG OBJECT | - |
| DEBUG SEGFAULT | - |
| DECR | X |
| DECRBY | X |
| DEL | X |
| DISCARD | X |
| DUMP | X |

| Command | Template Support |
|---|---|
| ECHO | X |
| EVAL | X |
| EVALSHA | X |
| EXEC | X |
| EXISTS | X |
| EXPIRE | X |
| EXPIREAT | X |
| FLUSHALL | X |
| FLUSHDB | X |
| GET | X |
| GETBIT | X |
| GETRANGE | X |
| GETSET | X |
| HDEL | X |
| HEXISTS | X |
| HGET | X |
| HGETALL | X |
| HINCRBY | X |
| HINCRBYFLOAT | X |
| HKEYS | X |
| HLEN | X |
| HMGET | X |
| HMSET | X |
| HSCAN | X |
| HSET | X |
| HSETNX | X |
| HVALS | X |
| INCR | X |
| INCRBY | X |
| INCRBYFLOAT | X |
| INFO | X |
| KEYS | X |
| LASTSAVE | X |
| LINDEX | X |
| LINSERT | X |

| Command | Template Support |
|---|---|
| LLEN | X |
| LPOP | X |
| LPUSH | X |
| LPUSHX | X |
| LRANGE | X |
| LREM | X |
| LSET | X |
| LTRIM | X |
| MGET | X |
| MIGRATE | - |
| MONITOR | - |
| MOVE | X |
| MSET | X |
| MSETNX | X |
| MULTI | X |
| OBJECT | - |
| PERSIST | X |
| PEXIPRE | X |
| PEXPIREAT | X |
| PFADD | X |
| PFCOUNT | X |
| PFMERGE | X |
| PING | X |
| PSETEX | X |
| PSUBSCRIBE | X |
| PTTL | X |
| PUBLISH | X |
| PUBSUB | - |
| PUBSUBSCRIBE | - |
| QUIT | X |
| RANDOMKEY | X |
| RENAME | X |
| RENAMENX | X |
| RESTORE | X |
| ROLE | - |

| Command | Template Support |
|---|---|
| RPOP | X |
| RPOPLPUSH | X |
| RPUSH | X |
| RPUSHX | X |
| SADD | X |
| SAVE | X |
| SCAN | X |
| SCARD | X |
| SCRIPT EXITS | X |
| SCRIPT FLUSH | X |
| SCRIPT KILL | X |
| SCRIPT LOAD | X |
| SDIFF | X |
| SDIFFSTORE | X |
| SELECT | X |
| SENTINEL FAILOVER | X |
| SENTINEL GET-MASTER-ADD-BY-NAME | - |
| SENTINEL MASTER | - |
| SENTINEL MASTERS | X |
| SENTINEL MONITOR | X |
| SENTINEL REMOVE | X |
| SENTINEL RESET | - |
| SENTINEL SET | - |
| SENTINEL SLAVES | X |
| SET | X |
| SETBIT | X |
| SETEX | X |
| SETNX | X |
| SETRANGE | X |
| SHUTDOWN | X |
| SINTER | X |
| SINTERSTORE | X |
| SISMEMBER | X |
| SLAVEOF | X |

| Command | Template Support |
|---|---|
| SLOWLOG | - |
| SMEMBERS | X |
| SMOVE | X |
| SORT | X |
| SPOP | X |
| SRANDMEMBER | X |
| SREM | X |
| SSCAN | X |
| STRLEN | X |
| SUBSCRIBE | X |
| SUNION | X |
| SUNIONSTORE | X |
| SYNC | - |
| TIME | X |
| TTL | X |
| TYPE | X |
| UNSUBSCRIBE | X |
| UNWATCH | X |
| WATCH | X |
| ZADD | X |
| ZCARD | X |
| ZCOUNT | X |
| ZINCRBY | X |
| ZINTERSTORE | X |
| ZLEXCOUNT | - |
| ZRANGE | X |
| ZRANGEBYLEX | - |
| ZREVRANGEBYLEX | - |
| ZRANGEBYSCORE | X |
| ZRANK | X |
| ZREM | X |
| ZREMRANGEBYLEX | - |
| ZREMRANGEBYRANK | X |
| ZREVRANGE | X |
| ZREVRANGEBYSCORE | X |

| Command | Template Support |
|---|---|
| ZREVRANK | X |
| ZSCAN | X |
| ZSCORE | X |
| ZUNIONSTORE | X |