spring by Pivotal

Spring Data Redis

Costin Leau, Jennifer Hickey, Christoph Strobl, Thomas Darimont, Mark Paluch, Jay Bryant · – Version 2.6.0, 2021-11-12

Preface

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

This section provides an easy-to-follow guide for getting started with the Spring Data Redis module.

1. Learning Spring

Spring Data uses Spring framework’s core functionality, including:

- IoC container
- type conversion system
- expression language
- JMX integration
- DAO exception hierarchy.

While you need not know the Spring APIs, understanding the concepts behind them is important. At a minimum, the idea behind Inversion of Control (IoC) should be familiar, and you should be familiar with whatever IoC container you choose to use.

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

To learn more about Spring, you can refer to the comprehensive documentation that explains the Spring Framework in detail. There are a lot of articles, blog entries, and books on the subject. See the Spring framework home page for more information.

In general, this should be the starting point for developers wanting to try Spring Data Redis.

2. Learning 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 you be 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 does not take more then five to ten minutes to go through them and, if you come from an RDMBS-only background, many times these exercises can be eye-openers.

2.1. Trying out the Samples

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

3. Requirements

Spring Data Redis 2.x binaries require JDK level 8.0 and above and Spring Framework 5.3.13 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 4.0 release.

4. Additional Help Resources

Learning a new framework is not always straightforward. In this section, we try to provide what we think is an easy-to-follow guide for starting with the Spring Data Redis module. However, if you encounter issues or you need advice, feel free to use one of the following links:

Community Forum

Spring Data on Stack Overflow is a tag for all Spring Data (not just Document) users to share information and help each other. Note that registration is needed only for posting.

Professional Support

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

5. Following Development

For information on the Spring Data source code repository, nightly builds, and snapshot artifacts, 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 Stack Overflow at either spring-data or spring-data-redis.

If you encounter a bug or want to suggest an improvement (including to this documentation), please create a ticket on Github.

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.

6. New & Noteworthy

This section briefly covers items that are new and noteworthy in the latest releases.

6.1. New in Spring Data Redis 2.6

- Support for `SubscriptionListener` when using `MessageListener` for subscription confirmation callbacks. `ReactiveRedisMessageListenerContainer` and `ReactiveRedisOperations` provide `receiveLater(…)` and `listenToLater(…)` methods to await until Redis acknowledges the subscription.
- Support Redis 6.2 commands (`LPOP` / `RPOP` with `count` , `LMOVE` / `BLMOVE` , `COPY` , `GETEX` , `GETDEL` , `GEOSEARCH` , `GEOSEARCHSTORE` , `ZPOPMIN` , `BZPOPMIN` , `ZPOPMAX` , `BZPOPMAX` , `ZMSCORE` , `ZDIFF` , `ZDIFFSTORE` , `ZINTER` , `ZUNION` , `HRANDFIELD` , `ZRANDMEMBER` , `SMISMEMBER`).

6.2. New in Spring Data Redis 2.5

- `MappingRedisConverter` no longer converts byte arrays to a collection representation.

6.3. New in Spring Data Redis 2.4

- `RedisCache` now exposes `CacheStatistics`.
- ACL authentication support for Redis Standalone, Redis Cluster and Master/Replica.
- Password support for Redis Sentinel using Jedis.
- Support for `ZREVRANGEBYLEX` and `ZLEXCOUNT` commands.
- Support for Stream Commands using Jedis.

6.4. New in Spring Data Redis 2.3

- Template API Method Refinements for `Duration` and `Instant`.
- Extension of Stream Commands.

6.5. New in Spring Data Redis 2.2

- Redis Streams
- Refined `union` / `diff` / `intersect` set-operation methods accepting a single collection of keys.
- Upgrade to Jedis 3.
- Add support for scripting commands using Jedis Cluster.

6.6. New in Spring Data Redis 2.1

- Unix domain socket connections using Lettuce.
- Write to Master, read from Replica support using Lettuce.
- Query by Example integration.
- `@TypeAlias` Support for Redis repositories.
- Cluster-wide `SCAN` using Lettuce and `SCAN` on a selected node supported by both drivers.
- Reactive Pub/Sub to send and receive a message stream.
- `BITFIELD`, `BITPOS`, and `OBJECT` command support.
- Align return types of `BoundZSetOperations` with `ZSetOperations`.
- Reactive `SCAN`, `HSCAN`, `SSCAN`, and `ZSCAN` support.
- Usage of `IsTrue` and `IsFalse` keywords in repository query methods.

6.7. 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 with `JedisClientConfiguration` and `LettuceClientConfiguration`.
- Revised `RedisCache` implementation.

- Add `SPOP` with count command for Redis 3.2.

6.8. 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 repositories (see Persisting Partial Updates).
- SSL support for connections to Redis cluster.
- Support for `client name` through `ConnectionFactory` when using Jedis.

6.9. New in Spring Data Redis 1.7

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

6.10. 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`, including `+inf` / `-inf`.
- Performance improvements in `RedisCache`, now releasing connections earlier.

- Generic Jackson2 `RedisSerializer` making use of Jackson’s polymorphic deserialization.

6.11. 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).

7. Dependencies

Due to the different inception dates of individual Spring Data modules, most of them carry different major and minor version numbers. The easiest way to find compatible ones is to rely on the Spring Data Release Train BOM that we ship with the compatible versions defined. In a Maven project, you would declare this dependency in the `<dependencyManagement />` section of your POM as follows:

Example 1. Using the Spring Data release train BOM

```xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-bom</artifactId>
      <version>2021.1.0</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The current release train version is `2021.1.0`. The train version uses calver with the pattern `YYYY.MINOR.MICRO`. The version name follows `${calver}` for GA releases and service releases and the following pattern for all other versions: `${calver}-${modifier}`, where `modifier` can be one of the following:

- `SNAPSHOT`: Current snapshots
- `M1`, `M2`, and so on: Milestones
- `RC1`, `RC2`, and so on: Release candidates

You can find a working example of using the BOMs in our Spring Data examples repository. With that in place, you can declare the Spring Data modules you would like to use without a version in the `<dependencies />` block, as follows:

Example 2. Declaring a dependency to a Spring Data module

```xml
<dependencies>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
  </dependency>
```

```
<dependencies>
```

7.1. Dependency Management with Spring Boot

Spring Boot selects a recent version of Spring Data modules for you. If you still want to upgrade to a newer version, set the `spring-data-releasetrain.version` property to the train version and iteration you would like to use.

7.2. Spring Framework

The current version of Spring Data modules require Spring Framework 5.3.13 or better. The modules might also work with an older bugfix version of that minor version. However, using the most recent version within that generation is highly recommended.

Reference Documentation

8. Introduction

8.1. Document Structure

This part of the reference documentation explains the core functionality offered by Spring Data Redis. It explains Key-Value module concepts and semantics and the syntax for various stores namespaces. For an introduction to key-value stores, Spring, or Spring Data examples, see Learning NoSQL and Key Value Stores. This documentation refers only to Spring Data Redis Support and assumes the user is familiar with key-value storage and Spring concepts.

“Redis support” introduces the Redis module feature set.

“Redis Repositories” introduces the repository support for Redis.

This document is the reference guide for Spring Data Redis (SDR) Support.

9. 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 storage systems 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 (SDR) framework makes it easy to write Spring applications that use the Redis key-value store by eliminating the redundant tasks and boilerplate code required for interacting with the store through Spring’s excellent infrastructure support.

10. Redis support

One of the key-value stores supported by Spring Data is Redis. To quote the Redis 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.

10.1. Getting Started

An easy way to setting up a working environment is to create a Spring-based project in STS.

First, you need to set up a running Redis server.

To create a Spring project in STS:

1. Go to File → New → Spring Template Project → Simple Spring Utility Project, and press Yes when prompted. Then enter a project and a package name, such as `org.spring.redis.example` . .Add the following to the pom.xml files `dependencies` element:

```xml
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>2.6.0</version>
  </dependency>

</dependencies>
```

2. Change the version of Spring in the pom.xml to be

```xml
<spring.framework.version>5.3.13</spring.framework.version>
```

3. Add the following location of the Spring Milestone repository for Maven to your `pom.xml` such that it is at the same level of your `<dependencies/>` element:

```xml
<repositories>
  <repository>
    <id>spring-milestone</id>
```

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

The repository is also browseable here.

10.2. Redis Requirements

Spring Redis requires Redis 2.6 or above and Spring Data Redis integrates with Lettuce and Jedis, two popular open-source Java libraries for Redis.

10.3. Redis Support High-level View

The Redis support provides several components. For most tasks, the high-level abstractions and support services are the best choice. Note that, at any point, you can move between layers. For example, you can get a low-level connection (or even the native library) to communicate directly with Redis.

10.4. 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 you choose, you need to use only one set of Spring Data Redis APIs (which behaves consistently across all connectors): the `org.springframework.data.redis.connection` package and its `RedisConnection` and `RedisConnectionFactory` interfaces for working with and retrieving active connections to Redis.

10.4.1. RedisConnection and RedisConnectionFactory

`RedisConnection` provides the core 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 that you can switch the connectors without any code changes, as the operation semantics remain the same.

For the corner cases where the native library API is required, `RedisConnection` provides a dedicated method (`getNativeConnection`) that returns the raw, underlying object used for communication.

Active `RedisConnection` objects are created through `RedisConnectionFactory` . In addition, the factory acts as `PersistenceExceptionTranslator` objects, meaning that, once declared, they let you do transparent exception translation. For example, you can do exception translation through the use of the `@Repository` annotation and AOP. For more information, see the dedicated section in the Spring Framework documentation.

Depending on the underlying configuration, the factory can return a new connection or an existing connection (when 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.

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. The following overview explains features that are supported by the individual Redis connectors:

Table 1. Feature Availability across Redis Connectors

Supported Feature	Lettuce	Jedis
Standalone Connections	X	X

Supported Feature	**Lettuce**	**Jedis**
Master/Replica Connections	X	
Redis Sentinel	Master Lookup, Sentinel Authentication, Replica Reads	Master Lookup
Redis Cluster	Cluster Connections, Cluster Node Connections, Replica Reads	Cluster Connections, Cluster Node Connections
Transport Channels	TCP, OS-native TCP (epoll, kqueue), Unix Domain Sockets	TCP
Connection Pooling	X (using `commons-pool2`)	X (using `commons-pool2`)
Other Connection Features	Singleton-connection sharing for non-blocking commands	`JedisShardInfo` support
SSL Support	X	X
Pub/Sub	X	X
Pipelining	X	X
Transactions	X	X
Datatype support	Key, String, List, Set, Sorted Set, Hash, Server, Stream, Scripting, Geo, HyperLogLog	Key, String, List, Set, Sorted Set, Hash, Server, Scripting, Geo, HyperLogLog
Reactive (non-blocking) API	X	

10.4.2. Configuring the Lettuce Connector

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

Add the following to the pom.xml files `dependencies` element:

XML

```xml
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>io.lettuce</groupId>
    <artifactId>lettuce-core</artifactId>
    <version>6.1.5.RELEASE</version>
  </dependency>

</dependencies>
```

The following example shows how to create a new Lettuce connection factory:

JAVA

```java
@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` instances created by the `LettuceConnectionFactory` share the same thread-safe native connection for all non-blocking and non-transactional operations. To use a dedicated connection each time, set `shareNativeConnection` to `false` . `LettuceConnectionFactory` can also be configured to use a `LettucePool` for pooling blocking and transactional connections or all connections if `shareNativeConnection` is set to `false`.

Lettuce integrates with Netty's native transports, letting you use Unix domain sockets to communicate with Redis. Make sure to include the appropriate native transport dependencies that match your runtime environment. The following example shows how to create a Lettuce Connection factory for a Unix domain socket at `/var/run/redis.sock`:

```java
@Configuration
class AppConfig {

  @Bean
  public LettuceConnectionFactory redisConnectionFactory() {

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

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

10.4.3. Configuring the Jedis Connector

Jedis is a community-driven connector supported by the Spring Data Redis module through the `org.springframework.data.redis.connection.jedis` package.

Add the following to the pom.xml files `dependencies` element:

```xml
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>3.7.0</version>
  </dependency>

</dependencies>
```

In its simplest form, the Jedis configuration looks as follow:

```java
@Configuration
class AppConfig {

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

For production use, however, you might want to tweak settings such as the host or password, as shown in the following example:

```java
@Configuration
class RedisConfiguration {

  @Bean
  public JedisConnectionFactory redisConnectionFactory() {

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

10.4.4. Write to Master, Read from Replica

The Redis Master/Replica setup — without automatic failover (for automatic failover see: Sentinel) — not only allows data to be safely stored at more nodes. It also allows, by using Lettuce, reading data from replicas while pushing writes to the master. You can set the read/write strategy to be used by using `LettuceClientConfiguration`, as shown in the following example:

```java
@Configuration
class WriteToMasterReadFromReplicaConfiguration {

  @Bean
  public LettuceConnectionFactory redisConnectionFactory() {

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

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

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

For environments reporting non-public addresses through the `INFO` command (for example, when using AWS), use `RedisStaticMasterReplicaConfiguration` instead of `RedisStandaloneConfiguration`. Please note that `RedisStaticMasterReplicaConfiguration` does not support Pub/Sub because of missing Pub/Sub message propagation across individual servers.

10.5. Redis Sentinel Support

For dealing with high-availability Redis, Spring Data Redis has support for Redis Sentinel, using `RedisSentinelConfiguration`, as shown in the following example:

```java
/**
 * 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);
}
```

`RedisSentinelConfiguration` can also be defined with a `PropertySource`, which lets you set the following properties:

Configuration Properties

- `spring.redis.sentinel.master`: name of the master node.
- `spring.redis.sentinel.nodes`: Comma delimited list of host:port pairs.
- `spring.redis.sentinel.password`: The password to apply when authenticating with Redis Sentinel

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

10.6. 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 the Redis command reference) that offer rich, generified interfaces for working against a certain type or certain key (through the `KeyBound` interfaces) as described in the following table:

Table 2. Operational views

Interface	Description
Key Type Operations	
`GeoOperations`	Redis geospatial operations, such as `GEOADD`, `GEORADIUS`,…
`HashOperations`	Redis hash operations
`HyperLogLogOperations`	Redis HyperLogLog operations, such as `PFADD`, `PFCOUNT`,…
`ListOperations`	Redis list operations
`SetOperations`	Redis set operations

Interface	Description
`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.

`RedisTemplate` uses a Java-based serializer for most of its operations. This means that any object written or read by the template is serialized and deserialized through Java. You can change the serialization mechanism on the template, and the Redis module offers several implementations, which are 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. However, values can be null as long as the underlying serializer accepts them. Read the Javadoc of each serializer for more information.

For cases where you need a certain template view, declare the view as a dependency and inject the template. The container automatically performs the conversion, eliminating the `opsFor[X]` calls, as shown in the following example:

```xml
<?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 https://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="jedisConnectionFactor
  ...

</beans>
```

```java
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());
  }
}
```

10.7. String-focused Convenience Classes

Since it is 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). The following listings show an example:

XML
```xml
<?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 https://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="jedisConn
  ...
</beans>
```

JAVA
```java
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` let you talk directly to Redis through the `RedisCallback` interface. This feature gives complete control to you, as it talks directly to the `RedisConnection`. Note that the callback receives an instance of `StringRedisConnection` when a `StringRedisTemplate` is used. The following example shows how to use the `RedisCallback` interface:

JAVA
```java
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");
    }
   });
}
```

10.8. Serializers

From the framework perspective, the data stored in Redis is only 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.

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

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

- Two-way serializers based on `RedisSerializer`.
- Element readers and writers that use `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 (including two that have been already mentioned in this documentation):

- `JdkSerializationRedisSerializer`, which is used by default for `RedisCache` and `RedisTemplate`.

- the `StringRedisSerializer`.

However one can use `OxmSerializer` for Object/XML mapping through Spring OXM support or `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.

By default, `RedisCache` and `RedisTemplate` are configured to use Java native serialization. Java native serialization is known for allowing the running of remote code caused by payloads that exploit vulnerable libraries and classes injecting unverified bytecode. Manipulated input could lead to unwanted code being run in the application during the deserialization step. As a consequence, do not use serialization in untrusted environments. In general, we strongly recommend any other message format (such as JSON) instead.

If you are concerned about security vulnerabilities due to Java serialization, consider the general-purpose serialization filter mechanism at the core JVM level, originally developed for JDK 9 but backported to JDK 8, 7, and 6:

- Filter Incoming Serialization Data.
- JEP 290.
- OWASP: Deserialization of untrusted data.

10.9. Hash mapping

Data can be stored by using various data structures within Redis. `Jackson2JsonRedisSerializer` can convert objects in JSON format. Ideally, JSON can be stored as a value by using plain keys. You can achieve a more sophisticated mapping of structured objects by using Redis hashes. Spring Data Redis offers various strategies for mapping data to hashes (depending on the use case):

- Direct mapping, by using `HashOperations` and a serializer
- Using Redis Repositories
- Using `HashMapper` and `HashOperations`

10.9.1. Hash Mappers

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

Multiple implementations are available:

- `BeanUtilsHashMapper` using Spring's BeanUtils.
- `ObjectHashMapper` using Object-to-Hash Mapping.
- `Jackson2HashMapper` using FasterXML Jackson.

The following example shows one way to implement hash mapping:

JAVA

```java
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);
  }
}
```

10.9.2. Jackson2HashMapper

`Jackson2HashMapper` provides Redis Hash mapping for domain objects by using FasterXML Jackson. `Jackson2HashMapper` can 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, and so on) 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.

Consider the following class and the data structure it contains:

```java
public class Person {
  String firstname;
  String lastname;
  Address address;
  Date date;
  LocalDateTime localDateTime;
}

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

The following table shows how the data in the preceding class would appear in normal mapping:

Table 3. Normal Mapping

Hash Field	Value
firstname	`Jon`
lastname	`Snow`
address	`{ "city" : "Castle Black", "country" : "The North" }`
date	`1561543964015`
localDateTime	`2018-01-02T12:13:14`

The following table shows how the data in the preceding class would appear in flat mapping:

Table 4. Flat Mapping

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

Hash Field	Value
date	`1561543964015`
localDateTime	`2018-01-02T12:13:14`

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 when you use flattening. The resulting hash cannot be mapped back into an Object.

`java.util.Date` and `java.util.Calendar` are represented with milliseconds. JSR-310 Date/Time types are serialized to their `toString` form if `jackson-datatype-jsr310` is on the class path.

10.10. Redis Messaging (Pub/Sub)

Spring Data provides dedicated messaging integration for Redis, similar in functionality and naming to the JMS integration in Spring Framework.

Redis messaging can be roughly divided into two areas of functionality:

- Publication or production of messages
- Subscription or consumption of messages

This is an example of the pattern often called Publish/Subscribe (Pub/Sub for short). 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 `org.springframework.data.redis.connection` and `org.springframework.data.redis.listener` packages provide the core functionality for Redis messaging.

10.10.1. Publishing (Sending Messages)

To publish a message, you can use, as with the other operations, either the low-level `RedisConnection` or the high-level `RedisTemplate`. Both entities offer the `publish` method, which accepts the message and the destination channel as arguments. While `RedisConnection` requires raw data (array of bytes), the `RedisTemplate` lets arbitrary objects be passed in as messages, as shown in the following example:

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

10.10.2. Subscribing (Receiving 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 lets multiple subscriptions be created with one command but can also listen on channels not yet created at subscription time (as long as they match the pattern).

At the low-level, `RedisConnection` offers the `subscribe` and `pSubscribe` methods that map the Redis commands for subscribing by channel or by pattern, respectively. Note that multiple channels or patterns can be used as arguments. To change the subscription of a connection or query whether it is listening, `RedisConnection` provides the `getSubscription` and `isSubscribed` methods.

Subscription commands in Spring Data Redis are blocking. That is, calling subscribe on a connection causes the current thread to block as it starts waiting for messages. The thread is released only if the subscription is canceled, which happens when another thread invokes `unsubscribe` or `pUnsubscribe` on the **same** connection. See “Message Listener Containers” (later in this document) for a solution to this problem.

As mentioned earlier, once subscribed, a connection starts waiting for messages. Only commands that add new subscriptions, modify existing subscriptions, and cancel existing subscriptions are allowed. Invoking anything other than `subscribe`, `pSubscribe`, `unsubscribe`, or `pUnsubscribe` throws an exception.

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

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. If you are familiar with EJB and JMS, you should find the concepts familiar, as it is designed to be 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` instances 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 lets you as an application developer write the (possibly complex) business logic associated with receiving a message (and reacting to it) and delegates boilerplate Redis infrastructure concerns to the framework.

```
A `MessageListener` can additionally implement `SubscriptionListener` to receive notifications upon subscription/unsubscribe confirm
```

Furthermore, to minimize the application footprint, `RedisMessageListenerContainer` lets one connection and one thread 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 remains the same throughout its lifetime. Moreover, the container allows runtime configuration changes so that you can add or remove listeners while an application is running without the need for a 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 thread is released.

To help with the asynchronous nature 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, you should change or tweak the executor to better serve your needs. In particular, in managed environments (such as app servers), it is highly recommended to pick 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 lets you expose almost **any** class as a MDP (though there are some constraints).

Consider the following interface definition:

```java
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);
 }
```

Notice that, although the interface does not extend the `MessageListener` interface, it can still be used as a MDP by using 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`:

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

Notice 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 make into an MDP with the following configuration:

```xml
<?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 https://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/redis https://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>
```

The listener topic can be either a channel (for example, `topic="chatroom"`) or a pattern (for example, `topic="*room"`)

The preceding example uses the Redis namespace to declare the message listener container and automatically register the POJOs as listeners. The full blown beans definition follows:

```xml
<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 and transparently performs translation (using the configured `RedisSerializer`) between the low-level format and the required object type. Any exception caused by the method invocation is caught and handled by the container (by default, exceptions get logged).

10.11. Redis Streams

Redis Streams model a log data structure in an abstract approach. Typically, logs are append-only data structures and are consumed from the beginning on, at a random position, or by streaming new messages.

Learn more about Redis Streams in the Redis reference documentation.

Redis Streams can be roughly divided into two areas of functionality:

- Appending records
- Consuming records

Although this pattern has similarities to Pub/Sub, the main difference lies in the persistence of messages and how they are consumed.

While Pub/Sub relies on the broadcasting of transient messages (i.e. if you don’t listen, you miss a message), Redis Stream use a persistent, append-only data type that retains messages until the stream is trimmed. Another difference in consumption is that Pub/Sub registers a server-side subscription. Redis pushes arriving messages to the client while Redis Streams require active polling.

The `org.springframework.data.redis.connection` and `org.springframework.data.redis.stream` packages provide the core functionality for Redis Streams.

10.11.1. Appending

To send a record, you can use, as with the other operations, either the low-level `RedisConnection` or the high-level `StreamOperations`. Both entities offer the `add` (`xAdd`) method, which accepts the record and the destination stream as arguments. While `RedisConnection` requires raw data (array of bytes), the `StreamOperations` lets arbitrary objects be passed in as records, as shown in the following example:

```java
// append message through connection
RedisConnection con = …
byte[] stream = …
ByteRecord record = StreamRecords.rawBytes(…).withStreamKey(stream);
con.xAdd(record);

// append message through RedisTemplate
RedisTemplate template = …
```

```
StringRecord record = StreamRecords.string(…).withStreamKey("my-stream");
template.streamOps().add(record);
```

Stream records carry a `Map`, key-value tuples, as their payload. Appending a record to a stream returns the `RecordId` that can be used as further reference.

10.11.2. Consuming

On the consuming side, one can consume one or multiple streams. Redis Streams provide read commands that allow consumption of the stream from an arbitrary position (random access) within the known stream content and beyond the stream end to consume new stream record.

At the low-level, `RedisConnection` offers the `xRead` and `xReadGroup` methods that map the Redis commands for reading and reading within a consumer group, respectively. Note that multiple streams can be used as arguments.

> Subscription commands in Redis can be blocking. That is, calling `xRead` on a connection causes the current thread to block as it starts waiting for messages. The thread is released only if the read command times out or receives a message.

To consume stream messages, one can either poll for messages in application code, or use one of the two Asynchronous reception through Message Listener Containers, the imperative or the reactive one. Each time a new records arrives, the container notifies the application code.

Synchronous reception

While stream consumption is typically associated with asynchronous processing, it is possible to consume messages synchronously. The overloaded `StreamOperations.read(…)` methods provide this functionality. During a synchronous receive, the calling thread potentially blocks until a message becomes available. The property `StreamReadOptions.block` specifies how long the receiver should wait before giving up waiting for a message.

```java
// Read message through RedisTemplate
RedisTemplate template = …

List<MapRecord<K, HK, HV>> messages = template.streamOps().read(StreamReadOptions.empty().count(2),
                StreamOffset.latest("my-stream"));

List<MapRecord<K, HK, HV>> messages = template.streamOps().read(Consumer.from("my-group", "my-consumer"),
                StreamReadOptions.empty().count(2),
                StreamOffset.create("my-stream", ReadOffset.lastConsumed()))
```

Asynchronous reception through Message Listener Containers

Due to its blocking nature, low-level polling is not attractive, as it requires connection and thread management for every single consumer. To alleviate this problem, Spring Data offers message listeners, which do all the heavy lifting. If you are familiar with EJB and JMS, you should find the concepts familiar, as it is designed to be as close as possible to the support in Spring Framework and its message-driven POJOs (MDPs).

Spring Data ships with two implementations tailored to the used programming model:

- `StreamMessageListenerContainer` acts as message listener container for imperative programming models. It is used to consume records from a Redis Stream and drive the `StreamListener` instances that are injected into it.
- `StreamReceiver` provides a reactive variant of a message listener. It is used to consume messages from a Redis Stream as potentially infinite stream and emit stream messages through a `Flux` .

`StreamMessageListenerContainer` and `StreamReceiver` are responsible for all threading of message reception and dispatch into the listener for processing. A message listener container/receiver 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 lets you as an application developer write the (possibly complex) business logic associated with receiving a message (and reacting to it) and delegates boilerplate Redis infrastructure concerns to the framework.

Both containers allow runtime configuration changes so that you can add or remove subscriptions while an application is running without the need for a restart. Additionally, the container uses a lazy subscription approach, using a `RedisConnection` only when needed. If all the listeners are unsubscribed, it automatically performs a cleanup, and the thread is released.

Imperative `StreamMessageListenerContainer`

In a fashion similar to a Message-Driven Bean (MDB) in the EJB world, the Stream-Driven POJO (SDP) acts as a receiver for Stream messages. The one restriction on an SDP is that it must implement the `org.springframework.data.redis.stream.StreamListener` interface. Please also be aware that in the case where your POJO receives messages on multiple threads, it is important to ensure that your implementation is thread-safe.

```java
class ExampleStreamListener implements StreamListener<String, MapRecord<String, String, String>> {

    @Override
    public void onMessage(MapRecord<String, String, String> message) {

        System.out.println("MessageId: " + message.getId());
        System.out.println("Stream: " + message.getStream());
        System.out.println("Body: " + message.getValue());
    }
}
```

`StreamListener` represents a functional interface so implementations can be rewritten using their Lambda form:

```java
message -> {

    System.out.println("MessageId: " + message.getId());
    System.out.println("Stream: " + message.getStream());
    System.out.println("Body: " + message.getValue());
};
```

Once you've implemented your `StreamListener`, it's time to create a message listener container and register a subscription:

```java
JAVA
RedisConnectionFactory connectionFactory = …
StreamListener<String, MapRecord<String, String, String>> streamListener = …

StreamMessageListenerContainerOptions<String, MapRecord<String, String, String>> containerOptions = StreamMessageListenerContainerOp
            .builder().pollTimeout(Duration.ofMillis(100)).build();

StreamMessageListenerContainer<String, MapRecord<String, String, String>> container = StreamMessageListenerContainer.create(connecti
                containerOptions);

Subscription subscription = container.receive(StreamOffset.fromStart("my-stream"), streamListener);
```

Please refer to the Javadoc of the various message listener containers for a full description of the features supported by each implementation.

Reactive `StreamReceiver`

Reactive consumption of streaming data sources typically happens through a `Flux` of events or messages. The reactive receiver implementation is provided with `StreamReceiver` and its overloaded `receive(…)` messages. The reactive approach requires fewer infrastructure resources such as threads in comparison to `StreamMessageListenerContainer` as it is leveraging threading resources provided by the driver. The receiving stream is a demand-driven publisher of `StreamMessage`:

```java
JAVA
Flux<MapRecord<String, String, String>> messages = …

return messages.doOnNext(it -> {
    System.out.println("MessageId: " + message.getId());
    System.out.println("Stream: " + message.getStream());
    System.out.println("Body: " + message.getValue());
});
```

Now we need to create the `StreamReceiver` and register a subscription to consume stream messages:

```java
JAVA
ReactiveRedisConnectionFactory connectionFactory = …
```

```
StreamReceiverOptions<String, MapRecord<String, String, String>> options = StreamReceiverOptions.builder().pollTimeout(Duration.ofMi
                .build();
StreamReceiver<String, MapRecord<String, String, String>> receiver = StreamReceiver.create(connectionFactory, options);

Flux<MapRecord<String, String, String>> messages = receiver.receive(StreamOffset.fromStart("my-stream"));
```

Please refer to the Javadoc of the various message listener containers for a full description of the features supported by each implementation.

Demand-driven consumption uses backpressure signals to activate and deactivate polling. `StreamReceiver` subscriptions pause polling if the demand is satisfied until subscribers signal further demand. Depending on the `ReadOffset` strategy, this can cause messages to be skipped.

`Acknowledge` strategies

When you read with messages via a `Consumer Group`, the server will remember that a given message was delivered and add it to the Pending Entries List (PEL). A list of messages delivered but not yet acknowledged.
Messages have to be acknowledged via `StreamOperations.acknowledge` in order to be removed from the Pending Entries List as shown in the snippet below.

```java
StreamMessageListenerContainer<String, MapRecord<String, String, String>> container = ...

container.receive(Consumer.from("my-group", "my-consumer"),  1
    StreamOffset.create("my-stream", ReadOffset.lastConsumed()),
    msg -> {

        // ...
        redisTemplate.opsForStream().acknowledge("my-group", msg);  2
    });
```

1 Read as *my-consumer* from group *my-group*. Received messages are not acknowledged.

2 Acknowledged the message after processing.

To auto acknowledge messages on receive use `receiveAutoAck` instead of `receive`.

`ReadOffset` strategies

Stream read operations accept a read offset specification to consume messages from the given offset on. `ReadOffset` represents the read offset specification. Redis supports 3 variants of offsets, depending on whether you consume the stream standalone or within a consumer group:

- `ReadOffset.latest()` – Read the latest message.
- `ReadOffset.from(…)` – Read after a specific message Id.
- `ReadOffset.lastConsumed()` – Read after the last consumed message Id (consumer-group only).

In the context of a message container-based consumption, we need to advance (or increment) the read offset when consuming a message. Advancing depends on the requested `ReadOffset` and consumption mode (with/without consumer groups). The following matrix explains how containers advance `ReadOffset`:

Table 5. ReadOffset Advancing

Read offset	Standalone	Consumer Group
Last Consumed	Use last seen message as the next MessageId	Last consumed message as per consumer group

Read offset	Standalone	Consumer Group
Latest	Read latest message	Read latest message
Specific Message Id	Use last seen message as the next MessageId	Use last seen message as the next MessageId
Last Consumed	Use last seen message as the next MessageId	Last consumed message as per consumer group

Reading from a specific message id and the last consumed message can be considered safe operations that ensure consumption of all messages that were appended to the stream. Using the latest message for read can skip messages that were added to the stream while the poll operation was in the state of dead time. Polling introduces a dead time in which messages can arrive between individual polling commands. Stream consumption is not a linear contiguous read but split into repeating `XREAD` calls.

Serialization

Any Record sent to the stream needs to be serialized to its binary format. Due to the streams closeness to the hash data structure the stream key, field names and values use the according serializers configured on the `RedisTemplate`.

Table 6. Stream Serialization

Stream Property	Serializer	Description
key	keySerializer	used for `Record#getStream()`
field	hashKeySerializer	used for each map key in the payload
value	hashValueSerializer	used for each map value in the payload

Please make sure to review `RedisSerializer` s in use and note that if you decide to not use any serializer you need to make sure those values are binary already.

Object Mapping

Simple Values

`StreamOperations` allows to append simple values, via `ObjectRecord`, directly to the stream without having to put those values into a `Map` structure. The value will then be assigned to an *payload* field and can be extracted when reading back the value.

```java
ObjectRecord<String, String> record = StreamRecords.newRecord()
    .in("my-stream")
    .ofObject("my-value");

redisTemplate()
    .opsForStream()
    .add(record);  1

List<ObjectRecord<String, String>> records = redisTemplate()
    .opsForStream()
    .read(String.class, StreamOffset.fromStart("my-stream"));
```

1 XADD my-stream * "_class" "java.lang.String" "_raw" "my-value"

`ObjectRecord` s pass through the very same serialization process as the all other records, thus the Record can also obtained using the untyped read operation returning a `MapRecord`.

Complex Values

Adding a complex value to the stream can be done in 3 ways:

- Convert to simple value using eg. a String JSON representation.
- Serialize the value with a suitable `RedisSerializer`.
- Convert the value into a `Map` suitable for serialization using a `HashMapper`.

The first variant is the most straight forward one but neglects the field value capabilities offered by the stream structure, still the values in the stream will be readable for other consumers. The 2nd option holds the same benefits as the first one, but may lead to a very specific consumer limitations as the all consumers must implement the very same serialization mechanism. The `HashMapper` approach is the a bit more complex one making use of the steams hash structure, but flattening the source. Still other consumers remain able to read the records as long as suitable serializer combinations are chosen.

HashMappers convert the payload to a `Map` with specific types. Make sure to use Hash-Key and Hash-Value serializers that are capable of (de-)serializing the hash.

JAVA

```
ObjectRecord<String, User> record = StreamRecords.newRecord()
    .in("user-logon")
    .ofObject(new User("night", "angel"));

redisTemplate()
    .opsForStream()
    .add(record);  1

List<ObjectRecord<String, User>> records = redisTemplate()
    .opsForStream()
    .read(User.class, StreamOffset.fromStart("user-logon"));
```

1 XADD user-logon * "_class" "com.example.User" "firstname" "night" "lastname" "angel"

`StreamOperations` use by default ObjectHashMapper. You may provide a `HashMapper` suitable for your requirements when obtaining `StreamOperations` .

JAVA

```
redisTemplate()
    .opsForStream(new Jackson2HashMapper(true))
    .add(record);  1
```

1 XADD user-logon * "firstname" "night" "@class" "com.example.User" "lastname" "angel"

A `StreamMessageListenerContainer` may not be aware of any `@TypeAlias` used on domain types as those need to be resolved through a `MappingContext`. Make sure to initialize `RedisMappingContext` with a `initialEntitySet`.

JAV

```java
@Bean
RedisMappingContext redisMappingContext() {
    RedisMappingContext ctx = new RedisMappingContext();
    ctx.setInitialEntitySet(Collections.singleton(Person.class));
    return ctx;
}

@Bean
RedisConverter redisConverter(RedisMappingContext mappingContext) {
    return new MappingRedisConverter(mappingContext);
}

@Bean
ObjectHashMapper hashMapper(RedisConverter converter) {
    return new ObjectHashMapper(converter);
}

@Bean
StreamMessageListenerContainer streamMessageListenerContainer(RedisConnectionFactory connectionFactory, ObjectHashMapper hashMapper) {
    StreamMessageListenerContainerOptions<String, ObjectRecord<String, Object>> options = StreamMessageListenerContainerOptions.builder()
            .objectMapper(hashMapper)
            .build();

    return StreamMessageListenerContainer.create(connectionFactory, options);
}
```

10.12. 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 run all the operations in the transaction with the same connection.

Spring Data Redis provides the `SessionCallback` interface for use when multiple operations need to be performed with the same `connection`, such as when using Redis transactions.The following example uses the `multi` method:

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

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

As of version 1.1, an important change has been made to the `exec` methods of `RedisConnection` and `RedisTemplate`. 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 whether 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 such as `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, set `convertPipelineAndTxResults` to `false` on your `RedisConnectionFactory` to disable this behavior.

10.12.1. @Transactional Support

By default, `RedisTemplate` does not participate in managed Spring transactions. If you want `RedisTemplate` to make use of Redis transaction when using `@Transactional` or `TransactionTemplate`, you need to be explicitly enable transaction support for each `RedisTemplate` by setting `setEnableTransactionSupport(true)`. Enabling transaction support binds `RedisConnection` to the current transaction backed by a `ThreadLocal`. If the transaction finishes without errors, the Redis transaction gets commited with `EXEC`, otherwise rolled back with `DISCARD`. Redis transactions are batch-oriented. Commands issued during an ongoing transaction are queued and only applied when committing the transaction.

Spring Data Redis distinguishes between read-only and write commands in an ongoing transaction. Read-only commands, such as `KEYS`, are piped to a fresh (non-thread-bound) `RedisConnection` to allow reads. Write commands are queued by `RedisTemplate` and applied upon commit.

The following example shows how to configure transaction management:

Example 3. Configuration enabling Transaction Management

```java
@Configuration
@EnableTransactionManagement                                 1
public class RedisTxContextConfiguration {

  @Bean
  public StringRedisTemplate redisTemplate() {
    StringRedisTemplate template = new StringRedisTemplate(redisConnectionFactory());
    // explicitly enable transaction support
    template.setEnableTransactionSupport(true);              2
    return template;
  }
  @Bean
  public RedisConnectionFactory redisConnectionFactory() {
    // jedis || Lettuce
  }

  @Bean
  public PlatformTransactionManager transactionManager() throws SQLException {
    return new DataSourceTransactionManager(dataSource());    3
  }

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

1 Configures a Spring Context to enable declarative transaction management.

2 Configures `RedisTemplate` to participate in transactions by binding connections to the current thread.

3 Transaction management requires a `PlatformTransactionManager`.
Spring Data Redis does not ship with a `PlatformTransactionManager` implementation.
Assuming your application uses JDBC, Spring Data Redis can participate in transactions by using existing transaction managers.

The following examples each demonstrate a usage constraint:

Example 4. Usage Constraints

JAVA

```
// must be performed on thread-bound connection
template.opsForValue().set("thing1", "thing2");

// read operation must be run on a free (not transaction-aware) connection
template.keys("*");
```

```
// returns null as values set within a transaction are not visible
template.opsForValue().get("thing1");
```

10.13. 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 running commands in a pipeline. If you do not care about the results of the pipelined operations, you can use the standard `execute` method, passing `true` for the `pipeline` argument. The `executePipelined` methods run the provided `RedisCallback` or `SessionCallback` in a pipeline and return the results, as shown in the following example:

```java
//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 preceding example runs 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 preceding example are Strings. There are additional `executePipelined` methods that let you 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.

The Lettuce driver supports fine grained flush control that allows to either flush commands as they appear, buffer or send them at connection close.

```java
LettuceConnectionFactory factory = // ...
factory.setPipeliningFlushPolicy(PipeliningFlushPolicy.buffered(3));  1
```

1 Buffer locally and flush after every 3rd command.

As of version 1.1, an important change has been made to the `exec` methods of `RedisConnection` and `RedisTemplate`. 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 whether 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 such as `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, set `convertPipelineAndTxResults` to `false` on your `RedisConnectionFactory` to disable this behavior.

10.14. Redis Scripting

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

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

The following example runs a common “check-and-set” scenario by using a Lua script. This is an ideal use case for a Redis script, as it requires that running a set of commands atomically, and the behavior of one command is influenced by the result of another.

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

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

JAVA
```java
public class Example {

  @Autowired
  RedisScript<Boolean> script;

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

LUA
```lua
-- 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 preceding code 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 a deserialized value type. It can also be `null` if the script returns a throw-away status (specifically, `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 run.

The `checkAndSet` method above then runs the scripts. Scripts can be run 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 lets you schedule Redis scripts for periodic running by using the Spring Task and Scheduler abstractions. See the Spring Framework documentation for more details.

10.14.1. Redis Cache

Changed in 2.0

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

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

`RedisCacheManager` behavior can be configured with `RedisCacheManagerBuilder`, letting you set the default `RedisCacheConfiguration`, transaction behavior, and predefined caches.

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

As shown in the preceding example, `RedisCacheManager` allows definition of configurations on a per-cache basis.

The behavior of `RedisCache` created with `RedisCacheManager` is defined with `RedisCacheConfiguration`. The configuration lets you set key expiration times, prefixes, and `RedisSerializer` implementations for converting to and from the binary storage format, as shown in the following example:

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

`RedisCacheManager` defaults to a lock-free `RedisCacheWriter` for reading and writing binary values. Lock-free caching improves throughput. The lack of entry locking can lead to overlapping, non-atomic commands for the `putIfAbsent` and `clean` methods, as those require multiple commands to be 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.

Locking applies on the **cache level**, not per **cache entry**.

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

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

By default, any `key` for a cache entry gets prefixed with the actual cache name followed by two colons. This behavior can be changed to a static as well as a computed prefix.

The following example shows how to set a static prefix:

```java
// static key prefix
RedisCacheConfiguration.defaultCacheConfig().prefixKeysWith("( ͡° ͜ʖ ͡°)");

The following example shows how to set a computed prefix:

// computed key prefix
RedisCacheConfiguration.defaultCacheConfig().computePrefixWith(cacheName -> "¯\_(ツ)_/¯" + cacheName);
```

The cache implementation defaults to use `KEYS` and `DEL` to clear the cache. `KEYS` can cause performance issues with large keyspaces. Therefore, the default `RedisCacheWriter` can be created with a `BatchStrategy` to switch to a `SCAN` -based batch strategy. The `SCAN` strategy requires a batch size to avoid excessive Redis command roundtrips:

```java
RedisCacheManager cm = RedisCacheManager.build(RedisCacheWriter.nonLockingRedisCacheWriter(connectionFactory, BatchStrategies.scan(1
    .cacheDefaults(defaultCacheConfig())
    ...
```

The `KEYS` batch strategy is fully supported using any driver and Redis operation mode (Standalone, Clustered). `SCAN` is fully supported when using the Lettuce driver. Jedis supports `SCAN` only in non-clustered modes.

The following table lists the default settings for `RedisCacheManager`:

Table 7. `RedisCacheManager` defaults

Setting	Value
Cache Writer	Non-locking, `KEYS` batch strategy
Cache Configuration	`RedisCacheConfiguration#defaultConfiguration`
Initial Caches	None
Transaction Aware	No

The following table lists the default settings for `RedisCacheConfiguration`:

Table 8. RedisCacheConfiguration defaults

Key Expiration	None
Cache `null`	Yes
Prefix Keys	Yes
Default Prefix	The actual cache name
Key Serializer	`StringRedisSerializer`
Value Serializer	`JdkSerializationRedisSerializer`

Key Expiration	**None**
Conversion Service	`DefaultFormattingConversionService` with default cache key converters

By default `RedisCache`, statistics are disabled. Use `RedisCacheManagerBuilder.enableStatistics()` to collect local *hits* and *misses* through `RedisCache#getStatistics()`, returning a snapshot of the collected data.

10.15. 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`. `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. The following example shows the configuration for a bean that uses a `RedisList`:

```xml
<?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 https://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>
```

The following example shows a Java configuration example for a `Deque` :

```
JAVA
public class AnotherExample {

  // injected
  private Deque<String> queue;

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

As shown in the preceding example, 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 be replaced with an in-memory one).

11. Reactive Redis support

This section covers reactive Redis support and how to get started. Reactive Redis support naturally has certain overlaps with imperative Redis support.

11.1. Redis Requirements

Spring Data Redis currently integrates with Lettuce as the only reactive Java connector. Project Reactor is used as reactive composition library.

11.2. Connecting to Redis by 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 you choose, you must use the `org.springframework.data.redis.connection` package and its `ReactiveRedisConnection` and `ReactiveRedisConnectionFactory` interfaces to work with and retrieve active `connections` to Redis.

11.2.1. Redis Operation Modes

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

11.2.2. `ReactiveRedisConnection` and `ReactiveRedisConnectionFactory`

`ReactiveRedisConnection` is the core of 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 you can switch the connectors without any code changes, as the operation semantics remain the same.

`ReactiveRedisConnectionFactory` creates active `ReactiveRedisConnection` instances. In addition, the factories act as `PersistenceExceptionTranslator` instances, meaning that, once declared, they let you 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 the Spring Framework documentation.

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.

11.2.3. Configuring a Lettuce Connector

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

You can set up `ReactiveRedisConnectionFactory` for Lettuce as follows:

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

The following example shows a more sophisticated configuration, including SSL and timeouts, that uses `LettuceClientConfigurationBuilder`:

```java
@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, see `LettuceClientConfiguration`.

11.3. Working with Objects through ReactiveRedisTemplate

Most users are likely to use `ReactiveRedisTemplate` and its corresponding package, `org.springframework.data.redis.core`. Due to its rich feature set, the template is, in fact, the central class of the Redis module. 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 you 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 in the following table:

Table 9. Operational views

Interface	Description
Key Type Operations	
ReactiveGeoOperations	Redis geospatial operations such as `GEOADD` , `GEORADIUS` , and others)
ReactiveHashOperations	Redis hash operations
ReactiveHyperLogLogOperations	Redis HyperLogLog operations such as (`PFADD` , `PFCOUNT` , and others)
ReactiveListOperations	Redis list operations
ReactiveSetOperations	Redis set operations
ReactiveValueOperations	Redis string (or value) operations

Interface	Description
ReactiveZSetOperations	Redis zset (or sorted set) operations

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

`ReactiveRedisTemplate` uses a Java-based serializer for most of its operations. This means that any object written or read by the template is serialized or deserialized through `RedisElementWriter` or `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.

The following example shows a `ReactiveRedisTemplate` being used to return a `Mono`:

JAVA

```java
@Configuration
class RedisConfiguration {

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

JAVA

```java
public class Example {

  @Autowired
  private ReactiveRedisTemplate<String, String> template;

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

11.4. String-focused Convenience Classes

Since it is 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`. It 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`, which means the stored keys and values are human readable (assuming the same encoding is used in both Redis and your code). The following example shows `ReactiveStringRedisTemplate` in use:

```java
@Configuration
class RedisConfiguration {

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

```java
public class Example {

  @Autowired
  private ReactiveStringRedisTemplate redisTemplate;

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

11.5. 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 `ReactiveRedisTemplate` class is used for message production. For asynchronous reception, Spring Data provides a dedicated message listener container that is used consume a stream of messages. For the purpose of just subscribing `ReactiveRedisTemplate` offers stripped down alternatives to utilizing a listener container.

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

11.5.1. Sending/Publishing messages

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

```java
// send message through ReactiveRedisConnection
ByteBuffer msg = …
ByteBuffer channel = …
Mono<Long> publish = con.publish(msg, channel);

// send message through ReactiveRedisTemplate
ReactiveRedisTemplate template = …
Mono<Long> publish = template.convertAndSend("channel", "message");
```

11.5.2. 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, `ReactiveRedisConnection` 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 a subscription, simply query the channels and patterns of `ReactiveSubscription`.

Reactive subscription commands in Spring Data Redis are non-blocking and may end without emitting an element.

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. Commands other than `subscribe`, `pSubscribe`, `unsubscribe`, or `pUnsubscribe` are illegal and will cause an exception.

In order to receive messages, one needs to obtain the message stream. Note that a subscription only publishes messages for channels and patterns that are registered with that particular subscription. The message stream itself is a hot sequence that produces elements without regard to demand. Make sure to register sufficient demand to not exhaust the message buffer.

Message Listener Containers

Spring Data offers `ReactiveRedisMessageListenerContainer` which does all the heavy lifting of conversion and subscription state management on behalf of the user.

`ReactiveRedisMessageListenerContainer` acts as a message listener container. It is used to receive messages from a Redis channel and expose a stream of messages that emits channel messages with deserialization applied. It 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. Message streams register a subscription in Redis upon publisher subscription and unregister if the subscription gets canceled.

Furthermore, to minimize the application footprint, `ReactiveRedisMessageListenerContainer` 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 `ReactiveRedisConnection` only when needed - if all the listeners are unsubscribed, cleanup is automatically performed.

The message listener container itself does not require external threading resources. It uses the driver threads to publish messages.

```java
ReactiveRedisConnectionFactory factory = …
ReactiveRedisMessageListenerContainer container = new ReactiveRedisMessageListenerContainer(factory);

Flux<ChannelMessage<String, String>> stream = container.receive(ChannelTopic.of("my-channel"));
```

To await and ensure proper subscription, you can use the `receiveLater` method that returns a `Mono<Flux<ChannelMessage>>`. The resulting `Mono` completes with an inner publisher as a result of completing the subscription to the given topics. By intercepting `onNext` signals, you can synchronize server-side subscriptions.

```java
ReactiveRedisConnectionFactory factory = …
ReactiveRedisMessageListenerContainer container = new ReactiveRedisMessageListenerContainer(factory);

Mono<Flux<ChannelMessage<String, String>>> stream = container.receiveLater(ChannelTopic.of("my-channel"));

stream.doOnNext(inner -> // notification hook when Redis subscriptions are synchronized with the server)
    .flatMapMany(Function.identity())
    .…;
```

Subscribing via template API

As mentioned above you can directly use `ReactiveRedisTemplate` to subscribe to channels / patterns. This approach offers a straight forward, though limited solution as you lose the option to add subscriptions after the initial ones. Nevertheless you still can control the message stream via the returned `Flux` using eg. `take(Duration)`. When done reading, on error or cancellation all bound resources are freed again.

```java
redisTemplate.listenToChannel("channel1", "channel2").doOnNext(msg -> {
    // message processing ...
}).subscribe();
```

11.6. Reactive Scripting

You can run Redis scripts with the reactive infrastructure by using the `ReactiveScriptExecutor`, which is best accessed through `ReactiveRedisTemplate`.

```java
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);
  }
}
```

See to the scripting section for more details on scripting commands.

12. Redis Cluster

Working with Redis Cluster requires Redis Server version 3.0+. See the Cluster Tutorial for more information.

12.1. Enabling Redis Cluster

Cluster support is based on the 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` instances are created with the `RedisConnectionFactory`, which has to be set up with the associated `RedisClusterConfiguration`, as shown in the following example:

Example 5. Sample RedisConnectionFactory Configuration for Redis Cluster

```java
@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()));
    }
}
```

`RedisClusterConfiguration` can also be defined through `PropertySource` and has the following properties:

Configuration Properties

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

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

12.2. Working With Redis Cluster Connection

As mentioned earlier, Redis Cluster behaves differently from single-node Redis or even a Sentinel-monitored master-replica environment. This is because the automatic sharding maps a key to one of 16384 slots, which are distributed across the nodes. Therefore, commands that involve more than one key must assert all keys map to the exact same slot to avoid cross-slot errors. A single cluster node serves only a dedicated set of keys. Commands issued against one particular server return results only for those keys served by that server. As a simple example, consider the `KEYS` command. When issued to a server in a cluster environment, it returns only 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 a cluster environment, you must read the keys from all the known master nodes.

While redirects for specific keys to the corresponding slot-serving node are handled by the driver libraries, higher-level functions, such as collecting information across nodes or sending commands to all nodes in the cluster, are covered by `RedisClusterConnection`. Picking up the keys example from earlier, this means that the `keys(pattern)` method picks up every master node in the cluster and simultaneously runs the `KEYS` command on every master node 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 methods (for example, `keys(node, pattern)`).

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

The following example shows a set of commands being run across the cluster:

Example 6. Sample of Running Commands Across the Cluster

```text
TEXT
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
```

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

```
connection.set("thing1", value);                                         5
connection.set("thing2", 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 replica at 7382

2 Master node serving slots 5461 to 10922

3 Master node serving slots 10923 to 16383

4 Replica node holding replicants of the 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 → [thing1, thing2]

8 Request routed to node at 7379 → [thing2]

9 Request routed to node at 7380 → []

10 Request routed to node at 7381 → [thing1]

11 Request routed to node at 7382 → [thing2]

When all keys map to the same slot, the native driver library automatically serves cross-slot requests, such as `MGET` . However, once this is not the case, `RedisClusterConnection` runs multiple parallel `GET` commands against the slot-serving nodes and again returns an accumulated result. This is less performant than the single-slot approach and, therefore, should be used with care. If in doubt, consider pinning keys to the same slot by providing a prefix in curly brackets, such as `{my-prefix}.thing1` and `{my-`

`prefix}.thing2` , which will both map to the same slot number. The following example shows cross-slot request handling:

Example 7. Sample of Cross-Slot Request Handling

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

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

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

connection.set("thing1", value);           // slot: 12182
connection.set("{thing1}.thing2", value);  // slot: 12182
connection.set("thing2", value);           // slot:  5461

connection.mGet("thing1", "{thing1}.thing2");                                 2

connection.mGet("thing1", "thing2");                                          3
```

1 Same Configuration as in the sample before.

2 Keys map to same slot → 127.0.0.1:7381 MGET thing1 {thing1}.thing2

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 thing2

→ 127.0.0.1:7381 GET thing1

The preceding examples demonstrate the general strategy followed by Spring Data Redis. Be aware that some operations might require loading huge amounts of data into memory to compute the desired command. Additionally, not all cross-slot requests can safely be ported to multiple single slot requests and error if misused (for example, `PFCOUNT`).

12.3. Working with `RedisTemplate` and `ClusterOperations`

See the Working with Objects through RedisTemplate section for information about the general purpose, configuration, and usage of `RedisTemplate` .

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 through the `ClusterOperations` interface, which can be obtained from `RedisTemplate.opsForCluster()` . This lets you explicitly run commands on a single node within the cluster while retaining the serialization and deserialization features configured for the template. It also provides administrative commands (such as `CLUSTER MEET`) or more high-level operations (for example, resharding).

The following example shows how to access `RedisClusterConnection` with `RedisTemplate` :

Example 8. Accessing `RedisClusterConnection` with `RedisTemplate`

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

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

13. Redis Repositories

Working with Redis Repositories lets you seamlessly convert and store domain objects in Redis Hashes, apply custom mapping strategies, and use secondary indexes.

Redis Repositories require at least Redis Server version 2.8.0 and do not work with transactions. Make sure to use a `RedisTemplate` with disabled transaction support.

13.1. Usage

Spring Data Redis lets you easily implement domain entities, as shown in the following example:

Example 9. Sample Person Entity

```java
@RedisHash("people")
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 `@RedisHash` annotation on its type and a property named `id` that is annotated with `org.springframework.data.annotation.Id`. Those two items are responsible for creating the actual key used to persist the hash.

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, as shown in the following example:

Example 10. Basic Repository Interface To Persist Person Entities

```java
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 corresponding Spring configuration, shown in the following example:

Example 11. JavaConfig for Redis Repositories

```java
@Configuration
@EnableRedisRepositories
public class ApplicationConfig {

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

  @Bean
  public RedisTemplate<?, ?> redisTemplate(RedisConnectionFactory redisConnectionFactory) {
    RedisTemplate<byte[], byte[]> template = new RedisTemplate<byte[], byte[]>();
    template.setConnectionFactory(redisConnectionFactory);
    return template;
  }
}
```

Given the preceding setup, we can inject `PersonRepository` into our components, as shown in the following example:

Example 12. Access to Person Entities

```java
@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 the current value is `null` or reuses an already set `id` value and stores properties of type `Person` inside the Redis Hash with a key that has a pattern of `keyspace:id` — in this case, it might be `people: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, `people` , defined by `@RedisHash` on `Person` .

4 Removes the key for the given object from Redis.

13.2. Object Mapping Fundamentals

This section covers the fundamentals of Spring Data object mapping, object creation, field and property access, mutability and immutability. Note, that this section only applies to Spring Data modules that do not use the object mapping of the underlying data store (like JPA). Also be sure to consult the store-specific sections for store-specific object mapping, like indexes, customizing column or field names or the like.

Core responsibility of the Spring Data object mapping is to create instances of domain objects and map the store-native data structures onto those. This means we need two fundamental steps:

1. Instance creation by using one of the constructors exposed.
2. Instance population to materialize all exposed properties.

13.2.1. Object creation

Spring Data automatically tries to detect a persistent entity’s constructor to be used to materialize objects of that type. The resolution algorithm works as follows:

1. If there is a single constructor, it is used.
2. If there are multiple constructors and exactly one is annotated with `@PersistenceConstructor`, it is used.
3. If there’s a no-argument constructor, it is used. Other constructors will be ignored.

The value resolution assumes constructor argument names to match the property names of the entity, i.e. the resolution will be performed as if the property was to be populated, including all customizations in mapping (different datastore column or field name etc.). This also requires either parameter names information available in the class file or an `@ConstructorProperties` annotation being present on the constructor.

The value resolution can be customized by using Spring Framework's `@Value` value annotation using a store-specific SpEL expression. Please consult the section on store specific mappings for further details.

Object creation internals

To avoid the overhead of reflection, Spring Data object creation uses a factory class generated at runtime by default, which will call the domain classes constructor directly. I.e. for this example type:

```java
class Person {
  Person(String firstname, String lastname) { … }
}
```

we will create a factory class semantically equivalent to this one at runtime:

```java
class PersonObjectInstantiator implements ObjectInstantiator {

  Object newInstance(Object... args) {
    return new Person((String) args[0], (String) args[1]);
  }
}
```

This gives us a roundabout 10% performance boost over reflection. For the domain class to be eligible for such optimization, it needs to adhere to a set of constraints:

- it must not be a private class
- it must not be a non-static inner class
- it must not be a CGLib proxy class

- the constructor to be used by Spring Data must not be private

If any of these criteria match, Spring Data will fall back to entity instantiation via reflection.

13.2.2. Property population

Once an instance of the entity has been created, Spring Data populates all remaining persistent properties of that class. Unless already populated by the entity's constructor (i.e. consumed through its constructor argument list), the identifier property will be populated first to allow the resolution of cyclic object references. After that, all non-transient properties that have not already been populated by the constructor are set on the entity instance. For that we use the following algorithm:

1. If the property is immutable but exposes a `with…` method (see below), we use the `with…` method to create a new entity instance with the new property value.
2. If property access (i.e. access through getters and setters) is defined, we're invoking the setter method.
3. If the property is mutable we set the field directly.
4. If the property is immutable we're using the constructor to be used by persistence operations (see Object creation) to create a copy of the instance.
5. By default, we set the field value directly.

Property population internals

Similarly to our optimizations in object construction we also use Spring Data runtime generated accessor classes to interact with the entity instance.

```java
class Person {

  private final Long id;
  private String firstname;
  private @AccessType(Type.PROPERTY) String lastname;

  Person() {
    this.id = null;
  }

  Person(Long id, String firstname, String lastname) {
    // Field assignments
  }

  Person withId(Long id) {
    return new Person(id, this.firstname, this.lastame);
  }

  void setLastname(String lastname) {
    this.lastname = lastname;
  }
}
```

Example 13. A generated Property Accessor

```java
class PersonPropertyAccessor implements PersistentPropertyAccessor {

  private static final MethodHandle firstname;              2

  private Person person;                                    1

  public void setProperty(PersistentProperty property, Object value) {

    String name = property.getName();

    if ("firstname".equals(name)) {
      firstname.invoke(person, (String) value);             2
    } else if ("id".equals(name)) {
      this.person = person.withId((Long) value);            3
    } else if ("lastname".equals(name)) {
      this.person.setLastname((String) value);              4
    }
    }
  }
```

1 PropertyAccessor's hold a mutable instance of the underlying object. This is, to enable mutations of otherwise immutable properties.

2 By default, Spring Data uses field-access to read and write property values. As per visibility rules of `private` fields, `MethodHandles` are used to interact with fields.

3 The class exposes a `withId(…)` method that's used to set the identifier, e.g. when an instance is inserted into the datastore and an identifier has been generated. Calling `withId(…)` creates a new `Person` object. All subsequent mutations will take place in the new instance leaving the previous untouched.

4 Using property-access allows direct method invocations without using `MethodHandles`.

This gives us a roundabout 25% performance boost over reflection. For the domain class to be eligible for such optimization, it needs to adhere to a set of constraints:

- Types must not reside in the default or under the `java` package.
- Types and their constructors must be `public`
- Types that are inner classes must be `static`.
- The used Java Runtime must allow for declaring classes in the originating `ClassLoader`. Java 9 and newer impose certain limitations.

By default, Spring Data attempts to use generated property accessors and falls back to reflection-based ones if a limitation is detected.

Let's have a look at the following entity:

Example 14. A sample entity

```java
class Person {

  private final @Id Long id;                                                1
  private final String firstname, lastname;                                 2
  private final LocalDate birthday;
  private final int age;                                                    3

  private String comment;                                                   4
  private @AccessType(Type.PROPERTY) String remarks;                        5

  static Person of(String firstname, String lastname, LocalDate birthday) {  6

    return new Person(null, firstname, lastname, birthday,
      Period.between(birthday, LocalDate.now()).getYears());
  }

  Person(Long id, String firstname, String lastname, LocalDate birthday, int age) {  6

    this.id = id;
    this.firstname = firstname;
    this.lastname = lastname;
    this.birthday = birthday;
    this.age = age;
  }

  Person withId(Long id) {                                                  1
    return new Person(id, this.firstname, this.lastname, this.birthday, this.age);
  }

  void setRemarks(String remarks) {                                         5
    this.remarks = remarks;
  }
}
```

1 The identifier property is final but set to `null` in the constructor.

The class exposes a `withId(…)` method that’s used to set the identifier, e.g. when an instance is inserted into the datastore and an identifier has been generated.
The original `Person` instance stays unchanged as a new one is created.
The same pattern is usually applied for other properties that are store managed but might have to be changed for persistence operations.
The wither method is optional as the persistence constructor (see 6) is effectively a copy constructor and setting the property will be translated into creating a fresh instance with the new identifier value applied.

2 The `firstname` and `lastname` properties are ordinary immutable properties potentially exposed through getters.

3 The `age` property is an immutable but derived one from the `birthday` property.
With the design shown, the database value will trump the defaulting as Spring Data uses the only declared constructor.
Even if the intent is that the calculation should be preferred, it’s important that this constructor also takes `age` as parameter (to potentially ignore it) as otherwise the property population step will attempt to set the age field and fail due to it being immutable and no `with…` method being present.

4 The `comment` property is mutable is populated by setting its field directly.

5 The `remarks` properties are mutable and populated by setting the `comment` field directly or by invoking the setter method for

6 The class exposes a factory method and a constructor for object creation.
The core idea here is to use factory methods instead of additional constructors to avoid the need for constructor disambiguation through `@PersistenceConstructor` .
Instead, defaulting of properties is handled within the factory method.

13.2.3. General recommendations

- *Try to stick to immutable objects* — Immutable objects are straightforward to create as materializing an object is then a matter of calling its constructor only. Also, this avoids your domain objects to be littered with setter methods that allow client code to manipulate the objects state. If you need those, prefer to make them package protected so that they can only be invoked by a limited amount of co-located types. Constructor-only materialization is up to 30% faster than properties population.

- *Provide an all-args constructor* — Even if you cannot or don’t want to model your entities as immutable values, there’s still value in providing a constructor that takes all properties of the entity as arguments, including the mutable ones, as this allows the object mapping to skip the property population for optimal performance.
- *Use factory methods instead of overloaded constructors to avoid `@PersistenceConstructor`* — With an all-argument constructor needed for optimal performance, we usually want to expose more application use case specific constructors that omit things like auto-generated identifiers etc. It’s an established pattern to rather use static factory methods to expose these variants of the all-args constructor.
- *Make sure you adhere to the constraints that allow the generated instantiator and property accessor classes to be used* —
- *For identifiers to be generated, still use a final field in combination with an all-arguments persistence constructor (preferred) or a `with…` method* —
- *Use Lombok to avoid boilerplate code* — As persistence operations usually require a constructor taking all arguments, their declaration becomes a tedious repetition of boilerplate parameter to field assignments that can best be avoided by using Lombok’s `@AllArgsConstructor`.

Overriding Properties

Java’s allows a flexible design of domain classes where a subclass could define a property that is already declared with the same name in its superclass. Consider the following example:

```java
public class SuperType {

   private CharSequence field;

   public SuperType(CharSequence field) {
      this.field = field;
   }

   public CharSequence getField() {
      return this.field;
```

```
    }

    public void setField(CharSequence field) {
       this.field = field;
    }
 }

 public class SubType extends SuperType {

    private String field;

    public SubType(String field) {
       super(field);
       this.field = field;
    }

    @Override
    public String getField() {
       return this.field;
    }

    public void setField(String field) {
       this.field = field;

       // optional
       super.setField(field);
    }
 }
```

Both classes define a `field` using assignable types. `SubType` however shadows `SuperType.field`. Depending on the class design, using the constructor could be the only default approach to set `SuperType.field`. Alternatively, calling `super.setField(…)` in the setter could set the `field` in `SuperType`. All these mechanisms create conflicts to some degree because the properties share the same name yet might represent two distinct values. Spring Data skips super-type properties if types are not assignable. That is, the type of the overridden property must be assignable to its super-type property type to be registered as override, otherwise the super-type property is considered transient. We generally recommend using distinct property names.

Spring Data modules generally support overridden properties holding different values. From a programming model perspective there are a few things to consider:

1. Which property should be persisted (default to all declared properties)? You can exclude properties by annotating these with `@Transient`.
2. How to represent properties in your data store? Using the same field/column name for different values typically leads to corrupt data so you should annotate least one of the properties using an explicit field/column name.
3. Using `@AccessType(PROPERTY)` cannot be used as the super-property cannot be generally set without making any further assumptions of the setter implementation.

13.2.4. Kotlin support

Spring Data adapts specifics of Kotlin to allow object creation and mutation.

Kotlin object creation

Kotlin classes are supported to be instantiated , all classes are immutable by default and require explicit property declarations to define mutable properties. Consider the following `data` class `Person`:

```kotlin
data class Person(val id: String, val name: String)
```

The class above compiles to a typical class with an explicit constructor.We can customize this class by adding another constructor and annotate it with `@PersistenceConstructor` to indicate a constructor preference:

```kotlin
data class Person(var id: String, val name: String) {
    @PersistenceConstructor
    constructor(id: String) : this(id, "unknown")
}
```

Kotlin supports parameter optionality by allowing default values to be used if a parameter is not provided. When Spring Data detects a constructor with parameter defaulting, then it leaves these parameters absent if the data store does not provide a value (or simply returns `null`) so Kotlin can apply parameter defaulting.Consider the following class that applies parameter defaulting for `name`

```kotlin
data class Person(var id: String, val name: String = "unknown")
```

Every time the `name` parameter is either not part of the result or its value is `null` , then the `name` defaults to `unknown` .

Property population of Kotlin data classes

In Kotlin, all classes are immutable by default and require explicit property declarations to define mutable properties. Consider the following `data` class `Person` :

```kotlin
data class Person(val id: String, val name: String)
```

This class is effectively immutable. It allows creating new instances as Kotlin generates a `copy(…)` method that creates new object instances copying all property values from the existing object and applying property values provided as arguments to the method.

Kotlin Overriding Properties

Kotlin allows declaring property overrides to alter properties in subclasses.

```kotlin
open class SuperType(open var field: Int)

class SubType(override var field: Int = 1) :
    SuperType(field) {
}
```

Such an arrangement renders two properties with the name `field`. Kotlin generates property accessors (getters and setters) for each property in each class. Effectively, the code looks like as follows:

```java
public class SuperType {

   private int field;

   public SuperType(int field) {
      this.field = field;
   }

   public int getField() {
      return this.field;
   }

   public void setField(int field) {
      this.field = field;
   }
}

public final class SubType extends SuperType {

   private int field;

   public SubType(int field) {
      super(field);
```

```
        this.field = field;
    }

    public int getField() {
        return this.field;
    }

    public void setField(int field) {
        this.field = field;
    }
}
```

Getters and setters on `SubType` set only `SubType.field` and not `SuperType.field`. In such an arrangement, using the constructor is the only default approach to set `SuperType.field`. Adding a method to `SubType` to set `SuperType.field` via `this.SuperType.field = …` is possible but falls outside of supported conventions. Property overrides create conflicts to some degree because the properties share the same name yet might represent two distinct values. We generally recommend using distinct property names.

Spring Data modules generally support overridden properties holding different values. From a programming model perspective there are a few things to consider:

1. Which property should be persisted (default to all declared properties)? You can exclude properties by annotating these with `@Transient`.
2. How to represent properties in your data store? Using the same field/column name for different values typically leads to corrupt data so you should annotate least one of the properties using an explicit field/column name.
3. Using `@AccessType(PROPERTY)` cannot be used as the super-property cannot be set.

13.3. Object-to-Hash Mapping

The Redis Repository support persists Objects to 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:

TEXT
```
_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 the 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.

The following table describes the default mapping rules:

Table 10. Default Mapping Rules

Type	Sample	Mapped Value
Simple Type (for example, String)	String firstname = "rand";	firstname = "rand"
Byte array (`byte[]`)	byte[] image = "rand".getBytes();	image = "rand"
Complex Type (for example,	Address address = new Address("emond’s field");	address.city = "emond’s field"

Type	Sample	Mapped Value
Address)		

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 = "…

Due to the flat representation structure, Map keys need to be simple types, such as `String` or `Number` .

Mapping behavior can be customized by registering the corresponding `Converter` in `RedisCustomConversions` . Those converters can take care of converting from and to a single `byte[]` as well as `Map<String,byte[]>` . The first one is suitable for (for example) converting a complex type to (for example) 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 deletes the content from the hash and re-creates the whole hash, so data that has not been mapped is lost.

The following example shows two sample byte array converters:

Example 15. Sample byte[] Converters

JAVA

```java
@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 preceding byte array `Converter` produces output similar to the following:

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

The following example shows two examples of `Map` converters:

Example 16. Sample Map<String,byte[]> Converters

```java
@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<Map<String, byte[]>, Address> {

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

Using the preceding Map `Converter` produces output similar to the following:

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

Custom conversions have no effect on index resolution. Secondary Indexes are still created, even for custom converted types.

13.3.1. Customizing Type Mapping

If you want to avoid writing the entire Java class name as type information and would rather like to use a key, you can use the `@TypeAlias` annotation on the entity class being persisted. If you need to customize the mapping even more, look at the `TypeInformationMapper` interface. An instance of that interface can be configured at the `DefaultRedisTypeMapper`, which can be configured on `MappingRedisConverter`.

The following example shows how to define a type alias for an entity:

Example 17. Defining `@TypeAlias` for an entity

```java
@TypeAlias("pers")
class Person {

}
```

The resulting document contains `pers` as the value in a `_class` field.

Configuring Custom Type Mapping

The following example demonstrates how to configure a custom `RedisTypeMapper` in `MappingRedisConverter`:

Example 18. Configuring a custom `RedisTypeMapper` via Spring Java Config

JAVA
```java
class CustomRedisTypeMapper extends DefaultRedisTypeMapper {
  //implement custom type mapping here
}
```

JAVA
```java
@Configuration
class SampleRedisConfiguration {

  @Bean
  public MappingRedisConverter redisConverter(RedisMappingContext mappingContext,
        RedisCustomConversions customConversions, ReferenceResolver referenceResolver) {

    MappingRedisConverter mappingRedisConverter = new MappingRedisConverter(mappingContext, null, referenceResolver,
            customTypeMapper());

    mappingRedisConverter.setCustomConversions(customConversions);

    return mappingRedisConverter;
  }

  @Bean
  public RedisTypeMapper customTypeMapper() {
    return new CustomRedisTypeMapper();
  }
}
```

13.4. Keyspaces

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

The following example shows how to set the keyspace configuration with the `@EnableRedisRepositories` annotation:

Example 19. Keyspace Setup via `@EnableRedisRepositories`

JAVA

```java
@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, "people"));
    }
  }
}
```

The following example shows how to programmatically set the keyspace:

Example 20. Programmatic Keyspace setup

JAVA

```java
@Configuration
@EnableRedisRepositories
public class ApplicationConfig {
  //... RedisConnectionFactory and RedisTemplate Bean definitions omitted

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

  public static class MyKeyspaceConfiguration extends KeyspaceConfiguration {

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

13.5. 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.

13.5.1. Simple Property Index

Given the sample `Person` entity shown earlier, we can create an index for `firstname` by annotating the property with `@Indexed` , as shown in the following example:

Example 21. Annotation driven indexing

```java
@RedisHash("people")
public class Person {
  @Id String id;
  @Indexed String firstname;
  String lastname;
  Address address;
}
```

Indexes are built up for actual property values. Saving two Persons (for example, "rand" and "aviendha") results in setting up indexes similar to the following:

```text
SADD people:firstname:rand e2c7dcee-b8cd-4424-883e-736ce564363e
SADD people: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, as shown in the following example:

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

Furthermore, the programmatic setup lets you define indexes on map keys and list properties, as shown in the following example:

```java
@RedisHash("people")
public class Person {

  // ... other properties omitted

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

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

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

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

Indexes cannot be resolved on References.

As with keyspaces, you can configure indexes without needing to annotate the actual domain type, as shown in the following example:

Example 22. Index Setup with @EnableRedisRepositories

```java
@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("people", "firstname"));
    }
  }
}
```

Again, as with keyspaces, you can programmatically configure indexes, as shown in the following example:

Example 23. Programmatic Index setup

```java
@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("people", "firstname"));
    }
  }
}
```

13.5.2. Geospatial Index

Assume the `Address` type contains a `location` property of type `Point` that holds the geo coordinates of the particular address. By annotating the property with `@GeoIndexed`, Spring Data Redis adds those values by using Redis `GEO` commands, as shown in the following example:

```java
@RedisHash("people")
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 a nested property, using `Point` and `Distance`.

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

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

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

In the preceding example the, longitude and latitude values are stored by using `GEOADD` that use the object's `id` as the member's name. The finder methods allow usage of `Circle` or `Point, Distance` combinations for querying those values.

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

13.6. Query by Example

13.6.1. Introduction

This chapter provides an introduction to Query by Example and explains how to use it.

Query by Example (QBE) is a user-friendly querying technique with a simple interface. It allows dynamic query creation and does not require you to write queries that contain field names. In fact, Query by Example does not require you to write queries by using store-specific query languages at all.

13.6.2. Usage

The Query by Example API consists of three parts:

- Probe: The actual example of a domain object with populated fields.
- `ExampleMatcher` : The `ExampleMatcher` carries details on how to match particular fields. It can be reused across multiple Examples.
- `Example` : An `Example` consists of the probe and the `ExampleMatcher` . It is used to create the query.

Query by Example is well suited for several use cases:

- Querying your data store with a set of static or dynamic constraints.
- Frequent refactoring of the domain objects without worrying about breaking existing queries.
- Working independently from the underlying data store API.

Query by Example also has several limitations:

- No support for nested or grouped property constraints, such as `firstname = ?0 or (firstname = ?1 and lastname = ?2)`.
- Only supports starts/contains/ends/regex matching for strings and exact matching for other property types.

Before getting started with Query by Example, you need to have a domain object. To get started, create an interface for your repository, as shown in the following example:

Example 24. Sample Person object

```java
public class Person {

  @Id
  private String id;
  private String firstname;
  private String lastname;
  private Address address;

  // … getters and setters omitted
}
```

The preceding example shows a simple domain object. You can use it to create an `Example`. By default, fields having `null` values are ignored, and strings are matched by using the store specific defaults.

Inclusion of properties into a Query by Example criteria is based on nullability. Properties using primitive types (`int` , `double` , …) are always included unless ignoring the property path.

Examples can be built by either using the `of` factory method or by using `ExampleMatcher` . `Example` is immutable. The following listing shows a simple Example:

Example 25. Simple Example

```java
Person person = new Person();                          1
person.setFirstname("Dave");                           2

Example<Person> example = Example.of(person);          3
```

1 Create a new instance of the domain object.

2 Set the properties to query.

3 Create the `Example` .

You can run the example queries by using repositories. To do so, let your repository interface extend `QueryByExampleExecutor<T>` . The following listing shows an excerpt from the `QueryByExampleExecutor` interface:

Example 26. The `QueryByExampleExecutor`

```java
public interface QueryByExampleExecutor<T> {

  <S extends T> S findOne(Example<S> example);

  <S extends T> Iterable<S> findAll(Example<S> example);

  // … more functionality omitted.
}
```

13.6.3. Example Matchers

Examples are not limited to default settings. You can specify your own defaults for string matching, null handling, and property-specific settings by using the `ExampleMatcher`, as shown in the following example:

Example 27. Example matcher with customized matching

```java
Person person = new Person();                          1
person.setFirstname("Dave");                           2

ExampleMatcher matcher = ExampleMatcher.matching()     3
  .withIgnorePaths("lastname")                         4
  .withIncludeNullValues()                             5
  .withStringMatcher(StringMatcher.ENDING);            6

Example<Person> example = Example.of(person, matcher); 7
```

1. Create a new instance of the domain object.
2. Set properties.
3. Create an `ExampleMatcher` to expect all values to match.

It is usable at this stage even without further configuration.

4 Construct a new `ExampleMatcher` to ignore the `lastname` property path.

5 Construct a new `ExampleMatcher` to ignore the `lastname` property path and to include null values.

6 Construct a new `ExampleMatcher` to ignore the `lastname` property path, to include null values, and to perform suffix string matching.

7 Create a new `Example` based on the domain object and the configured `ExampleMatcher`.

By default, the `ExampleMatcher` expects all values set on the probe to match. If you want to get results matching any of the predicates defined implicitly, use `ExampleMatcher.matchingAny()`.

You can specify behavior for individual properties (such as "firstname" and "lastname" or, for nested properties, "address.city"). You can tune it with matching options and case sensitivity, as shown in the following example:

Example 28. Configuring matcher options

JAVA
```java
ExampleMatcher matcher = ExampleMatcher.matching()
  .withMatcher("firstname", endsWith())
  .withMatcher("lastname", startsWith().ignoreCase());
}
```

Another way to configure matcher options is to use lambdas (introduced in Java 8). This approach creates a callback that asks the implementor to modify the matcher. You need not return the matcher, because configuration options are held within the matcher instance. The following example shows a matcher that uses lambdas:

Example 29. Configuring matcher options with lambdas

```java
ExampleMatcher matcher = ExampleMatcher.matching()
  .withMatcher("firstname", match -> match.endsWith())
  .withMatcher("firstname", match -> match.startsWith());
}
```

Queries created by `Example` use a merged view of the configuration. Default matching settings can be set at the `ExampleMatcher` level, while individual settings can be applied to particular property paths. Settings that are set on `ExampleMatcher` are inherited by property path settings unless they are defined explicitly. Settings on a property patch have higher precedence than default settings. The following table describes the scope of the various `ExampleMatcher` settings:

Table 11. Scope of `ExampleMatcher` settings

Setting	Scope
Null-handling	`ExampleMatcher`
String matching	`ExampleMatcher` and property path
Ignoring properties	Property path
Case sensitivity	`ExampleMatcher` and property path
Value transformation	Property path

13.6.4. Running an Example

The following example uses Query by Example against a repository:

Example 30. Query by Example using a Repository

```java
interface PersonRepository extends QueryByExampleExecutor<Person> {
}

class PersonService {

  @Autowired PersonRepository personRepository;

  List<Person> findPeople(Person probe) {
    return personRepository.findAll(Example.of(probe));
  }
}
```

Redis Repositories support, with their secondary indexes, a subset of Spring Data's Query by Example features. In particular, only exact, case-sensitive, and non-null values are used to construct a query.

Secondary indexes use set-based operations (Set intersection, Set union) to determine matching keys. Adding a property to the query that is not indexed returns no result, because no index exists. Query by Example support inspects indexing configuration to include only properties in the query that are covered by an index. This is to prevent accidental inclusion of non-indexed properties.

Case-insensitive queries and unsupported `StringMatcher` instances are rejected at runtime.

The following list shows the supported Query by Example options:

- Case-sensitive, exact matching of simple and nested properties
- Any/All match modes
- Value transformation of the criteria value
- Exclusion of `null` values from the criteria

The following list shows properties not supported by Query by Example:

- Case-insensitive matching
- Regex, prefix/contains/suffix String-matching
- Querying of Associations, Collection, and Map-like properties
- Inclusion of `null` values from the criteria
- `findAll` with sorting

13.7. Time To Live

Objects stored in Redis may be valid only 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 reach their end of life. The expiration time in seconds can be set with `@RedisHash(timeToLive=…)` as well as by using `KeyspaceSettings` (see Keyspaces).

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

Example 31. Expirations

```java
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();
  }
}
```

Annotating a property explicitly with `@TimeToLive` reads back the actual `TTL` or `PTTL` value from Redis. -1 indicates that the object has no associated expiration.

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

When the expiration is set to a positive value, the corresponding `EXPIRE` command is run. In addition to persisting the original, a phantom copy is persisted in Redis and set to expire five minutes after the original one. This is done to enable the Repository support to publish `RedisKeyExpiredEvent` , holding the expired value in Spring's `ApplicationEventPublisher` whenever a key expires, even though the original values have already been removed. Expiry events are received on all connected applications that use 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` holds a copy of the expired domain object as well as the key.

Delaying or disabling the expiry event listener startup impacts `RedisKeyExpiredEvent` publishing. A disabled event listener does not publish expiry events. A delayed startup can cause loss of events because of the delayed listener initialization.

The keyspace notification message listener alters `notify-keyspace-events` settings in Redis, if those are not already set. Existing settings are not overridden, so you must set up those settings correctly (or leave them empty). Note that `CONFIG` is disabled on AWS ElastiCache, and enabling the listener leads to an error. To work around this behavior, set the `keyspaceNotificationsConfigParameter` parameter to an empty string. This prevents `CONFIG` command usage.

Redis Pub/Sub messages are not persistent. If a key expires while the application is down, the expiry event is not processed, which may lead to secondary indexes containing references to the expired object.

`@EnableKeyspaceEvents(shadowCopy = OFF)` disable storage of phantom copies and reduces data size within Redis. `RedisKeyExpiredEvent` will only contain the `id` of the expired key.

13.8. 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, as shown in the following example:

Example 32. Sample Property Reference

```
TEXT
_class = org.example.Person
id = e2c7dcee-b8cd-4424-883e-736ce564363e
```

```
firstname = rand
lastname = al’thor
mother = people:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56      1
```

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

Referenced Objects are not persisted when the referencing object is saved. You must persist changes on referenced objects separately, since only the reference is stored. Indexes set on properties of referenced types are not resolved.

13.9. Persisting Partial Updates

In some cases, you need not load and rewrite the entire entity just to set a new value within it. A session timestamp for the last active time might be such a scenario where you want to alter one property. `PartialUpdate` lets you define `set` and `delete` actions on existing objects while taking care of updating potential expiration times of both the entity itself and index structures. The following example shows a partial update:

Example 33. Sample Partial Update

```java
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 `firstname` property to `mat`.

2 Set the simple 'address.city' property 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 `age` property.

4 Set complex `address` property.

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

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

Updating complex objects as well as map (or other collection) structures requires further interaction with Redis to determine existing values, which means that rewriting the entire entity might be faster.

13.10. Queries and Query Methods

Query methods allow automatic derivation of simple finder queries from the method name, as shown in the following example:

Example 34. Sample Repository finder Method

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

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

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

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 run. `RedisCallback` offers more control over the actual matching of index structures or even custom indexes. To do so, provide a `RedisCallback` that returns a single or `Iterable` set of `id` values, as shown in the following example:

Example 35. Sample finder using RedisCallback

```java
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);
```

The following table provides an overview of the keywords supported for Redis and what a method containing that keyword essentially translates to:

Table 12. 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`
`IsTrue`	`FindByAliveIsTrue`	`SINTER …:alive:1`
`IsFalse`	`findByAliveIsFalse`	`SINTER …:alive:0`
`Top,First`	`findFirst10ByFirstname` , `findTop5ByFirstname`	

13.10.1. Sorting Query Method results

Redis repositories allow various approaches to define sorting order. Redis itself does not support in-flight sorting when retrieving hashes or sets. Therefore, Redis repository query methods construct a `Comparator` that is applied to the result before returning results as `List` . Let’s take a look at the following example:

Example 36. Sorting Query Results

```java
interface PersonRepository extends RedisRepository<Person, String> {
  List<Person> findByFirstnameOrderByAgeDesc(String firstname);  1

  List<Person> findByFirstname(String firstname, Sort sort);    2
}
```

1 Static sorting derived from method name.

2 Dynamic sorting using a method argument.

13.11. Redis Repositories Running on a Cluster

You can use the Redis repository support in a clustered Redis environment. See the "Redis Cluster" section for `ConnectionFactory` configuration details. Still, some additional configuration must be done, because the default key distribution spreads entities and secondary indexes through out the whole cluster and its slots.

The following table shows the details of data on a cluster (based on previous examples):

Key	Type	Slot	Node
people:e2c7dcee-b8cd-4424-883e-736ce564363e	id for hash	15171	127.0.0.1:7381
people:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56	id for hash	7373	127.0.0.1:7380
people:firstname:rand	index	1700	127.0.0.1:7379

Some commands (such as `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 is useful to pin keyspaces to a single slot, which lets make use of Redis server side computation right away. The following table shows what happens when you do (note the change in the slot column and the port value in the node column):

Key	Type	Slot	Node
{people}:e2c7dcee-b8cd-4424-883e-736ce564363e	id for hash	2399	127.0.0.1:7379
{people}:a9d4b3a0-50d3-4538-a2fc-f7fc2581ee56	id for hash	2399	127.0.0.1:7379
{people}:firstname:rand	index	2399	127.0.0.1:7379

Define and pin keyspaces by using `@RedisHash("{yourkeyspace}")` to specific slots when you use Redis cluster.

13.12. CDI Integration

Instances of the repository interfaces are usually created by a container, for which Spring is the most natural choice when working with Spring Data. Spring offers sophisticated for creating bean instances. Spring Data Redis ships with a custom CDI extension that lets you use the repository abstraction in CDI environments. The extension is part of the JAR, so, to activate it, drop the Spring Data Redis JAR into your classpath.

You can then set up the infrastructure by implementing a CDI Producer for the `RedisConnectionFactory` and `RedisOperations` , as shown in the following example:

```java
JAVA
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 your JavaEE environment.

The Spring Data Redis CDI extension picks up all available repositories as CDI beans and creates 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, as shown in the following example:

```java
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`.

13.13. Redis Repositories Anatomy

Redis as a store itself offers a very narrow low-level API leaving higher level functions, such as secondary indexes and query operations, up to the user.

This section provides a more detailed view of commands issued by the repository abstraction for a better understanding of potential performance implications.

Consider the following entity class as the starting point for all operations:

Example 37. Example entity

```java
@RedisHash("people")
public class Person {

  @Id String id;
  @Indexed String firstname;
  String lastname;
  Address hometown;
}

public class Address {

  @GeoIndexed Point location;
}
```

13.13.1. Insert new

```java
repository.save(new Person("rand", "al'thor"));
```

```text
HMSET "people:19315449-cda2-4f5c-b696-9cb8018fa1f9" "_class" "Person" "id" "19315449-cda2-4f5c-b696-9cb8018fa1f9" "firstname" '
SADD  "people" "19315449-cda2-4f5c-b696-9cb8018fa1f9"                           2
SADD  "people:firstname:rand" "19315449-cda2-4f5c-b696-9cb8018fa1f9"            3
SADD  "people:19315449-cda2-4f5c-b696-9cb8018fa1f9:idx" "people:firstname:rand"  4
```

1 Save the flattened entry as hash.

2 Add the key of the hash written in <1> to the helper index of entities in the same keyspace.

3 Add the key of the hash written in <2> to the secondary index of firstnames with the properties value.

4 Add the index of <3> to the set of helper structures for entry to keep track of indexes to clean on delete/update.

13.13.2. Replace existing

```java
repository.save(new Person("e82908cf-e7d3-47c2-9eec-b4e0967ad0c9", "Dragon Reborn", "al'thor"));
```

```text
DEL       "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"                            1
HMSET     "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9" "_class" "Person" "id" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9" "firstnam
SADD      "people" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"                          3
SMEMBERS  "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9:idx"                        4
```

```
TYPE      "people:firstname:rand"                                           5
SREM      "people:firstname:rand" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"    6
DEL       "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9:idx"                 7
SADD      "people:firstname:Dragon Reborn" "e82908cf-e7d3-47c2-9eec-b4e0967ad0c9"  8
SADD      "people:e82908cf-e7d3-47c2-9eec-b4e0967ad0c9:idx" "people:firstname:Dragon Reborn"  9
```

1 Remove the existing hash to avoid leftovers of hash keys potentially no longer present.

2 Save the flattened entry as hash.

3 Add the key of the hash written in <1> to the helper index of entities in the same keyspace.

4 Get existing index structures that might need to be updated.

5 Check if the index exists and what type it is (text, geo, …).

6 Remove a potentially existing key from the index.

7 Remove the helper holding index information.

8 Add the key of the hash added in <2> to the secondary index of firstnames with the properties value.

9 Add the index of <6> to the set of helper structures for entry to keep track of indexes to clean on delete/update.

13.13.3. Save Geo Data

Geo indexes follow the same rules as normal text based ones but use geo structure to store values. Saving an entity that uses a Geo-indexed property results in the following commands:

TEXT
```
GEOADD "people:hometown:location" "13.361389" "38.115556" "76900e94-b057-44bc-abcf-8126d51a621b"   1
SADD   "people:76900e94-b057-44bc-abcf-8126d51a621b:idx" "people:hometown:location"                 2
```

1 Add the key of the saved entry to the the geo index.

2 Keep track of the index structure.

13.13.4. Find using simple index

```java
repository.findByFirstname("egwene");
```

```text
SINTER  "people:firstname:egwene"                     1
HGETALL "people:d70091b5-0b9a-4c0a-9551-519e61bc9ef3"  2
HGETALL ...
```

1 Fetch keys contained in the secondary index.

2 Fetch each key returned by <1> individually.

13.13.5. Find using Geo Index

```java
repository.findByHometownLocationNear(new Point(15, 37), new Distance(200, KILOMETERS));
```

```text
GEORADIUS "people:hometown:location" "15.0" "37.0" "200.0" "km"  1
HGETALL   "people:76900e94-b057-44bc-abcf-8126d51a621b"          2
HGETALL   ...
```

1 Fetch keys contained in the secondary index.

2 Fetch each key returned by <1> individually.

Appendixes

Appendix Document Structure

The appendix contains various additional detail that complements the information in the rest of the reference documentation:

- “Schema” defines the schemas provided by Spring Data Redis.
- “Command Reference” details which commands are supported by `RedisTemplate`.

Appendix A: Schema

Spring Data Redis Schema (redis-namespace)

Appendix B: Command Reference

Supported Commands

Table 13. Redis commands supported by `RedisTemplate`

Command	Template Support
APPEND	X
AUTH	X
BGREWRITEAOF	X
BGSAVE	X
BITCOUNT	X
BITFIELD	X
BITOP	X
BLPOP	X
BRPOP	X
BRPOPLPUSH	X
CLIENT KILL	X

Command	Template Support
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	-

Command	Template Support
DECR	X
DECRBY	X
DEL	X
DISCARD	X
DUMP	X
ECHO	X
EVAL	X
EVALSHA	X
EXEC	X
EXISTS	X
EXPIRE	X
EXPIREAT	X
FLUSHALL	X
FLUSHDB	X
GEOADD	X

Command	Template Support
GEODIST	X
GEOHASH	X
GEOPOS	X
GEORADIUS	X
GEORADIUSBYMEMBER	X
GEOSEARCH	X
GEOSEARCHSTORE	X
GET	X
GETBIT	X
GETRANGE	X
GETSET	X
HDEL	X
HEXISTS	X
HGET	X
HGETALL	X

Command	Template Support
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

Command	Template Support
LASTSAVE	X
LINDEX	X
LINSERT	X
LLEN	X
LPOP	X
LPUSH	X
LPUSHX	X
LRANGE	X
LREM	X
LSET	X
LTRIM	X
MGET	X
MIGRATE	-
MONITOR	-
MOVE	X

Command	Template Support
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

Command	Template Support
PUBSUB	-
PUBSUBSCRIBE	-
QUIT	X
RANDOMKEY	X
RENAME	X
RENAMENX	X
RESTORE	X
ROLE	-
RPOP	X
RPOPLPUSH	X
RPUSH	X
RPUSHX	X
SADD	X
SAVE	X
SCAN	X

Command	Template Support
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	-

Command	Template Support
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
SLOWLOG	-
SMEMBERS	X
SMOVE	X

Command	Template Support
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

Command	Template Support
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

Command	Template Support
ZREVRANGE	X
ZREVRANGEBYSCORE	X
ZREVRANK	X
ZSCAN	X
ZSCORE	X
ZUNINONSTORE	X

Version 2.6.0
Last updated 2021-11-12 11:05:07 +0100