

Spring Data Redis

Copyright ©

Table of Contents

..... iii
I. Preface 1
II. New Features 2
1. New in Spring Data Redis 1.5 3
III. Introduction 4
2. Why Spring Data Redis? 5
3. Requirements 6
4. Getting Started 7
4.1. First Steps 7
Knowing Spring 7
Knowing NoSQL and Key Value stores 7
Trying Out The Samples 7
4.2. Need Help? 7
Community Support 7
Professional Support 8
4.3. Following Development 8
IV. Reference Documentation 9
5. Redis support 10
5.1. Redis Requirements 10
5.2. Redis Support High Level View 10
5.3. Connecting to Redis 10
RedisConnection and RedisConnectionFactory 10
Configuring Jedis connector 11
Configuring JRedis connector 11
Configuring SRP connector 12
Configuring Lettuce connector 12
5.4. Redis Sentinel Support 13
5.5. Working with Objects through RedisTemplate 14
5.6. String-focused convenience classes 15
5.7. Serializers 16
5.8. Redis Messaging/PubSub 16
Sending/Publishing messages 17
Receiving/Subscribing for messages 17
Message Listener Containers 17
The MessageListenerAdapter 18
5.9. Redis Transactions 19
@Transactional Support 20
5.10. Pipelining 21
5.11. Redis Scripting 22
5.12. Support Classes 23
Support for Spring Cache Abstraction 24
5.13. Roadmap ahead 24
V. Appendixes 25
A. Schema 26
A.1. Core schema 26
B. Command Reference 27
B.1. Supported commands 27

© 2011-2014 The original authors.

Note

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Part I. Preface

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

Part II. New Features

1. 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: Section 5.4, “Redis Sentinel Support”).

Part III. Introduction

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

For an introduction to key value stores or Spring, or Spring Data examples, please refer to Chapter 4, *Getting Started* - this documentation refers only to Spring Data Redis Support and assumes the user is familiar with the key value storages and Spring concepts.

2. Why Spring Data Redis?

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

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

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

3. Requirements

Spring Data Redis 1.2.x binaries requires JDK level 6.0 and above, and Spring Framework 3.2.8 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 2.6 and 2.8 releases.

4. Getting Started

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

4.1 First Steps

As explained in Chapter 2, *Why Spring Data Redis?*, Spring Data Redis (SDR) provides integration between Spring framework and the Redis key value store. Thus, it is important to become acquainted with both of these frameworks (storages or environments depending on how you want to name them). Throughout the SDR documentation, each section provides links to resources relevant however, it is best to become familiar with these topics beforehand.

Knowing Spring

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

Knowing NoSQL and Key Value stores

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

Trying Out The Samples

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

4.2 Need Help?

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

Community Support

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

Professional Support

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

4.3 Following Development

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

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

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

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

Lastly, you can follow the Spring blog or the project team (Thomas and Christoph) on Twitter.

Part IV. Reference Documentation

Document structure

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

Chapter 5, *Redis support* introduces the Redis module feature set.

5. Redis support

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

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

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

5.1 Redis Requirements

Spring Redis requires Redis 2.6 or above and Java SE 6.0 or above . In terms of language bindings (or connectors), Spring Redis integrates with Jedis, JRedis, SRP and Lettuce, four popular open source Java libraries for Redis. If you are aware of any other connector that we should be integrating with please send us feedback.

5.2 Redis Support High Level View

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

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

5.3 Connecting to Redis

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

RedisConnection and RedisConnectionFactory

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

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

Active `RedisConnection s` are created through `RedisConnectionFactory`. In addition, the factories act as `PersistenceExceptionTranslator s`, meaning once declared, they allow one

to do transparent exception translation. For example, exception translation through the use of the `@Repository` annotation and AOP. For more information see the dedicated section in Spring Framework documentation.

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

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

> **Important**
>
> Unfortunately, currently, not all connectors support all Redis features. When invoking a method on the Connection` API that is unsupported by the underlying library, an `UnsupportedOperationException is thrown. This situation is likely to be fixed in the future, as the various connectors mature.

Configuring Jedis connector

Jedis is one of the connectors supported by the Spring Data Redis module through the `org.springframework.data.redis.connection.jedis` package. In its simplest form, the Jedis configuration looks as follow:

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

  <!-- Jedis ConnectionFactory -->
  <bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory"/
>

</beans>
```

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

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

  <bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory" p:host-
name="server" p:port="6379" />

</beans>
```

Configuring JRedis connector

JRedis is another popular, open-source connector supported by Spring Data Redis through the `org.springframework.data.redis.connection.jredis` package.

A typical JRedis configuration can looks like this:

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

  <bean id="jredisConnectionFactory" class="org.springframework.data.redis.connection.jredis.JredisConnectionFactory" p:hos
name="server" p:port="6379"/>

</beans>
```

The configuration is quite similar to Jedis, with one notable exception. By default, the `JredisConnectionFactory` pools connections. In order to use a connection pool with JRedis, configure the `JredisConnectionFactory` with an instance of `JredisPool`. For example:

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

  <bean id="jredisConnectionFactory" class="org.springframework.data.redis.connection.jredis.JredisConnectionFactory">
    <constructor-arg>
      <bean class="org.springframework.data.redis.connection.jredis.DefaultJredisPool">
        <constructor-arg value="localhost" />
        <constructor-arg value="6379" />
      </bean>
    </constructor-arg>
  </bean>

</beans>
```

Configuring SRP connector

SRP (an acronym for Sam's Redis Protocol) is the third open-source connector supported by Spring Data Redis through the `org.springframework.data.redis.connection.srp` package.

By now, its configuration is probably easy to guess:

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

  <bean id="srpConnectionFactory" class="org.springframework.data.redis.connection.srp.SrpConnectionFactory" p:host-
name="server" p:port="6379"/>

</beans>
```

Needless to say, the configuration is quite similar to that of the other connectors.

Configuring Lettuce connector

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

Its configuration is probably easy to guess:

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

  <bean id="lettuceConnectionFactory" class="org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory" p:
name="server" p:port="6379"/>

</beans>
```

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

5.4 Redis Sentinel Support

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

> **Note**
>
> Please note that currently only Jedis and lettuce Lettuce support Redis Sentinel.

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

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

> **Tip**
>
> `RedisSentinelConfiguration` can also be defined via `PropertySource`.
> *Configuration Properties*
>
> - `spring.redis.sentinel.master`: name of the master node.
> - `spring.redis.sentinel.nodes`: Comma delimited list of host:port pairs.

Sometimes direct interaction with the one of the Sentinels is required. Using `RedisConnectionFactory.getSentinelConnection()` or

`RedisConnection.getSentinelCommands()` gives you access to the first active Sentinel configured.

5.5 Working with Objects through RedisTemplate

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

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

Table 5.1. Operational views

Interface	Description
Key Type Operations	
ValueOperations	Redis string (or value) operations
ListOperations	Redis list operations
SetOperations	Redis set operations
ZSetOperations	Redis zset (or sorted set) operations
HashOperations	Redis hash operations
HyperLogLogOperations	Redis HyperLogLog operations like (pfadd, pfcount, …)
Key Bound Operations	
BoundValueOperations	Redis string (or value) key bound operations
BoundListOperations	Redis list key bound operations
BoundSetOperations	Redis set key bound operations
BoundZSetOperations	Redis zset (or sorted set) key bound operations
BoundHashOperations	Redis hash key bound operations

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

Out of the box, `RedisTemplate` uses a Java-based serializer for most of its operations. This means that any object written or read by the template will be serializer/deserialized through Java. The serialization mechanism can be easily changed on the template, and the Redis module offers several implementations available in the `org.springframework.data.redis.serializer` package - see Section 5.7, “Serializers” for more information. You can also set any of the serializers to null and use RedisTemplate with raw `byte` arrays by setting the `enableDefaultSerializer` property to false. Note that the template requires all keys to be non-null - values can be null as long as the underlying serializer accepts them; read the javadoc of each serializer for more information.

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

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

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

</beans>
```

```
public class Example {

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

  @Resource(name="redisTemplate")
  private ListOperations<String, String> listOps;

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

5.6 String-focused convenience classes

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

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

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

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

```
public class Example {

  @Autowired
  private StringRedisTemplate redisTemplate;

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

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

```
public void useCallback() {

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

5.7 Serializers

From the framework perspective, the data stored in Redis is just bytes. While Redis itself supports various types, for the most part these refer to the way the data is stored rather then what it represents. It is up to the user to decide whether the information gets translated into Strings or any other objects. The conversion between the user (custom) types and raw data (and vice-versa) is handled in Spring Data Redis through the `RedisSerializer` interface (package `org.springframework.data.redis.serializer`) which as the name implies, takes care of the serialization process. Multiple implementations are available out of the box, two of which have been already mentioned before in this documentation: the `StringRedisSerializer` and the `JdkSerializationRedisSerializer`. However one can use `OxmSerializer` for Object/XML mapping through Spring 3 OXM support or either `JacksonJsonRedisSerializer` or `Jackson2JsonRedisSerializer` for storing data in JSON format. Do note that the storage format is not limited only to values - it can be used for keys, values or hashes without any restrictions.

5.8 Redis Messaging/PubSub

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

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

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

Sending/Publishing messages

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

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

Receiving/Subscribing for messages

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

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

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

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

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

Message Listener Containers

Due to its blocking nature, low-level subscription is not attractive as it requires connection and thread management for every single listener. To alleviate this problem, Spring Data offers `RedisMessageListenerContainer` which does all the heavy lifting on behalf of the user - users familiar with EJB and JMS should find the concepts familiar as it is designed as close as possible to the support in Spring Framework and its message-driven POJOs (MDPs)

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

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

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

The MessageListenerAdapter

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

Consider the following interface definition. Notice that although the interface extends the `MessageListener` interface, it can still be used as a MDP via the use of the `MessageListenerAdapter` class. Notice also how the various message handling methods are strongly typed according to the **contents** of the various `Message` types that they can receive and handle. In addition, the channel or pattern to which a message is sent can be passed in to the method as the second argument of type String:

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

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

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

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

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

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

Note

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

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

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

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

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

5.9 Redis Transactions

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

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

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

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

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

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

@Transactional Support

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

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

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

  @Bean
  public RedisConnectionFactory redisConnectionFactory( // jedis, lettuce, srp,... );

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

```
/** Usage Constrainsts **/

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

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

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

5.10 Pipelining

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

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

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

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

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

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

5.11 Redis Scripting

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

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

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

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

```
@Bean
public RedisScript<Boolean> script() {
  DefaultRedisScript<Boolean> redisScript = new DefaultRedisScript<Boolean>();
  redisScript.setScriptSource(new ResourceScriptSource(new ClassPathResource("META-INF/scripts/
checkandset.lua")));
  redisScript.setResultType(Boolean.class);
}
```

```
public class Example {
  @Autowired
  RedisScript<Boolean> script;
  public boolean checkAndSet(String expectedValue, String newValue) {
    return redisTemplate.execute(script, Collections.singletonList("key"), expectedValue, newValue);
  }
}
```

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

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

The checkAndSet method above then executes th Scripts can be executed within a `SessionCallback` as part of a transaction or pipeline. See Section 5.9, "Redis Transactions" and Section 5.10, "Pipelining" for more information.

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

5.12 Support Classes

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

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

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

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

</beans>
```

```
public class AnotherExample {

  // injected
  private Deque<String> queue;

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

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

Support for Spring Cache Abstraction

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

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xmlns:c="http://www.springframework.org/schema/c"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/
beans/spring-beans.xsd
    http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache/spring-
cache.xsd">

  <!-- turn on declarative caching -->
  <cache:annotation-driven />

  <!-- declare Redis Cache Manager -->
  <bean id="cacheManager" class="org.springframework.data.redis.cache.RedisCacheManager" c:template-
ref="redisTemplate"/>
</beans>
```

Note

By default `RedisCacheManager` will lazily initialize `RedisCache` whenever a `Cache` is requested. This can be changed by predefining a `Set` of cache names.

Note

By default `RedisCacheManager` will not participate in any ongoing transaction. Use `setTransactionAware` to enable transaction support.

Note

By default `RedisCacheManager` does not prefix keys for cache regions, which can lead to an unexpected growth of a `ZSET` used to maintain known keys. It's highly recommended to enable the usage of prefixes in order to avoid this unexpected growth and potential key clashes using more than one cache region.

5.13 Roadmap ahead

Spring Data Redis project is in its early stages. We are interested in feedback, knowing what your use cases are, what are the common patters you encounter so that the Redis module better serves your needs. Do contact us using the channels mentioned above, we are interested in hearing from you!

Part V. Appendixes

Appendix Document structure

Various appendixes outside the reference documentation.

Appendix A, *Schema* defines the schemas provided by Spring Data Redis.

```
interface or provide a method named 'handleMessage'.
        ]]></xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="serializer" type="xsd:string">
      <xsd:annotation>
        <xsd:documentation><![CDATA[
A reference to the RedisSerializer strategy for converting Redis Messages to
listener method arguments. Default is a StringRedisSerializer.
        ]]></xsd:documentation>
        <xsd:appinfo>
          <tool:annotation kind="ref">
            <tool:expected-type type="org.springframework.data.redis.serializer.RedisSerializer"/>
          </tool:annotation>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:attribute>
  </xsd:complexType>

  <xsd:element name="collection">
    <xsd:annotation>
      <xsd:documentation><![CDATA[
Factory creating collections on top of Redis keys.
      ]]></xsd:documentation>
      <xsd:appinfo>
        <tool:annotation>

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

Appendix B. Command Reference

B.1 Supported commands

Table B.1. Redis commands supported by RedisTemplate.

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

Command	Template Support
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
GET	X
GETBIT	X
GETRANGE	X
GETSET	X
HDEL	X
HEXISTS	X
HGET	X
HGETALL	X
HINCRBY	X
HINCRBYFLOAT	X
HKEYS	X
HLEN	X
HMGET	X
HMSET	X
HSCAN	X
HSET	X
HSETNX	X

Command	Template Support
HVALS	X
INCR	X
INCRBY	X
INCRBYFLOAT	X
INFO	X
KEYS	X
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
MSET	X
MSETNX	X
MULTI	X
OBJECT	-
PERSIST	X
PEXIPRE	X
PEXPIREAT	X
PFADD	X
PFCOUNT	X

Command	Template Support
PFMERGE	X
PING	X
PSETEX	X
PSUBSCRIBE	X
PTTL	X
PUBLISH	X
PUBSUB	-
PUBSUBSCRIBE	-
QUIT	X
RANDOMKEY	X
RENAME	X
RENAMENX	X
RESTORE	X
ROLE	-
RPOP	X
RPOPLPUSH	X
RPUSH	X
RPUSHX	X
SADD	X
SAVE	X
SCAN	X
SCARD	X
SCRIPT EXITS	X
SCRIPT FLUSH	X
SCRIPT KILL	X
SCRIPT LOAD	X
SDIFF	X
SDIFFSTORE	X
SELECT	X
SENTINEL FAILOVER	X

Command	Template Support
SENTINEL GET-MASTER-ADD-BY-NAME	-
SENTINEL MASTER	-
SENTINEL MASTERS	X
SENTINEL MONITOR	X
SENTINEL REMOVE	X
SENTINEL RESET	-
SENTINEL SET	-
SENTINEL SLAVES	X
SET	X
SETBIT	X
SETEX	X
SETNX	X
SETRANGE	X
SHUTDOWN	X
SINTER	X
SINTERSTORE	X
SISMEMBER	X
SLAVEOF	X
SLOWLOG	-
SMEMBERS	X
SMOVE	X
SORT	X
SPOP	X
SRANDMEMBER	X
SREM	X
SSCAN	X
STRLEN	X
SUBSCRIBE	X
SUNION	X

Command	Template Support
SUNIONSTORE	X
SYNC	-
TIME	X
TTL	X
TYPE	X
UNSUBSCRIBE	X
UNWATCH	X
WATCH	X
ZADD	X
ZCARD	X
ZCOUNT	X
ZINCRBY	X
ZINTERSTORE	X
ZLEXCOUNT	-
ZRANGE	X
ZRANGEBYLEX	-
ZREVRANGEBYLEX	-
ZRANGEBYSCORE	X
ZRANK	X
ZREM	X
ZREMRANGEBYLEX	-
ZREMRANGEBYRANK	X
ZREVRANGE	X
ZREVRANGEBYSCORE	X
ZREVRANK	X
ZSCAN	X
ZSCORE	X
ZUNINONSTORE	X