http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/redis.html

|  |  |  |
| --- | --- | --- |
| **4. Redis support** | | |
| [Prev](http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/reference.html) | **Part II. Reference Documentation** | [Next](http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/appendixes.html) |

**4. Redis support**

One of the key value stores supported by Spring Data is [Redis](http://redis.io/). 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.

**4.1 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](https://github.com/xetorthio/jedis), [JRedis](https://github.com/alphazero/jredis), [SRP](https://github.com/spullara/redis-protocol) and[Lettuce](https://github.com/wg/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.

**4.2 Redis Support High Level View**

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

* *Low-Level Abstractions* - for configuring and handling communication with Redis through the various connector libraries supported as described in [Section 4.3, “Connecting to Redis”](http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/redis.html#redis:connectors).
* *High-Level Abstractions* - providing generified, user friendly template classes for interacting with Redis. [Section 4.4, “Working with Objects through RedisTemplate”](http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/redis.html#redis:template)explains the abstraction built on top of the low-level Connection API to handle the infrastructural concerns and object conversion.
* *Support Classes* - that offer reusable components (built on the aforementioned abstractions) such as java.util.Collection or Spring [cache](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html) implementation backed by Redis as documented in [Section 4.11, “Support Classes”](http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/redis.html#redis:support)

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.

**4.3 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 activeconnections to Redis.

**4.3.1 RedisConnection and RedisConnectionFactory**

RedisConnection provides the building block for Redis communication as it handles the communication with the Redis back-end. It also automatically translates the underlying connecting library exceptions to Spring's consistent DAO exception [hierarchy](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/dao.html#dao-exceptions) so one can switch the connectors without any code changes as the operation semantics remain the same.

|  |  |
| --- | --- |
| [Note] | **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 RedisConnections are created through RedisConnectionFactory. In addition, the factories act as PersistenceExceptionTranslators, 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](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/orm.html#orm-exception-translation) in Spring Framework documentation.

|  |  |
| --- | --- |
| [Note] | **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.

**Connector features**

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

**4.3.2 Configuring Jedis connector**

[Jedis](https://github.com/xetorthio/jedis) is one of the connectors supported by the Spring Data Redis module through theorg.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>

**4.3.3 Configuring JRedis connector**

[JRedis](https://github.com/alphazero/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:host-name="server" p:port="6379"/>

</beans>

The configuration is quite similar to Jedis, with one notable exception. By default, the JedisConnectionFactory 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>

**4.3.4 Configuring SRP connector**

[SRP](https://github.com/spullara/redis-protocol) (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.

**4.3.5 Configuring Lettuce connector**

[Lettuce](https://github.com/wg/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:host-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 LettuceConnectionFactoryshare 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.

**4.4 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](http://redis.io/commands)) that offer rich, generified interfaces for working against a certain type or certain key (through the KeyBound interfaces) as described below:

**Table 4.1. 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 |
| *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 4.6, “Serializers”](http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/redis.html#redis:serializer) 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());

}

}

**4.5 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 StringRedisSerializerunderneath 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");

...

}

});

}

**4.6 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](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/oxm.html) support or either JacksonJsonRedisSerializer or Jackson2JsonRedisSerializer for storing data in [JSON](https://en.wikipedia.org/wiki/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.

**4.7 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.

**4.7.1 Sending/Publishing messages**

To publish a message, one can use, as with the other operations, either the low-level RedisConnection or the high-level RedisTemplate. Both entities offer the publishmethod 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");

**4.7.2 Receiving/Subscribing for messages**

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

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

|  |  |
| --- | --- |
| [Important] | **Important** |
| When using Jedis or JRedis connectors, subscribing commands are synchronous and thus 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](http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/redis.html#redis:pubsub:subscribe:containers) 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 theMessageListenerAdapter 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] | **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:

*<!-- this is the Message Driven POJO (MDP) -->*

**<bean id="messageListener" class="org.springframework.data.redis.listener.adapter.MessageListenerAdapter">**

<constructor-arg>

<bean class="redisexample.DefaultMessageDelegate"/>

</constructor-arg>

</bean>

*<!-- and this is the message listener container... -->*

<bean id="redisContainer" class="org.springframework.data.redis.listener.RedisMessageListenerContainer">

<property name="connectionFactory" ref="connectionFactory"/>

<property name="messageListeners">

*<!-- map of listeners and their associated topics (channels or/and patterns) -->*

<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).

**4.8 Redis Transactions**

Redis provides support for [transactions](http://redis.io/topics/transactions) through the multi, exec, and discard commands. These operations are available on RedisTemplate, however RedisTemplateis 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] | **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. |

**4.9 Pipelining**

Redis provides support for [pipelining](http://redis.io/topics/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 orSessionCallback 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] | **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. |

**4.10 Redis Scripting**

Redis versions 2.6 and higher provide support for execution of Lua scripts through the [eval](http://redis.io/commands/eval) and [evalsha](http://redis.io/commands/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.

<?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="script" class="org.springframework.data.redis.core.script.DefaultRedisScript">

<property name="location" value="classpath:META-INF/scripts/checkandset.lua"/>

<property name="resultType" value="java.lang.Boolean"/>

</bean>

...

</beans>

**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 resultTypeshould 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-calcuation of the script's SHA1 on every script execution.

The checkAndSet method above then executes the configured RedisScript with the provided key and arguments and returns the result.

Scripts can be executed within a SessionCallback as part of a transaction or pipeline. See [Section 4.8, “Redis Transactions”](http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/redis.html#tx) and [Section 4.9, “Pipelining”](http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/redis.html#pipeline) 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.

**4.11 Support Classes**

Package org.springframework.data.redis.support offers various reusable components that rely on Redis as a backing store. Curently the package contains various JDK-based interface implementations on top of Redis such as [atomic](http://download.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html) counters and JDK [Collections](http://download.oracle.com/javase/6/docs/api/java/util/Collection.html).

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).

**4.11.1 Support for Spring Cache Abstraction**

Spring Redis provides an implementation for Spring [cache abstraction](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html) 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] | **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] | **Note** |
| By default RedisCacheManager will not participate in any ongoing transaction. Use setTransactionAware to enable transaction support. |

**4.12 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](http://docs.spring.io/spring-data/data-redis/docs/1.2.0.RC1/reference/html/get-started.html#get-started:help:community) above, we are interested in hearing from you!