Spring Data Redis Reference Documentation

1.0.3.RELEASE

Costin Leau SpringSource

Table of Contents

Preface

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

Part I. 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 3, *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.

1. 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) member 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 place code required for interacting with the store through Spring's excellent infrastructure support.

2. Requirements

Spring Data Redis 1.x binaries requires JDK level 6.0 and above, and Spring Framework 3.0.x and above.

In terms of key value stores, Redis 2.2.x is required (though 2.4 or higher is recommended).

3. 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 Spring Data Key Value 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.

3.1 First Steps

As explained in Chapter 1, *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. These being said, the more knowledge one has about the Spring, the faster she will pick Spring Data Key Value. 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 framework home page for more information. In general, this should be the starting point for developers wanting to try Spring DKV.

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 worth 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 SDKV. The best way to get acquainted to this 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/SpringSource/spring-data-keyvalue-examples. For Spring 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.

3.2 Need Help?

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

Community Support

The Spring Data forum is a message board for all Spring Data (not just Key Value) 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 SpringSource, the company behind Spring Data and Spring.

3.3 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 through the Spring Community forums.

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 SpringSource Data blog or the project team on Twitter (Costin)

Part II. Reference Documentation

Document structure

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

Chapter 4, *Redis support* introduces the Redis module feature set.

4. 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 application. Offers both low-level and high-level abstraction for interacting with the store, freeing the user from infrastructural concerns.

4.1 Redis Requirements

Spring Redis requires Redis 2.0 or above and Java SE 5.0 or above . In terms of language bindings (or connectors), Spring Redis integrates with Jedis, JRedis, RJC, SRP and Lettuce, five popular open source Java libraries for Redis. If you are aware of any other connector that we should be integrating is, 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”.
- *High-Level Abstractions* - providing a generified, user friendly template classes for interacting with Redis. Section 4.4, “Working with Objects through `RedisTemplate`” explains the abstraction builds 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 3.1 cache implementation backed by Redis as documented in Section 4.8, “Support Classes”

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 libray) 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 only one set of Spring 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 respectively for retrieving active `connection` 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` are created through `RedisConnectionFactory`. In addition, the factories act as `PersistenceExceptionTranslator` meaning once declared, allow one to do transparent exception translation for example 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 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 connectors support all of Redis features - in particular JRedis does not have support for hashes yet though this is currently being worked on. 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.

Configuring Jedis connector

Jedis is one of the connectors supported by the Key Value module through the `org.springframework.data.redis.connection.jedis` package. In its simples 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.JedisConnectionFa
>
</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.JedisConnectionFa
        p:host-name="server" p:port="6379"/>
</beans>
```

Configuring JRedis connector

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

Note

Since JRedis itself does not support (yet) Redis 2.x commands, Spring Redis uses an updated fork available here.

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.JredisConnecti
        p:host-name="server" p:port="6379"/>
</beans>
```

As one can note, the configuration is quite similar to the Jedis one.

Important

Currently, JRedis does not have support for binary keys. This forces the `JredisConnection` to perform encoding internally (through base64 schema). In practice, this means it's safe to read/write arbitrary data however the Redis key stored values will differ from the decoded ones, even in the simplest cases, since everything (no matter the format) is encoded. This will not be the case for Redis values.

This issue is currently being addressed in the JRedis project and once fixed, will be incorporated by Spring Data Redis.

Configuring RJC connector

RJC is the third, open-source connector supported by Spring Redis through the `org.springframework.data.redis.connection.rjc` package.

Similar to the other connectors, a typical RJC 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="rjcConnectionFactory" class="org.springframework.data.redis.connection.rjc.RjcConnectionFactory"
        p:host-name="server" p:port="6379"/>
</beans>
```

As one can note, the configuration is quite similar to the Jredis or Jedis one.

Important

Currently, RJC does not have support for binary keys. This forces the `RjcConnection` to perform encoding internally (through base64 schema). In practice, this means it's safe to read/write arbitrary data however the Redis key stored values will differ from the decoded ones, even in the simplest cases, since everything (no matter the format) is encoded. This will not be the case for Redis values.

This issue is currently being addressed in the RJC project and once fixed, will be incorporated by Spring Data Redis.

Configuring SRP connector

SRP (an acronym for Sam's Redis Protocol) is the forth, open-source connector supported by Spring 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 fifth, open-source connector supported by Spring 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="srpConnectionFactory" class="org.springframework.data.redis.connection.lettuce.LettuceConnectio
        p:host-name="server" p:port="6379"/>
</beans>
```

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

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 interaction - while `RedisConnection` offer 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 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” for more information. 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, one 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.JedisConnectionFa
        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
  @Resource(name="redisTemplate")
  private RedisTemplate<String, String> template;

  // inject the template as ListOperations
  @Autowired
  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 the keys and values stored in Redis can 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 be 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.JedisConnectionF
        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`.

```
public void useCallback() {
  redisTemplate.execute(new RedisCallback<Object>() {

    public Object doInRedis(RedisConnection connection) throws DataAccessException {
      Long size = connection.dbSize();
      ...
    }
  });
}
```

4.6 Serializers

From the framework perspective, the data stored in Redis are 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 Redis 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 `JacksonJsonRedisSerializer` 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.

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 containers 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 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.

Important

Subscribing commands are synchronized 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` respectively `pUnsubscribe` on the *same* connection. See message listener container below for a solution to these problem.

As mentioned above, one 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 else then `subscribe`, `pSubscribe`, `unsubscribe`, `pUnsubscribe` or is illegal and will through an exception.

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

Message Listener Containers

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

`RedisMessageListenerContainer` acts as a message listener container; it is used to receive messages from a Redis channel and drive the MessageListener 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 suchlike. 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.

Further more, to minimize the application footprint, `RedisMessageListenerContainer` performs 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:

```
<!-- 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 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 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="redisTemplat"/>
    <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 3.1 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>
```

4.9 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 III. Appendixes

Document structure

Various appendixes outside the reference documentation.

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

Appendix A. Spring Data Redis Schema(s)

Core schema

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

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

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

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

  <xsd:element name="listener-container">
    <xsd:annotation>
      <xsd:documentation><![CDATA[
Container of Redis listeners. All listeners will be hosted by the same container.
      ]]></xsd:documentation>
      <xsd:appinfo>
        <tool:annotation>

  <tool:exports type="org.springframework.data.redis.listener.RedisMessageListenerContainer"/
>
        </tool:annotation>
      </xsd:appinfo>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>

  <xsd:element name="listener" type="listenerType" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="connection-
factory" type="xsd:string" default="redisConnectionFactory">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
A reference to the Redis ConnectionFactory bean.
Default is "redisConnectionFactory".
          ]]></xsd:documentation>
          <xsd:appinfo>
            <tool:annotation kind="ref">
              <tool:expected-
type type="org.springframework.data.redis.connection.ConnectionFactory"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="task-executor" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
A reference to a Spring TaskExecutor (or standard JDK 1.5 Executor) for executing
Redis listener invokers. Default is a SimpleAsyncTaskExecutor.
          ]]></xsd:documentation>
          <xsd:appinfo>
            <tool:annotation kind="ref">
              <tool:expected-type type="java.util.concurrent.Executor"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="subscription-task-executor" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
```