Spring Data Key-Value - Reference Documentation

1.0.0.M3

Costin Leau (SpringSource), Jon Brisbin (SpringSource)

Preface

The Spring Data Key-Value 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 - Key Value 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 Key Value Support and assumes the user is familiar with the key value storages and Spring concepts.

Chapter 1. Why Spring Data - Key Value?

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 Key Value (or SDKV) framework makes it easy to write Spring applications that use a Key Value store by eliminating the redundant tasks and boiler place code required for interacting with the store through Spring's excellent infrastructure support.

Chapter 2. Requirements

Spring Data Key Value 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 2.0.x and Riak 0.13 are required.

Chapter 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 - Key Value?*, Spring Data Key Value (SDKV) provides integration between Spring framework and key value (KV) stores. 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 SDKV documentation, each section provides links to resources relevant however, it is best to become familiar with these topics beforehand.

3.1.1. Knowing Spring

Spring Data uses heavily Spring framework's core functionality, such as the IoC container, resource abstract or AOP infrastructure. While it is not important to know the Spring APIs, understanding the concepts behind them is. At a minimum, the idea behind IoC should be familiar. 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.

3.1.2. Knowing NoSQL and Key Value stores

NoSQL stores have taken the storage world by storm. It is a vast domain with a plethora of solutions, terms and patterns (to make things 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.

3.1.3. Trying Out The Samples

Unfortunately the SDKV project is very young and there are no samples available yet. However we are working on them and plan to make them available as soon as possible. In the meantime however, one can use our test suite as a code example (assuming the documentation is not enough) - we provide extensive integration tests for our code base.

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:

3.2.1. 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.

3.2.2. 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 Key Value.

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

Chapter 5, *Riak Support* introduces the Riak module feature set.

Chapter 4. Redis support

One of the key value stores supported by SDKV 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 Key Value 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

SDKV requires Redis 2.0 or above (Redis 2.2 is recommended) and Java SE 6.0 or above. In terms of language bindings (or connectors), SDKV integrates with Jedis, JRedis and RJC, three 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` 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; currently SDKV has support for Jedis and JRedis. No matter the library one chooses, there only one set of SDKV API that one needs to use that behaves consistently across all connectors, namely the `org.springframework.data.keyvalue.redis.connection` package and its `RedisConnection` and `RedisConnectionFactory` interfaces for working respectively for retrieving active `connection` 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 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.

4.3.2. Configuring Jedis connector

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

4.3.3. Configuring JRedis connector

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

Note

Since JRedis itself does not support (yet) Redis 2.x commands, SDKV 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.keyvalue.redis.connection.jredis.JredisConn
        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.

4.4. Working with Objects through `RedisTemplate`

Most users are likely to use `RedisTemplate` and its coresponding package `org.springframework.data.keyvalue.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

Interface	Description
`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.keyvalue.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.keyvalue.redis.connection.jedis.JedisConnect
        p:use-pool="true"/>

  <!-- redis template definition -->
  <bean id="redisTemplate" class="org.springframework.data.keyvalue.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
  @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.keyvalue.redis.connection.jedis.JedisConnect
        p:use-pool="true"/>

  <bean id="stringRedisTemplate" class="org.springframework.data.keyvalue.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 SDKV Redis through the `RedisSerializer` interface (package `org.springframework.data.keyvalue.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.keyvalue.redis.connection` and `org.springframework.data.keyvalue.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 `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");
```

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 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.

4.7.2.1. 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.

4.7.2.2. 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.

```
public interface MessageDelegate {

    void handleMessage(String message);

    void handleMessage(Map message);

    void handleMessage(byte[] message);

    void handleMessage(Serializable message);
}
```

```
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/
        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" channel="chatroom" />
  </redis:listener-container>

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

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.keyvalue.redis.listener.adapter.MessageListenerAdapte
    <constructor-arg>
        <bean class="redisexample.DefaultMessageDelegate"/>
    </constructor-arg>
</bean>

<!-- and this is the message listener container... -->
<bean id="redisContainer" class="org.springframework.data.keyvalue.redis.listener.RedisMessageListenerContainer"
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="messageListeners">
      <!-- map of listeners and their associated topics (channels or topics) -->
      <map>
        <entry key-ref="messageListener">
            <bean class="org.springframework.data.keyvalue.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.keyvalue.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.keyvalue.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).

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!

Chapter 5. Riak Support

Riak is a Key/Value datastore that supports Internet-scale data replication for high performance and high availability. Spring Data Key/Value (SDKV) provides access to the Riak datastore over the HTTP REST API using a built-in driver based on Spring 3.0's `RestTemplate`. In addition to making Key/Value datastore access easier from Java, the `RiakTemplate` has been designed, from the ground up, to be used from alternative JVM languages like Groovy or JRuby.

Since the SDKV support for Riak uses the stateless REST API, there are no connection factories to manage or other stateful objects to keep tabs on. The helper you'll spend the most time working with is likely the thread-safe `RiakTemplate` or `RiakKeyValueTemplate`. Your choice of which to use will depend on how you want to manage buckets and keys. SDKV supports two ways to interact with Riak. If you want to use the convention you're likely already familiar with, namely of storing an entry with a given key in a "bucket" by passing the bucket and key name separately, you'll want to use the `RiakTemplate`. If you want to use a single object to represent your bucket and key pair, you can use the `RiakKeyValueTemplate`. It supports a key object that is encoded using one of several different methods:

- *Using a `String`* - You can concatenate two strings, separated by a colon: "mybucket:mykey".
- *Using a `BucketKeyPair`* - You can pass an instance of `BucketKeyPair`, like `SimpleBucketKeyPair`.
- *Using a `Map`* - You can pass a `Map` with keys for "bucket" and "key".

5.1. Configuring the `RiakTemplate`

This is likely the easiest path to using SDKV for Riak, as the bucket and key are passed separately. The examples that follow will assume you're using this version of the the template.

There are only two options you need to set to specify the Riak server to use in your `RiakTemplate` object: "defaultUri" and "mapReduceUri". Encoded with the URI should be placeholders for the bucket and the key, which will be filled in by the `RestTemplate` when the request is made.

Important

You can also turn the internal, ETag-based object cache off by setting `useCache="false"`. It's generally recommended, however, to leave the internal cache on as the ETag matching will pick up any changes made to the entry on the Riak side and your application will benefit from greatly-increased performance for often-requested objects.

```
<?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="riakTemplate" class="org.springframework.data.keyvalue.riak.core.RiakTemplate"
          p:defaultUri="http://localhost:8098/riak/{bucket}/{key}"
          p:mapReduceUri="http://localhost:8098/mapred"
          p:useCache="true"/>

</beans>
```

5.1.1. Advanced Template Configuration

There are a couple additional properties on the `RiakTemplate` that can be changed from their defaults. If you want to specify your own ConversionService to use when converting objects for storage inside Riak, then set it on the "conversionService" property:

```
<?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="conversionService" class="com.mycompany.convert.MyConversionService"/>
    <bean id="riakTemplate" class="org.springframework.data.keyvalue.riak.core.RiakTemplate"
          p:defaultUri="http://localhost:8098/riak/{bucket}/{key}"
          p:mapReduceUri="http://localhost:8098/mapred"
          p:conversionService-ref="conversionService"/>

</beans>
```

Depending on the application, it might be useful to set default Quality-of-Service parameters. In Riak paralance, these are the "dw", "w", and "r" parameters. They can be set to an integer representing the number of vnodes that need to report having received the data before declaring the operation a success, or the string "one", "all", or (the default) "quorum". These values can be overridden by passing a different set of `QosParameters` to the set/get operation you're performing.

```
<?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="qos" class="org.springframework.data.keyvalue.riak.core.RiakQosParameters"
          p:durableWriteThreshold="all"
          p:writeThreshold="all"/>
    <bean id="riakTemplate" class="org.springframework.data.keyvalue.riak.core.RiakTemplate"
          p:defaultUri="http://localhost:8098/riak/{bucket}/{key}"
          p:mapReduceUri="http://localhost:8098/mapred"
          p:defaultQosParameters-ref="qos"/>

</beans>
```

You can also set a specific `ClassLoader` to use when loading objects from Riak. Just set the `classLoader` property:

```
<?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="riakTemplate" class="org.springframework.data.keyvalue.riak.core.RiakTemplate"
          p:defaultUri="http://localhost:8098/riak/{bucket}/{key}"
          p:mapReduceUri="http://localhost:8098/mapred"
          p:classLoader-ref="customClassLoader"/>

</beans>
```

5.2. Working with Objects using the `RiakTemplate`

One of the primary goals of the SDKV project is to make accessing Key/Value stores easier for the developer by taking away the mundane tasks of basic IO, buffering, type conversion, exception handling, and sundry other logistical concerns so the developer can focus on creating great applications. SDKV for Riak works toward this goal by making basic persistence and data access as easy as using a `Map`.

5.2.1. Saving data into Riak

To store data in Riak, use one of the six different `set` methods:

```
import org.springframework.data.keyvalue.riak.core.RiakTemplate;

public class Example {

    @Autowired
    RiakTemplate riak;

    public void setData(String bucket, String key, String data) throws Exception {
        riak.set(bucket, key, data); // Set as Content-Type: text/plain
        riak.setAsBytes(bucket, key, data.getBytes()); // Set as Content-Type: application/octet-stream
    }

    public void setData(String bucket, String key, MyPojo data) throws Exception {
        riak.set(bucket, key, data); // Converted to JSON automatically, Content-Type: application/json
    }

}
```

Additionally, there is a `setWithMetaData` method that takes a `Map` of metadata that will be set as the outgoing HTTP headers. To set custom metadata, your key should be prefixed with `X-Riak-Meta-` e.g. `X-Riak-Meta-Custom-Header`.

5.2.1.1. Letting Riak generate the key

Riak has the ability to generate random IDs for you when storing objects. The `RiakTemplate` exposes this capability via the `put` method. It will return the ID it generated for you as a `String`.

```
import org.springframework.data.keyvalue.riak.core.RiakTemplate;

public class Example {

    @Autowired
    RiakTemplate riak;

    public String setData(String bucket, String data) throws Exception {
        String id = riak.put(bucket, data); // Returns the generated ID
        return id;
    }

}
```

5.2.2. Retrieving data from Riak

Retrieving data from Riak is just as easy. There are actually 13 different `get` methods on `RiakTemplate` that give the developer a wide range options for accessing and converting your data.

Assuming you've stored a POJO using an appropriate `set` method, you can retrieve that object from Riak using a `get`:

```
import org.springframework.data.keyvalue.riak.core.RiakTemplate;
```

```
public class Example {

    @Autowired
    RiakTemplate riak;

    public void getData(String bucket, String key) throws Exception {
        // What you get depends on Content-Type.
        // application/json=Map, text/plain=String, etc...
        Object o = riak.get(bucket, key);

        // If your entry is Content-Type: application/json...
        // It will automatically be converted when retrieved.
        MyPojo s = riak.getAsType(bucket, key, MyPojo.class);

        // If your entry is Content-Type: application/octet-stream,
        // you can access the raw bytes.
        byte[] b = riak.getAsBytes(bucket, key);
    }

}
```

5.3. Linking Entries

Riak has the ability to link entries together using an arbitrary tag. This relationship information is stored in the `Link` header. The `RiakTemplate` exposes a method for linking entries together called `link`. Its usage is quite simple:

Important

A link is uni-directional, so keep in mind that the bucket and key you pass first should be that of the child (or target) object and the second set of bucket/key pairs you pass to the `link` method is the source of the relationship. It's this second entry that will receive an updated `Link` header that points to the child or target entry.

```
@Autowired
RiakTemplate riak;

riak.link("childbucket", "childkey", "sourcebucket", "sourcekey", "tagname");
```

Now, querying the metadata on the entry at `sourcebucket:sourcekey` will result in a Link header that points to the child object: `</riak/childbucket/childkey>; riaktag="tagname"`

5.3.1. Link Walking

When entries are linked together in Riak, those relationships can be efficiently traversed on the server using a feature called Link Walking. Rather than requesting each object in a link's relationship individually, a link walk pulls all the related objects at once and sends that data back to the client as MIME-encoded multipart data. As such, it requires special processing to convert those multiple entries into a `List` of objects, just as if you had used a `get` method. If you don't specify a type to convert the objects to, the `linkWalk` method will try to infer it from the bucket name. If the bucket name is not a valid class name, it will default to using a `java.util.Map`.

To link walk a relationship and return a list of custom POJOs, you would do something like this:

```
@Autowired
RiakTemplate riak;
```

```
List<MyPojo> result = riak.linkWalk("sourcebucket", "sourcekey", "tagname", MyPojo.class);
```

5.4. Map/Reduce

Riak supports Map/Reduce functionality in a couple different ways. You can specify the Javascript source to execute (termed "anonymous" Javascript), you can reference some Javascript already stored in Riak at a specfic bucket and key, or you can reference an Erlang module and function. The Map/Reduce support in SDKV covers all these bases by giving you meaningful abstractions over the Map/Reduce job that represent the various aspects of the Map/Reduce process.

At the highest level, every Map/Reduce request is represented by a MapReduceJob. The `MapReduceJob` represents the `inputs`, the `phases`, and the optional `arg` to send to Riak to execute the Map/Reduce job. The `toJson` method is responsible for serializing the entire job into the appropriate JSON data to send to Riak.

5.4.1. Specifying Inputs

Riak will accept either a string denoting the bucket in which to get the list of keys to operate on, or a `List` of `Lists` denoting the bucket/key pairs to operate on while executing this Map/Reduce job. If you call the `addInputs` method on the job passing a `List` with a single string entry, the job will assume you want to operate on an entire bucket. Otherwise, you'll need to pass a multi-dimensional `List` of bucket/key pairs.

To operate on an entire bucket:

```
@Autowired
RiakTemplate riak;

RiakMapReduceJob job = riak.createMapReduceJob();
List<String> bucket = new ArrayList<String>() {{
  add("mybucket");
}};
job.addInputs(bucket); // Will M/R entire bucket
```

To operate on a set of keys:

```
import org.springframework.data.keyvalue.riak.mapreduce.*;

@Autowired
RiakTemplate riak;

RiakMapReduceJob job = riak.createMapReduceJob();

List<String> pair = new ArrayList<String>() {{
  add("mybucket");
  add("mykey");
}};
List<List<String>> keys = new ArrayList<List<String>>() {{
  add(pair);
}};
job.addInputs(keys); // Will M/R only specified keys
```

5.4.2. Defining Phases

Map/Reduce operations in Riak are broken up into phases. Phases contain a `MapReduceOperation`. There are

currently two implementations to handle Javascript or Erlang M/R operations: `JavascriptMapReduceOperation` and `ErlangMapReduceOperation`.

An example Map/Reduce job defining a single "map" phase defined in anonymous Javascript might look like this:

```
import org.springframework.data.keyvalue.riak.mapreduce.*;

@Autowired
RiakTemplate riak;

RiakMapReduceJob job = riak.createMapReduceJob();
List<String> bucket = new ArrayList<String>() {{
  add("mybucket");
}};

job.addInputs(bucket); // M/R the entire bucket

MapReduceOperation mapOper = new JavascriptMapReduceOperation("function(v){ ...M/R function body... }");
MapReducePhase mapPhase = new RiakMapReducePhase("map", "javascript", mapOper);

job.addPhase(mapPhase);
```

5.4.3. Executing and Working with the Result

To execute a configured job on your Riak server, use either the synchronous `execute` or asynchronous `submit` methods of your configured `RiakTemplate`:

```
Object o = riak.execute(job); // Results of last Map or Reduce phase. Should be a List<?>

...or...

List<MyPojo> o = riak.execute(job, MyPojo.class); // Coerce to given type

...or...

Future<List<?>> f = riak.submit(job); // Job runs in a separate thread
```

5.5. Managing Bucket Properties

It's sometimes useful to manage settings like the Quality-of-Service parameters `w` and `dw` (write and durable write thresholds) and the `n_val` setting at the bucket level. It's also possible to list the keys in a particular bucket by calling the `getBucketSchema` method, passing `true` as the second parameter, which tells the `RiakTemplate` to list the keys.

To list the keys in a bucket, you would do something like this:

```
@Autowired
RiakTemplate riak;

Map<String, Object> schema = riak.getBucketSchema("mybucket", true);
List<String> keys = schema.get("keys")
for(String key : keys) {
  ...do something with each key...
}
```

To update the bucket settings, pass a `Map` of properties:

```
@Autowired
RiakTemplate riak;

Map<String, Integer> props = new HashMap<String, Integer>();
props.put("n_val", 6);
props.put("dw", 3);

riak.updateBucketSchema("mybucket", props);
```

Only the properties specified in the passed-in `Map` will be updated. Properties that have already been set in previous operations and not specified in this operation will be unaffected.

5.6. Asynchronous Access

SDKV for Riak also includes an asynchronous version of most of the methods available to the `RiakTemplate`, whose method calls are all synchronous. The asynchronous version of the template is called `AsyncRiakTemplate`.

5.6.1. Template Configuration

The `AsyncRiakTemplate` has the same basic configuration properties as the synchronous `RiakTemplate`. The only other property specific to the `AsyncRiakTemplate` you might want to configure is the thread pool the template uses to execute tasks asynchronously (by default a cached ThreadPoolExecutor). Set your `ExecutorService` on the template's `workerPool` property.

5.6.2. Callbacks

Using the asynchronous Riak support in SDKV means you'll be relying on callbacks to execute your business logic when the requested operation is completed. All asynchronous operations follow a similar pattern:

- They are named similarly to their synchronous counterparts.
- They take a `AsyncKeyValueStoreOperation<?, ?>` as a final parameter.
- They return a `Future<?>`.

To perform an asynchronous `get` on a JSON-serialized `Map` object which returns a custom object from the callback, you'd do something like:

```
@Autowired
AsyncRiakTemplate riak;

Future<MyObject> future = riak.get("mybucket", "mykey", new AsyncKeyValueStoreOperation<Map, MyObject>() {

  MyObject obj = new MyObject();

  MyObject completed(KeyValueStoreMetaData meta, Map result) {
    obj.setName(result.get("name"));
    return obj;
  }

  MyObject failed(Throwable error) {
    obj.setError(error);
    return obj;
  }

});
```

```
// Maybe do other work while waiting...
MyObject obj = future.get();
```

5.7. Groovy Builder Support

If your application uses Groovy, either in a standalone context, or as part of a Grails application, then you could benefit from using the Groovy `RiakBuilder` that comes with SDKV for Riak. Underneath, it uses the `AsyncRiakTemplate`. To use the `RiakBuilder`, pass the constructor a configured `AsyncRiakTemplate`.

Important

Instances of `RiakBuilder` are NOT thread-safe and should not be shared across threads.

The `RiakBuilder` implements an easy-to-use DSL for interacting with Riak. It doesn't implement the full set of methods available on the underlying `AsyncRiakTemplate` but a subset. The methods that the `RiakBuilder` responds to are:

- set
- setAsBytes
- put
- get
- getAsBytes
- getAsType
- containsKey
- delete
- foreach

5.7.1. Riak DSL Usage

The following example illustrates the different uses of the Riak DSL, including batching requests together into a logical group, using a default bucket name (the node directly beneath `riak` will be considered the default bucket to use for the contained operations unless a different one is specified on the operation itself):

```
def riak = new RiakBuilder(asyncRiakTemplate)
riak {
  test {
    put(value: [test: "value"]) { completed { v, meta -> meta.key }}
    put(value: [test: "value"]) { completed { v, meta -> meta.key }}
    put(value: [test: "value"]) { completed { v, meta -> meta.key }}
    put(value: [test: "value"]) { completed { v, meta -> meta.key }}

    mapreduce {
      query {
        map(arg: [test: "arg", alist: [1, 2, 3, 4]]) {
          source "function(v, keyInfo, arg){ return [1]; }"
        }
```

```
        reduce {
          source "function(v){ return Riak.reduceSum(v); }"
        }
      }
      failed { it.printStackTrace() }
    }
  }
}
def results = riak.results

riak.foreach(bucket: "test") {
  completed { v, meta ->
    riak.delete(bucket: "test", key: meta.key)
  }
}
```

Some important things to note from this example:

- Each operation in the Riak DSL has two callbacks: `completed` and `failed`.
- The `completed` closure is passed either the result object, or, if your closure is defined with two parameters, the result object and the metadata associated with that entry.
- Operations can be enclosed in an arbitrarily-named closure which the builder interprets as a default bucket name (in this case, the node "test" tells the builder to use the bucket name "test" for a default, unless one is specified on one of the enclosed operations).
- Each operation within a builder's execution will be accumulated inside the special `results` property. Code that needs to know the output of individual operations within the batch can get access to that object through this property. Note that this means that `RiakBuilder` instances are NOT thread-safe.

Important

Even though the Riak DSL uses an asynchronous template underneath, all operations performed through the DSL will, by default, block until complete. To get a truly asynchronous operation, pass the parameter `wait: 0` (or give a meaningful timeout in milliseconds to wait for the operation to complete) on the operation.

5.7.1.1. QosParameters on Riak DSL Operations

You can pass `QosParameters` to Riak DSL operations by simply defining them as parameters to the operation:

```
def riak = new RiakBuilder(asyncRiakTemplate)

def myobj = riak.set(bucket: "mybucket", key: "mykey", qos: ["dw": "all"])
```

5.7.1.2. Working with Riak DSL Output

The output of DSL operations will either be passed to the configured `completed` callback, or be returned to the caller if no callback is specified. In the example above, the `mapreduce` operation has no `completed` closure. Therefore, the return of the reduce phase is simply passed back to the builder, which makes that output available on the special `results` property.

To gain access to the operation's results immediately, simply assign it to a variable:

```
```

```
def riak = new RiakBuilder(asyncRiakTemplate)

def myobj = riak.get(bucket: "mybucket", key: "mykey")
```

If you add a non-zero `wait` value to the operation, "myobj" will contain a `Future<?>` rather than the result object itself.

5.8. Working with streams

SDKV for Riak includes a couple of useful helper objects to make reading and writing plain text or binary data in Riak really easy. If you want to store a file in Riak, then you can create a `RiakOutputStream` and simply write your data to it (making sure to call the "flush" method, which actually sends the data to Riak).

```
import org.springframework.data.keyvalue.riak.core.RiakTemplate;
import org.springframework.data.keyvalue.riak.core.io.RiakOutputStream;

public class Example {

    @Autowired
    RiakTemplate riak;

    public void writeToRiak(String bucket, String key, String data) throws Exception {
        OutputStream out = new RiakOutputStream(riak, bucket, key);
        try {
            out.write(data.getBytes());
        } finally {
            out.flush();
            out.close();
        }
    }

}
```

Reading data from Riak is similarly easy. SDKV provides a `java.io.File` subclass that represents a resource in Riak. There's also a Spring IO Resource abstraction called `RiakResource` that can be used anywhere a `Resource` is required. There's also an `InputStream` implementation called `RiakInputStream`.

```
import org.springframework.data.keyvalue.riak.core.RiakTemplate;
import org.springframework.data.keyvalue.riak.core.io.RiakInputStream;

public class Example {

    @Autowired
    RiakTemplate riak;

    public String readFromRiak(String bucket, String key) throws Exception {
        InputStream in = new RiakInputStream(riak, bucket, key);
        String data;
        ...read data and work with it...
        return data;
    }

}
```

Part III. Appendixes

Document structure

Various appendixes outside the reference documentation.

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

Appendix A. Spring Data Key Value Schema(s)

Spring Data - Redis support

```
<?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.

  <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.keyvalue.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.keyvalue.redis.connection.ConnectionFactory"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="task-executor" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
A reference to a Spring TaskExecutor (or standard JDK 1.5 Executor) for executing
Redis listener invokers. Default is a SimpleAsyncTaskExecutor.
          ]]></xsd:documentation>
          <xsd:appinfo>
            <tool:annotation kind="ref">
              <tool:expected-type type="java.util.concurrent.Executor"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="subscription-task-executor" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
A reference to a Spring TaskExecutor (or standard JDK 1.5 Executor) for listening
to Redis messages. By default reuses the 'task-executor' value.
          ]]></xsd:documentation>
          <xsd:appinfo>
```

```
            <tool:annotation kind="ref">
              <tool:expected-type type="java.util.concurrent.Executor"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="topic-serializer" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
A reference to the RedisSerializer strategy for converting Redis channels/patterns to
serialized format. Default is a StringRedisSerializer.
          ]]></xsd:documentation>
          <xsd:appinfo>
            <tool:annotation kind="ref">
              <tool:expected-type type="org.springframework.data.keyvalue.redis.serializer.RedisSerializer"/>
            </tool:annotation>
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="phase" type="xsd:string">
        <xsd:annotation>
          <xsd:documentation><![CDATA[
The lifecycle phase within which this container should start and stop. The lower
the value the earlier this container will start and the later it will stop. The
default is Integer.MAX_VALUE meaning the container will start as late as possible
and stop as soon as possible.
          ]]></xsd:documentation>
        </xsd:annotation>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>

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

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