Redis – As Spring Integraion

**What is a Redis?**

Redis is an open source (BSD licensed), in-memory key-store based data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries.

It allows to run atomic operations on these types, like appending to a string; incrementing the value in a hash; pushing an element to a list; computing set intersection, union and difference; or getting the member with highest ranking in a sorted set.

**Why, Redis?**

In order to achieve its outstanding performance, Redis works with an in-memory dataset. Depending on your use case, you can persist it either by dumping the dataset to disk every once in a while, or by appending each command to a log.

As NoSQL solutions are getting more and more popular for many kind of problems, more often the modern projects consider to use some (or several) of NoSQLs instead (or side-by-side) of traditional RDBMS, which led programmers to use Redis as efficient solution with great performance.

**Is Redis just a cache?**

These are important & popular use cases or features provided by Redis:

1. **Session Cache**
2. **Full Page Cache (FPC)**
3. **Queues**
4. **Publish / Subscribe**
5. **Leader Boards / Counting**
6. **Session Cache**

One of the most apparent use cases for Redis is using it as a session cache. The advantages of using Redis over other session stores, such as Memcached, is that Redis offers persistence. While maintaining a cache isn't typically mission critical with regards to consistency, most users wouldn't exactly enjoy if all their cart sessions went away, now would they?

Luckily, with the steam Redis has picked up over the years, it's pretty easy to find documentation on how to use Redis appropriately for session caching. Even the well-known ecommerce platform Magento has a plug in for Redis!

1. **Full Page Cache (FPC)**

Outside of your basic session tokens, Redis provides a very easy FPC platform to operate in. Going back to consistency, even across restarts of Redis instances, with disk persistence your users won't see a decrease in speed for their page loads—a drastic change from something like PHP native FPC.

1. **Queues**

Taking advantage of Redis' in memory storage engine to do list and set operations makes it an amazing platform to use for a message queue. Interacting with Redis as a queue should feel native to anyone used to using push/pop operations with lists in programming languages such as Java.

If you do a quick Google search on "Redis queues," you'll soon see that there are tons of open-source projects out there aimed at making Redis an awesome backend utility for all your queuing needs.

1. **Publish / Subscribe**

One of the greatest feature of Redis is Pub/Sub feature. The use cases for Pub/Sub are truly boundless. Keeping a map of who is interested in updates to what data is a common task in systems. Redis has a pub/sub feature to make this easy using commands like SUBSCRIBE, UNSUBSCRIBE, and PUBLISH. Many people use it for social network connections, for triggering scripts based on Pub/Sub events, and even a chat system built using Redis Pub/Sub!

1. **Leader Boards / Counting**

Redis does an amazing job at increments and decrements since it's in-memory. Sets and sorted sets also make our lives easier when trying to do these kinds of operations, and Redis just so happens to offer both of these data structures. So to pull the top 10 users from a sorted set—we'll call it “user\_scores”—one can simply run the following:

ZRANGE user\_scores 0 10

Of course, this is assuming you're ranking users on an incremental score. If you wanted to return both the users and their score, you could run something such as:

ZRANGE user\_scores 0 10 WITHSCORES

**Redis with Spring???**

Spring Integration introduces support for Redis: \_"an open source advanced key-value store". \_ This support comes in the form of a Redis-based MessageStore as well as Publish-Subscribe Messaging adapters that are supported by Redis via its PUBLISH, SUBSCRIBE and UNSUBSCRIBE commands.

**An introduction to** *Spring Data Redis*

Spring Data Regis provides the abstractions of the Spring Data platform to Redis – the popular in-memory data structure store. We’ll be able to use the common patterns of Spring Data (templates, etc.), while also having the traditional simplicity of all Spring Data projects.

It is very much simple to integrate Redis into your Spring application as follows:

**Setting up Redis in windows**

Redis uses a standard practice for its versioning: major, minor, patchlevel. The Redis project does not officially support Windows. However, the Microsoft Open Tech group develops and maintains this Windows port targeting Win64.

You can download a moderately out-of-date precompiled version of Redis for 32-bit and 64-bit Windows thanks to Dusan Majkic from his GitHub [page](https://github.com/dmajkic/redis/downloads): https://github.com/dmajkic/redis/downloads.

1. After you download Redis, you’ll need to extract the executables from the zip file.
2. Now extract zip file containing executables files.
3. After you’ve extracted either the 32- or 64-bit version of Redis to a location of your choice, you can start Redis by double-clicking on the redis-server.exe executable.
4. The same folder contains redis-cli.exe executable also, to run client application.
5. Redis will start with default configuration as given in redis.conf.exe file in same directory.

**Connecting with Redis using Spring**

To begin interacting with Redis we first need to connect to it. Spring Integration uses support provided by another Spring project, Spring Data Redis, which provides typical Spring constructs: ConnectionFactory and Template. These abstractions simplify integration with several Redis-client Java APIs. Currently Spring-Data-Redis supports *jedis*, *jredis* and *rjc*.

The Redis configuration

Java Configuration

@Bean

JedisConnectionFactory jedisConnectionFactory() {

return new JedisConnectionFactory();

}

@Bean

public RedisTemplate<String, Object> redisTemplate() {

RedisTemplate<String, Object> template = new RedisTemplate<String, Object>();

template.setConnectionFactory(jedisConnectionFactory());

return template;

}

In Spring’s XML configuration:

<!-- Redis Connection Factory -->

<bean id="jedisConnFactory" class="o.s.data.redis.connection.jedis.JedisConnectionFactory">

<property name="port" value="7379" />

</bean>

<!-- Redis Template Configuration-->

<bean id="metatemplate" class="org.springframework.data.redis.core.RedisTemplate"

<property name="connection-factory" ref="jedisConnFactory" />

</bean>

The above both configuration will result in creation of RedisTemplate. As with other template classes in Spring (e.g., JdbcTemplate, JmsTemplate) RedisTemplate is a helper class that simplifies Redis data access code. For more information about RedisTemplate and its variations (e.g., StringRedisTemplate) please refer to the Spring-Data-Redis documentation [here](http://projects.spring.io/spring-data-redis/).

Custom Connection Properties

Connection-related properties are missing in the above configuration. For example, the server address and port are missing in the configuration. In above case, we are using the defaults.

Configure Connection Properties:

@Bean

JedisConnectionFactory jedisConnectionFactory() {

JedisConnectionFactory jedisConFactory = new JedisConnectionFactory();

jedisConFactory.setHostName("localhost");

jedisConFactory.setPort(6379);

return jedisConFactory;

}

That's basically everything we need assuming we have single Redis server up and running on localhost with default configuration.

**Storing / Retrieving simple key-value pair**

Storing and retrieving a key/value pair is very simple: RedisTemplate provides various method to perform CRUD operations. Below snippet of code shows how can we use RedisTemplate’s opsForValue() to store/retrieve simple key-value pair.

@Autowired private RedisTemplate< String, Object > template;

public Object getValue( final String key ) {

return template.opsForValue().get( key );

}

public void setValue( final String key, final String value ) {

template.opsForValue().set( key, value );

}

Optionally, the key could be set to expire (yet another useful feature of Redis), f.e. let our keys expire in 1 second:

public void setValue( final String key, final String value ) {

template.opsForValue().set( key, value );

template.expire( key, 1, TimeUnit.SECONDS );

}

Arbitrary objects can also be saved into Redis as hashes (maps), for this we can use RedisTemplate’s opsForHash(), which in turn provide many methods to access & manipulate objects over data store. F.e. let’s save instance of some class User into Redis using key pattern "user:<id>":

User.java

public class User {

private final Long userId;

private String name;

private String email;

// Setters and getters are omitted for simplicity

}

UserServiceImpl.java

public void saveUser( final User user ) {

template.opsForHash().put(KEY, userBO.getUserId(), userBO);

}

Or, we can put each property separately in HashMap & store it as below:

public void setUser( final User user ) {

final String key = String.format( "user:%s", user.getUserId() );

final Map< String, Object > userMap = new HashMap< String, Object >();

userMap.put( "userid", user.getUserId() );

userMap.put( "name", user.getName() );

userMap.put( "email", user.getEmail() );

template.opsForHash().putAll( key, properties);

}

Respectively, object could easily be inspected and retrieved using the id.

public User getUserById( Long id ) {

return (User) template.opsForHash().get(KEY, userBO.getUserId());

}

Or, we can fetch each property separately & construct User object as:

public User getUserById( final Long id ) {

final String key = String.format( "user:%s", id );

final String name = ( String )template.opsForHash().get( key, "name" );

final String email = ( String )template.opsForHash().get( key, "email" );

return new User( id, name, email );

}

Respectively, we can perform other operations with simple key-value or any arbitrary object with help of RedisTemplate.

**Pub-Sub Messaging with Spring Data Redis**

Redis provides support for Publish-Subscribe messaging integrating with spring as JMS service of spring. For adding support to spring application below is the process to achieve using same spring-data-redis.

In Redis, publishers are not programmed to send their messages to specific subscribers. Rather, published messages are characterized into channels, without knowledge of what (if any) subscribers there may be.

Similarly, subscribers express interest in one or more topics and only receive messages that are of interest, without knowledge of what (if any) publishers there are. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology.

**Redis Configuration for Pub-Sub**

First, we’ll define a MessageListenerAdapter bean which contains a custom implementation of the MessageListener interface called RedisMessageSubscriber. This bean acts as a subscriber in the pub-sub messaging model:

@Bean

MessageListenerAdapter messageListener() {

return new MessageListenerAdapter(new RedisMessageSubscriber());

}

RedisMessageListenerContainer is a class provided by Spring Data Redis which provides asynchronous behavior for Redis message listeners. This is called internally and, according to the Spring Data Redis documentation – “handles the low level details of listening, converting and message dispatching.”

@Bean

RedisMessageListenerContainer redisContainer() {

RedisMessageListenerContainer container

= new RedisMessageListenerContainer();

container.setConnectionFactory(jedisConnectionFactory());

container.addMessageListener(messageListener(), topic());

return container;

}

Finally, we’ll set up a topic to which the publisher will send messages, and the subscriber will receive them:

@Bean

ChannelTopic topic() {

return new ChannelTopic("channel:user");

}

**Publishing Messages**

RedisTempalte : The template contains a very rich set of functions for wide range of operations – out of which convertAndSend is capable of sending a message to a queue through a topic:

public class RedisMessagePublisher implements MessagePublisher {

@Autowired

private RedisTemplate<String, Object> redisTemplate;

@Autowired

private ChannelTopic topic;

public void publish(String message) {

redisTemplate.convertAndSend(topic.getTopic(), message);

}

}

This publisher implementation is straightforward. It uses the convertAndSend() method of the redisTemplate to format and publish the given message to the configured topic. A topic implements publish and subscribe semantics: when a message is published, it goes to all the subscribers who are registered to listen on that topic.

We can publish messages to channel in other way also. Making use of template’s execute() method by passing RedisCallBack interface which in turns implements doInRedis() method to perform publish on RedisConnection. Here is code snippet:

@Autowired private RedisTemplate< String, Object > template;

public void publish( final String message ) {

template.execute( new RedisCallback< Long >() {

@Override

public Long doInRedis( RedisConnection connection ) throws DataAccessException {

return connection.publish(

new StringRedisSerializer().serialize("channelname"),

new StringRedisSerializer().serialize( message ) );

}

} );

}

**Subscribing to Messages**

A subscriber can implement MessageListener interface to become one. In our previous MessageListenerAdapter bean configuration we addes RedisMessageSubscriber which implements the Spring Data Redis-provided MessageListener interface. Upon receiving any message from subscribed topic, the implemented onMessage() method will be invoked.

@Service

public class RedisMessageSubscriber implements MessageListener {

public static List<String> messageList = new ArrayList<String>();

public void onMessage(Message message, byte[] pattern) {

messageList.add(message.toString());

System.out.println("Message received: " + message.toString());

}

}

Note: There is a second parameter called pattern, which we have not used in this example. The Spring Data Redis documentation states that this parameter represents the, “pattern matching the channel (if specified)”, but that it can be null.

When we call publish(message), the content is sent to Redis, where it is routed to the message queue topic defined in our publisher. Then it is distributed to the subscribers of that topic. Here RedisMessageSubscriber is a listener, which registers itself to the queue for retrieval of messages. On the arrival of the message, the subscriber’s onMessage() method defined triggered.

***For Reference:***

<http://docs.spring.io/spring-integration/reference/html/redis.html>

<http://www.baeldung.com/spring-data-redis-tutorial>

<https://redis.io/documentation>

<https://stackoverflow.com/questions/18376665/redis-distributed-or-not>