# Spring Data Redis Example

This page will walk through Spring Data Redis example. Spring Data provides API to perform Redis operations with ease. Redis is an open source, in memory data-structure store that can be used as database, cache and message broker. Redis supports data-structure such as strings, hashes, lists, sets etc. Redis is a NoSQL storage and uses key/value to store data.

Spring Data provides different connection factories to get Redis connections. The example of connection factories are JedisConnectionFactory, LettuceConnectionFactory etc. In our example we will use JedisConnectionFactory. Jedis is a Java Redis client that is easy to use and small in size.

Spring Data provides **RedisTemplate** to perform Redis operations. RedisTemplate has methods such as opsForList(), opsForSet(), opsForHash() etc. that return ListOperations, SetOperations, HashOperations respectively.

Most of the time we perform Redis operations with string datatype. So Spring Data has provided a string-focused extension as **StringRedisTemplate** that extends RedisTemplate. So StringRedisTemplate has all methods of RedisTemplate to perform Redis operations based on string datatype. In our example we will provide Spring Data and Redis configurations using JavaConfig as well as XML configurations with complete example.

## RedisTemplate

RedisTemplate is the central class to interact with the Redis data. It performs automatic serialization and deserialization between the given objects and binary data stored in Redis. By default RedisTemplate uses Java serialization. This class is thread-safe once configured.

RedisTemplate can be instantiated using different connection factory such as JedisConnectionFactory and LettuceConnectionFactory:

* JedisConnectionFactory uses JedisPoolConfig for connection pooling.
* LettuceConnectionFactory uses LettucePool for connection pooling. To work with Lettuce connector we need to add extra JAR.

In our example we will use JedisConnectionFactory to instantiate RedisTemplate. Jedis is a Java Redis client which is small in size and easy to use.   
RedisTemplate has different methods for data operations. Find some of them.

### ValueOperations<K,V> opsForValue()

opsForValue returns ValueOperations that performs operation on simple values. 

### ListOperations<K,V> opsForList()

opsForList() returns ListOperations that performs operation on list values. 

### SetOperations<K,V> opsForSet()

opsForSet() returns SetOperations that performs operation on set values. 

### HashOperations<K,HK,HV> opsForHash()

opsForHash() returns HashOperations that performs operation on hash values.   
  
If our most of the operation is string based then we can use StringRedisTemplate that is string-focused extension of RedisTemplate.

## Redis Java Configuration

JedisConnectionFactory uses host name as localhost and port as 6379 by default. To change these values use setHostName() and setPort() respectively.

To enable/disable using of connection pool, we need to pass Boolean values to setUsePool() method.

RedisTemplate does not provide transaction support by default. To enable it we need to call its method setEnableTransactionSupport(true)

We need to do same for StringRedisTemplate to enable transaction support.   
We will create demo project using JavaConfig as well as XML configuration. Here find the demo project structure using JavaConfig.

## Redis XML Configuration

Here we will provide XML configuration for Spring Data with Redis. We will configure JedisConnectionFactory with connection pooling, RedisTemplate and StringRedisTemplate using XML configuration. 

**spring-context.xml**

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

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

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

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<context:component-scan base-package="com.concretepage" />

<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig"

p:max-total="5" p:test-on-borrow="true" p:test-on-return="true"/>

<bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory"

p:host-name="localhost" p:port="6379" p:use-pool="true">

<constructor-arg ref="jedisPoolConfig"/>

</bean>

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

p:connection-factory-ref="jedisConnectionFactory" p:enable-transaction-support="true"/>

<bean id="stringRedisTemplate" class="org.springframework.data.redis.core.StringRedisTemplate"

p:connection-factory-ref="jedisConnectionFactory" p:enable-transaction-support="true"/>

</beans>

In our example we will create two demo project, one with JavaConfig and another with XML configuration. Here find the project structure using XML configuration.

## @Transactional Support

When we create any RedisTemplate bean then by default they will not support Spring @Transactional annotation. We need to enable it explicitly by calling setEnableTransactionSupport(true) for each and every RedisTemplate. We have also to enable transaction support for StringRedisTemplate explicitly by calling setEnableTransactionSupport(true). We enable transaction support using JavaConfig as following.

@Bean

public RedisTemplate<String, Person> redisTemplate() {

RedisTemplate<String, Person> redisTemplate = new RedisTemplate<>();

redisTemplate.setConnectionFactory(redisConnectionFactory());

redisTemplate.setEnableTransactionSupport(true);

return redisTemplate;

}

Find the code snippet of XML configuration to enable transaction support.

<bean id="stringRedisTemplate" class="org.springframework.data.redis.core.StringRedisTemplate"

p:connection-factory-ref="jedisConnectionFactory" p:enable-transaction-support="true"/>

Now we can use @Transactional as following.

@Repository

@Transactional

public class FriendDAO {

------

}

## ListOperations

It is used for Redis list specific operations. It is initialized as following.

@Autowired

private RedisTemplate<String, Person> redisTemplate;

ListOperations<String, Person> listOps = redisTemplate.opsForList();

We can also directly get the instance of ListOperations using javax.annotation.Resource annotation as following.

@Resource(name="redisTemplate")

ListOperations<String, Person> listOps;

Where **redisTemplate** is the name of RedisTemplate bean configured in JavaConfig or XML Configuration. Now find some of ListOperations methods.

* **leftPush(K key, V value)**: Prepends value to key.
* **rightPush(K key, V value)**: Appends value to key.
* **leftPop(K key)**: Removes and returns first element in list stored at key.
* **rightPop(K key)**: Removes and returns last element in list stored at key.
* **remove(K key, long count, Object value)**: Removes the first given number (count) of occurrences of value from the list stored at key.
* **index(K key, long index)**: Fetches element at index from list at key.
* **size(K key)**: Fetches the size of list stored at key.

Now find the example of ListOperations . Here we are performing create, read and delete operations.   
**FriendDAO.java**

package com.concretepage.dao;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.redis.core.RedisTemplate;

import org.springframework.stereotype.Repository;

import org.springframework.transaction.annotation.Transactional;

import com.concretepage.bean.Person;

@Repository

@Transactional

public class FriendDAO {

private static final String KEY = "friends";

@Autowired

private RedisTemplate<String, Person> redisTemplate;

public void addFriend(Person person) {

redisTemplate.opsForList().leftPush(KEY, person);

}

public long getNumberOfFriends() {

return redisTemplate.opsForList().size(KEY);

}

public Person getFriendAtIndex(Integer index) {

return redisTemplate.opsForList().index(KEY, index);

}

public void removeFriend(Person p) {

redisTemplate.opsForList().remove(KEY, 1, p);

}

}

Person.java

package com.concretepage.bean;

import java.io.Serializable;

public class Person implements Serializable {

private static final long serialVersionUID = 1L;

private int id;

private String name;

private int age;

public Person() { }

public Person(int id, String name, int age) {

this.id = id;

this.name = name;

this.age = age;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String toString() {

return id +" - " + name + " - " + age;

}

@Override

public boolean equals(final Object obj) {

if (obj == null) {

return false;

}

final Person person = (Person) obj;

if (this == person) {

return true;

} else {

return (this.name.equals(person.name) && this.age == person.age);

}

}

@Override

public int hashCode() {

int hashno = 7;

hashno = 13 \* hashno + (name == null ? 0 : name.hashCode());

return hashno;

}

}

Now we will test our FriendDAO.java. Find the code snippet.

System.out.println("--Example of RedisTemplate for ListOperations--");

FriendDAO friendDAO = context.getBean(FriendDAO.class);

Person p1 = new Person(1, "Mahesh", 30);

friendDAO.addFriend(p1);

Person p2 = new Person(2, "Krishna", 35);

friendDAO.addFriend(p2);

System.out.println("Number of friends: " + friendDAO.getNumberOfFriends());

System.out.println(friendDAO.getFriendAtIndex(1));

friendDAO.removeFriend(p2);

System.out.println(friendDAO.getFriendAtIndex(1)); //It will return null, because value is deleted.

Output 

--Example of RedisTemplate for ListOperations--

Number of friends: 2

1 - Mahesh - 30

null

## SetOperations

* SetOperations performs Redis set specific operations. Find some of its methods.   
  **add(K key, V... values)**: Adds values to set at key.
* **members(K key)**: Fetches all elements of set at key.
* **size(K key)**: Fetches size of set at key.
* **remove(K key, Object... values)**: Removes given values from set at key and returns the number of removed elements.

Now find the example of SetOperations. Here we will perform create, read and delete operations.   
**FamilyDAO.java**

package com.concretepage.dao;

import java.util.Set;

import javax.annotation.Resource;

import org.springframework.data.redis.core.SetOperations;

import org.springframework.stereotype.Repository;

import org.springframework.transaction.annotation.Transactional;

import com.concretepage.bean.Person;

@Repository

@Transactional

public class FamilyDAO {

private static final String KEY = "myfamily";

@Resource(name="redisTemplate")

private SetOperations<String, Person> setOps;

public void addFamilyMembers(Person... persons) {

setOps.add(KEY, persons);

}

public Set<Person> getFamilyMembers() {

return setOps.members(KEY);

}

public long getNumberOfFamilyMembers() {

return setOps.size(KEY);

}

public long removeFamilyMembers(Person... persons) {

return setOps.remove(KEY, (Object[])persons);

}

}

Find the code snippet to test the above code.

System.out.println("--Example of SetOperations--");

FamilyDAO familyDAO = context.getBean(FamilyDAO.class);

Person p11 = new Person(101, "Ram", 30);

Person p12 = new Person(102, "Lakshman", 25);

Person p13 = new Person(103, "Bharat", 35);

familyDAO.addFamilyMembers(p11, p12, p13);

System.out.println("Number of Family members: " + familyDAO.getNumberOfFamilyMembers());

System.out.println(familyDAO.getFamilyMembers());

System.out.println("No. of Removed Family Members: " + familyDAO.removeFamilyMembers(p11, p12));

System.out.println(familyDAO.getFamilyMembers());

Output 

--Example of SetOperations--

Number of Family members: 3

[101 - Ram - 30, 103 - Bharat - 35, 102 - Lakshman - 25]

No. of Removed Family Members: 2

[103 - Bharat - 35]

## HashOperations

HashOperations performs Redis map specific operations working on a hash. Find some of its methods.

* **putIfAbsent(H key, HK hashKey, HV value)**: Sets the value of a hash hashKey only if hashKey does not exist.
* **put(H key, HK hashKey, HV value)**: Sets the value of a hash hashKey.
* **get(H key, Object hashKey)**: Fetches value for given hashKey from hash at key.
* **size(H key)**: Fetches size of hash at key.
* **entries(H key)**: Fetches entire hash stored at key.
* **delete(H key, Object... hashKeys)**: Deletes given hash hashKeys at key.

Find the example of HashOperations with create, read, update and delete (CRUD) operations.   
**EmployeeDAO.java**

package com.concretepage.dao;

import java.util.Map;

import javax.annotation.Resource;

import org.springframework.data.redis.core.HashOperations;

import org.springframework.stereotype.Repository;

import org.springframework.transaction.annotation.Transactional;

import com.concretepage.bean.Person;

@Repository

@Transactional

public class EmployeeDAO {

private static final String KEY = "employees";

@Resource(name="redisTemplate")

private HashOperations<String, Integer, Person> hashOps;

public void addEmployee(Person person) {

hashOps.putIfAbsent(KEY, person.getId(), person);

}

public void updateEmployee(Person person) {

hashOps.put(KEY, person.getId(), person);

}

public Person getEmployee(Integer id) {

return hashOps.get(KEY, id);

}

public long getNumberOfEmployees() {

return hashOps.size(KEY);

}

public Map<Integer, Person> getAllEmployees() {

return hashOps.entries(KEY);

}

public long deleteEmployees(Integer... ids) {

return hashOps.delete(KEY, (Object)ids);

}

}

Find the code snippet to test above code.

System.out.println("--Example of HashOperations--");

EmployeeDAO empDAO = context.getBean(EmployeeDAO.class);

Person emp1 = new Person(11, "Ravan", 45);

Person emp2 = new Person(12, "Kumbhkarn", 35);

Person emp3 = new Person(13, "Vibhisan", 25);

empDAO.addEmployee(emp1);

empDAO.addEmployee(emp2);

empDAO.addEmployee(emp3);

System.out.println("No. of Employees: "+ empDAO.getNumberOfEmployees());

System.out.println(empDAO.getAllEmployees());

emp2.setAge(20);

empDAO.updateEmployee(emp2);

System.out.println(empDAO.getEmployee(12));

Output 

--Example of HashOperations--

No. of Employees: 3

{13=13 - Vibhisan - 25, 11=11 - Ravan - 45, 12=12 - Kumbhkarn - 35}

12 - Kumbhkarn - 20

## StringRedisTemplate

StringRedisTemplate is the string-focused extension of RedisTemplate. Most of the time we perform Redis operations with string and hence Spring Data provides a dedicated template i.e. StringRedisTemplate. Here we will discuss opsForValue() method of StringRedisTemplate that will return ValueOperations.

@Autowired

private StringRedisTemplate stringRedisTemplate;

ValueOperations<String, String> valueOps = StringRedisTemplate.opsForValue();

Find some of ValueOperations methods.

* **setIfAbsent(K key, V value)**: Sets key to hold the string value if key is absent.
* **set(K key, V value)**: Sets value for key.
* **get(Object key)**: Fetches the value of key.

Now find the example of CRUD operations using StringRedisTemplate.   
**UserDAO.java**

package com.concretepage.dao;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.redis.core.StringRedisTemplate;

import org.springframework.stereotype.Repository;

import org.springframework.transaction.annotation.Transactional;

@Repository

@Transactional

public class UserDAO {

private static final String KEY = "user";

@Autowired

private StringRedisTemplate stringRedisTemplate;

public void addUserName(String uname) {

stringRedisTemplate.opsForValue().setIfAbsent(KEY, uname);

}

public void updateUserName(String uname) {

stringRedisTemplate.opsForValue().set(KEY, uname);

}

public String getUserName() {

return stringRedisTemplate.opsForValue().get(KEY);

}

public void deleteUser() {

stringRedisTemplate.delete(KEY);

}

}

Find the code snippet to test the above code.

System.out.println("--Example of StringRedisTemplate--");

UserDAO userDAO = context.getBean(UserDAO.class);

userDAO.addUserName("sriram");

System.out.println(userDAO.getUserName());

userDAO.updateUserName("srikrishna");

System.out.println(userDAO.getUserName());

userDAO.deleteUser();

System.out.println(userDAO.getUserName()); //It will return null, because value is deleted.

Output 

--Example of StringRedisTemplate--

sriram

srikrishna

null

# Redis Commands

## Strings

You’ll remember from chapters 1 and 2 that STRINGs hold sequences of bytes, not significantly different from strings in many programming languages, or even C/C++– style char arrays. In Redis, STRINGs are used to store three types of values:

* Byte string values
* Integer values
* Floating-point values

Integers and floats can be incremented or decremented by an arbitrary numeric value (integers turning into floats as necessary). Integers have ranges that are equivalent to the platform’s long integer range (signed 32-bit integers on 32-bit platforms, and signed 64-bit integers on 64-bit platforms), and floats have ranges and values limited to IEEE 754 floating-point doubles. This three-way ability to look at the simplest of Redis values can be an advantage; it offers more flexibility in data representation than if only byte string values were allowed.

In this section, we’ll talk about the simplest structure available to Redis, the STRING. We’ll cover the basic numeric increment and decrement operations, followed later by the bit and substring manipulation calls, and you’ll come to understand that even the simplest of structures has a few surprises that can make it useful in a variety of powerful ways.

In table 3.1, you can see the available integer and float increment/decrement operations available on Redis STRINGs.

Table 3.1 Increment and decrement commands in Redis

|  |  |
| --- | --- |
| **Command** | **Example use and description** |
| INCR | INCR key-name — Increments the value stored at the key by 1 |
| DECR | DECR key-name — Decrements the value stored at the key by 1 |
| INCRBY | INCRBY key-name amount — Increments the value stored at the key by the provided integer value |
| DECRBY | DECRBY key-name amount — Decrements the value stored at the key by the provided integer value |
| INCRBYFLOAT | INCRBYFLOAT key-name amount — Increments the value stored at the key by the provided float value (available in Redis 2.6 and later) |

When setting a STRING value in Redis, if that value could be interpreted as a base-10 integer or a floating-point value, Redis will detect this and allow you to manipulate the value using the various INCR\* and DECR\*operations. If you try to increment or decrement a key that doesn’t exist or is an empty string, Redis will operate as though that key’s value were zero. If you try to increment or decrement a key that has a value that can’t be interpreted as an integer or float, you’ll receive an error. In the next listing, you can see some interactions with these commands.

Listing 3.1 A sample interaction showing INCR and DECR operations in Redis

>>> conn = redis.Redis()

>>> conn.get('key')

When we fetch a key that doesn’t exist, we get the None value, which isn’t displayed in the interactive console.

>>> conn.incr('key')

1

>>> conn.incr('key', 15)

16

We can increment keys that don’t exist, and we can pass an optional value to increment by more than 1.

>>> conn.decr('key', 5)

11

Like incrementing, decrementing takes an optional argument for the amount to decrement by.

>>> conn.get('key')

'11'

When we fetch the key, it acts like a string.

>>> conn.set('key', '13')

True

>>> conn.incr('key')

14

When we set the key, we can set it as a string, but still manipulate it like an integer.

After reading other chapters, you may notice that we really only call incr(). Internally, the Python Redis libraries call INCRBY with either the optional second value passed, or 1 if the value is omitted. As of this writing, the Python Redis client library supports the full command set of Redis 2.6, and offers INCRBYFLOAT support via an incrbyfloat() method that works the same as incr().

Redis additionally offers methods for reading and writing parts of byte string values (integer and float values can also be accessed as though they’re byte strings, though that use is somewhat uncommon). This can be useful if we were to use Redis STRING values to pack structured data in an efficient fashion, which we’ll talk about in chapter 9. Table 3.2 shows some methods that can be used to manipulate substrings and individual bits of STRINGs in Redis.

Table 3.2 Substring manipulation commands available to Redis

|  |  |
| --- | --- |
| **Command** | **Example use and description** |
| APPEND | APPEND key-name value — Concatenates the provided value to the string already stored at the given key |
| GETRANGE | GETRANGE key-name start end — Fetches the substring, including all characters from the start offset to the end offset, inclusive |
| SETRANGE | SETRANGE key-name offset value — Sets the substring starting at the provided offset to the given value |
| GETBIT | GETBIT key-name offset — Treats the byte string as a bit string, and returns the value of the bit in the string at the provided bit offset |
| SETBIT | SETBIT key-name offset value — Treats the byte string as a bit string, and sets the value of the bit in the string at the provided bit offset |
| BITCOUNT | BITCOUNT key-name [start end] — Counts the number of 1 bits in the string, optionally starting and finishing at the provided byte offsets |
| BITOP | BITOP operation dest-key key-name [key-name …] — Performs one of the bitwise operations, AND, OR, XOR, or NOT, on the strings provided, storing the result in the destination key |

GETRANGE AND SUBSTR In the past, GETRANGE was named SUBSTR, and the Python client continues to use the substr() method name to fetch ranges from the string. When using a version of Redis later than 2.6, you should use the getrange() method, and use substr() for Redis versions before 2.6.

When writing to strings using SETRANGE and SETBIT, if the STRING wasn’t previously long enough, Redis will automatically extend the STRING with nulls before updating and writing the new data. When reading STRINGs with GETRANGE, any request for data beyond the end of the STRING won’t be returned, but when reading bits with GETBIT, any bit beyond the end of the STRING is considered zero. In the following listing, you can see some uses of these STRING manipulation commands.

Listing 3.2 A sample interaction showing substring and bit operations in Redis

>>> conn.append('new-string-key', 'hello ')

Let’s append the string ‘hello ’ to the previously nonexistent key ‘new-string-key’.

6L

When appending a value, Redis returns the length of the string so far.

>>> conn.append('new-string-key', 'world!')

12L

When appending a value, Redis returns the length of the string so far.

>>> conn.substr('new-string-key', 3, 7)

Redis uses 0-indexing, and when accessing ranges, is inclusive of the endpoints by default.

'lo wo'

The string ‘lo wo’ is from the middle of ‘hello world!’

>>> conn.setrange('new-string-key', 0, 'H')

Let’s set a couple string ranges.

12

When setting a range inside a string, Redis also returns the total length of the string.

>>> conn.setrange('new-string-key', 6, 'W')

12

>>> conn.get('new-string-key')

Let’s see what we have now!

'Hello World!'

Yep, we capitalized our H and W.

>>> conn.setrange('new-string-key', 11, ', how are you?')

With setrange, we can replace anywhere inside the string, and we can make the string longer.

25

>>> conn.get('new-string-key')

'Hello World, how are you?'

We replace the exclamation point and add more to the end of the string.

>>> conn.setbit('another-key', 2, 1)

If we write to a bit beyond the size of the string, it’s filled with nulls.

0

Setting bits also returns the value of the bit before it was set.

>>> conn.setbit('another-key', 7, 1)

0

>>> conn.get('another-key')

If you want to interpret the bits stored in Redis, remember that offsets into bits are from the highest-order to the lowest-order.

'!'

We set bits 2 and 7 to 1, which gave us ‘!’, or character 33.

In many other key-value databases, data is stored as a plain string with no opportunities for manipulation. Some other key-value databases do allow you to prepend or append bytes, but Redis is unique in its ability to read and write substrings. In many ways, even if Redis only offered STRINGs and these methods to manipulate strings, Redis would be more powerful than many other systems; enterprising users could use the substring and bit manipulation calls along with WATCH/MULTI/EXEC (which we’ll briefly introduce in section 3.7.2, and talk about extensively in chapter 4) to build arbitrary data structures. In chapter 9, we’ll talk about using STRINGs to store a type of simple mappings that can greatly reduce memory use in some situations.

With a little work, we can store some types of sequences, but we’re limited in the kinds of manipulations we can perform. But if we use LISTs, we have a wider range of commands and ways to manipulate LIST items.

## Lists

As you may remember from chapter 1, LISTs allow you to push and pop items from both ends of a sequence, fetch individual items, and perform a variety of other operations that are expected of lists. LISTs by themselves can be great for keeping a queue of work items, recently viewed articles, or favorite contacts.

In this section, we’ll talk about LISTs, which store an ordered sequence of STRING values. We’ll cover some of the most commonly used LIST manipulation commands for pushing and popping items from LISTs. After reading this section, you’ll know how to manipulate LISTs using the most common commands. We’ll start by looking at table 3.3, where you can see some of the most frequently used LIST commands.

Table 3.3 Some commonly used LIST commands

|  |  |
| --- | --- |
| **Command** | **Example use and description** |
| RPUSH | RPUSH key-name value [value …] — Pushes the value(s) onto the right end of the list |
| LPUSH | LPUSH key-name value [value …] — Pushes the value(s) onto the left end of the list |
| RPOP | RPOP key-name — Removes and returns the rightmost item from the list |
| LPOP | LPOP key-name — Removes and returns the leftmost item from the list |
| LINDEX | LINDEX key-name offset — Returns the item at the given offset |
| LRANGE | LRANGE key-name start end — Returns the items in the list at the offsets from start to end, inclusive |
| LTRIM | LTRIM key-name start end — Trims the list to only include items at indices between start and end, inclusive |

The semantics of the LIST push commands shouldn’t be surprising, and neither should the pop commands. We covered a couple of these, along with both LINDEX and LRANGE, back in chapter 1. The next listing shows some uses of these push and pop commands.

Listing 3.3 A sample interaction showing LIST push and pop commands in Redis

>>> conn.rpush('list-key', 'last')

1L

When we push items onto the list, it returns the length of the list after the push has completed.

>>> conn.lpush('list-key', 'first')

We can easily push on both ends of the list.

2L

>>> conn.rpush('list-key', 'new last')

3L

>>> conn.lrange('list-key', 0, -1)

['first', 'last', 'new last']

Semantically, the left end of the list is the beginning, and the right end of the list is the end.

>>> conn.lpop('list-key')

'first'

>>> conn.lpop('list-key')

'last'

Popping off the left items repeatedly will return items from left to right.

>>> conn.lrange('list-key', 0, -1)

['new last']

>>> conn.rpush('list-key', 'a', 'b', 'c')

We can push multiple items at the same time.

4L

>>> conn.lrange('list-key', 0, -1)

['new last', 'a', 'b', 'c']

>>> conn.ltrim('list-key', 2, -1)

True

>>> conn.lrange('list-key', 0, -1)

['b', 'c']

We can trim any number of items from the start, end, or both.

The LTRIM command is new in this example, and we can combine it with LRANGE to give us something that functions much like an LPOP or RPOP call that returns and pops multiple items at once. We’ll talk more about how to make these kinds of composite commands atomic1 later in this chapter, as well as dive deeper into more advanced Redis-style transactions in chapter 4.

Among the LIST commands we didn’t introduce in chapter 1 are a few commands that allow you to move items from one list to another, and even block while waiting for other clients to add items to LISTs. Table 3.4 shows our blocking pop and item moving commands.

Table 3.4 Some LIST commands for blocking LIST pops and moving items between LISTs

|  |  |
| --- | --- |
| **Command** | **Example use and description** |
| BLPOP | BLPOP key-name [key-name …] timeout — Pops the leftmost item from the first non-empty LIST, or waits the timeout in seconds for an item |
| BRPOP | BRPOP key-name [key-name …] timeout — Pops the rightmost item from the first non-empty LIST, or waits the timeout in seconds for an item |
| RPOPLPUSH | RPOPLPUSH source-key dest-key — Pops the rightmost item from the source and LPUSHes the item to the destination, also returning the item to the user |
| BRPOPLPUSH | BRPOPLPUSH source-key dest-key timeout — Pops the rightmost item from the source and LPUSHes the item to the destination, also returning the item to the user, and waiting up to the timeout if the source is empty |

This set of commands is particularly useful when we talk about queues in chapter 6. The following listing shows some examples of moving items around with BRPOPLPUSH and popping items from multiple lists with BLPOP.

Listing 3.4 Blocking LIST pop and movement commands in Redis

>>> conn.rpush('list', 'item1')

1

>>> conn.rpush('list', 'item2')

2

>>> conn.rpush('list2', 'item3')

1

Let’s add some items to a couple of lists to start.

>>> conn.brpoplpush('list2', 'list', 1)

'item3'

Let’s move an item from one list to the other, also returning the item.

>>> conn.brpoplpush('list2', 'list', 1)

When a list is empty, the blocking pop will stall for the timeout, and return None (which isn’t displayed in the interactive console).

>>> conn.lrange('list', 0, -1)

['item3', 'item1', 'item2']

We popped the rightmost item from “list2” and pushed it to the left of “list”.

>>> conn.brpoplpush('list', 'list2', 1)

'item2'

>>> conn.blpop(['list', 'list2'], 1)

('list', 'item3')

>>> conn.blpop(['list', 'list2'], 1)

('list', 'item1')

>>> conn.blpop(['list', 'list2'], 1)

('list2', 'item2')

>>> conn.blpop(['list', 'list2'], 1)

Blocking left-popping items from these will check lists for items in the order that they are passed until they are empty.

>>>

## Sets

You’ll remember from chapter 1 that SETs hold unique items in an unordered fashion. You can quickly add, remove, and determine whether an item is in the SET. Among the many uses of SETs are storing who voted for an article and which articles belong to a specific group, as seen in chapter 1.

In this section, we’ll discuss some of the most frequently used commands that operate on SETs. You’ll learn about the standard operations for inserting, removing, and moving members between SETs, as well as commands to perform intersection, union, and differences on SETs. When finished with this section, you’ll be better prepared to fully understand how our search examples in chapter 7 work.

Let’s take a look at table 3.5 to see some of the more commonly used set commands.

Table 3.5 Some commonly used SET commands

|  |  |
| --- | --- |
| **Command** | **Example use and description** |
| SADD | SADD key-name item [item …] — Adds the items to the set and returns the number of items added that weren’t already present |
| SREM | SREM key-name item [item …] — Removes the items and returns the number of items that were removed |
| SISMEMBER | SISMEMBER key-name item — Returns whether the item is in the SET |
| SCARD | SCARD key-name — Returns the number of items in the SET |
| SMEMBERS | SMEMBERS key-name — Returns all of the items in the SET as a Python set |
| SRANDMEMBER | SRANDMEMBER key-name [count] — Returns one or more random items from the SET. When count is positive, Redis will return count distinct randomly chosen items, and when count is negative, Redis will return count randomly chosen items that may not be distinct. |
| SPOP | SPOP key-name — Removes and returns a random item from the SET |
| SMOVE | SMOVE source-key dest-key item — If the item is in the source, removes the item from the source and adds it to the destination, returning if the item was moved |

Some of those commands should be familiar from chapter 1, so let’s jump to the next listing to see some of these commands in action.

Listing 3.5 A sample interaction showing some common SET commands in Redis

>>> conn.sadd('set-key', 'a', 'b', 'c')

3

Adding items to the SET returns the number of items that weren’t already in the SET.

>>> conn.srem('set-key', 'c', 'd')

True

>>> conn.srem('set-key', 'c', 'd')

False

Removing items from the SET returns whether an item was removed; note that the client is buggy in that respect — Redis itself returns the total number of items removed.

>>> conn.scard('set-key')

2

We can get the number of items in the SET.

>>> conn.smembers('set-key')

set(['a', 'b'])

We can also fetch the whole SET.

>>> conn.smove('set-key', 'set-key2', 'a')

True

We can easily move items from one SET to another SET.

>>> conn.smove('set-key', 'set-key2', 'c')

False

>>> conn.smembers('set-key2')

set(['a'])

When an item doesn’t exist in the first set during a SMOVE, it isn’t added to the destination SET.

Using just these commands, we can keep track of unique events and items like we did in chapter 1 with voting and article groups. But the real power of SETs is in the commands that combine multiple SETs at the same time. Table 3.6 shows some of the ways that you can relate multiple SETs to each other.

Table 3.6 Operations for combining and manipulating SETs in Redis

|  |  |
| --- | --- |
| **Command** | **Example use and description** |
| SDIFF | SDIFF key-name [key-name …] — Returns the items in the first SET that weren’t in any of the other SETs (mathematical set difference operation) |
| SDIFFSTORE | SDIFFSTORE dest-key key-name [key-name …] — Stores at the dest-key the items in the first SET that weren’t in any of the other SETs (mathematical set difference operation) |
| SINTER | SINTER key-name [key-name …] — Returns the items that are in all of the SETs (mathematical set intersection operation) |
| SINTERSTORE | SINTERSTORE dest-key key-name [key-name …] — Stores at the dest-key the items that are in all of the SETs (mathematical set intersection operation) |
| SUNION | SUNION key-name [key-name …] — Returns the items that are in at least one of the SETs (mathematical set union operation) |
| SUNIONSTORE | SUNIONSTORE dest-key key-name [key-name …] — Stores at the dest-key the items that are in at least one of the SETs (mathematical set union operation) |

This group of commands are three fundamental SET operations, with both “return the result” and “store the result” versions. Let’s see a sample of what these commands are able to do.

Listing 3.6 A sample interaction showing SET difference, intersection, and union in Redis

>>> conn.sadd('skey1', 'a', 'b', 'c', 'd')

4

>>> conn.sadd('skey2', 'c', 'd', 'e', 'f')

4

First we’ll add a few items to a couple of SETs.

>>> conn.sdiff('skey1', 'skey2')

set(['a', 'b'])

We can calculate the result of removing all of the items in the second SET from the first SET.

>>> conn.sinter('skey1', 'skey2')

set(['c', 'd'])

We can also find out which items exist in both SETs.

>>> conn.sunion('skey1', 'skey2')

set(['a', 'c', 'b', 'e', 'd', 'f'])

And we can find out all of the items that are in either of the SETs.

If you’re comparing with Python sets, Redis SETs offer many of the same semantics and functionality, but are available remotely to potentially many clients. We’ll dig more deeply into what SETs are capable of in chapter 7, where we build a type of search engine with them.

Coming up next, we’ll talk about commands that manipulate HASHes, which allow us to group related keys and values together for easy fetching and updating.

## Hashes

As introduced in chapter 1, HASHes in Redis allow you to store groups of key-value pairs in a single higher-level Redis key. Functionally, the values offer some of the same features as values in STRINGs and can be useful to group related data together. This data grouping can be thought of as being similar to a row in a relational database or a document in a document store.

In this section, we’ll talk about the most commonly used commands that manipulate HASHes. You’ll learn more about the operations for adding and removing key-value pairs to HASHes, as well as commands to fetch all of the HASH contents along with the ability to increment or decrement values. When finished with this section, you’ll better understand the usefulness of storing your data in HASHes and how to do so. Look at table 3.7 to see some commonly used HASH commands.

Table 3.7 Operations for adding and removing items from HASHes

|  |  |
| --- | --- |
| **Command** | **Example use and description** |
| HMGET | HMGET key-name key [key …] — Fetches the values at the fields in the HASH |
| HMSET | HMSET key-name key value [key value …] — Sets the values of the fields in the HASH |
| HDEL | HDEL key-name key [key …] — Deletes the key-value pairs in the HASH, returning the number of pairs that were found and deleted |
| HLEN | HLEN key-name — Returns the number of key-value pairs in the HASH |

Some of those commands should be familiar from chapter 1, but we have a couple of new ones for getting and setting multiple keys at the same time. These bulk commands are mostly a matter of convenience and to improve Redis’s performance by reducing the number of calls and round trips between a client and Redis. Look at the next listing to see some of them in action.

Listing 3.7 A sample interaction showing some common HASH commands in Redis

>>> conn.hmset('hash-key', {'k1':'v1', 'k2':'v2', 'k3':'v3'})

True

We can add multiple items to the hash in one call.

>>> conn.hmget('hash-key', ['k2', 'k3'])

['v2', 'v3']

We can fetch a subset of the values in a single call.

>>> conn.hlen('hash-key')

3

The HLEN command is typically used for debugging very large HASHes.

>>> conn.hdel('hash-key', 'k1', 'k3')

True

The HDEL command handles multiple arguments without needing an HMDEL counterpart and returns True if any fields were removed.

The HMGET/HMSET commands are similar to their single-argument versions that we introduced in chapter 1, only differing in that they take a list or dictionary for arguments instead of the single entries.

Table 3.8 shows some other bulk commands and more STRING-like operations on HASHes.

With the availability of HGETALL, it may not seem as though HKEYS and HVALUES would be that useful, but when you expect your values to be large, you can fetch the keys, and then get the values one by one to keep from blocking other requests.

Table 3.8 More bulk operations and STRING-like calls over HASHes

|  |  |
| --- | --- |
| **Command** | **Example use and description** |
| HEXISTS | HEXISTS key-name key — Returns whether the given key exists in the HASH |
| HKEYS | HKEYS key-name — Fetches the keys in the HASH |
| HVALS | HVALS key-name — Fetches the values in the HASH |
| HGETALL | HGETALL key-name — Fetches all key-value pairs from the HASH |
| HINCRBY | HINCRBY key-name key increment — Increments the value stored at the given key by the integer increment |
| HINCRBYFLOAT | HINCRBYFLOAT key-name key increment — Increments the value stored at the given key by the float increment |

HINCRBY and HINCRBYFLOAT should remind you of the INCRBY and INCRBYFLOAT operations available on STRING keys, and they have the same semantics, applied to HASH values. Let’s look at some of these commands being used in the next listing.

Listing 3.8 A sample interaction showing some more advanced features of Redis HASHes

>>> conn.hmset('hash-key2', {'short':'hello', 'long':1000\*'1'})

True

>>> conn.hkeys('hash-key2')

['long', 'short']

Fetching keys can be useful to keep from needing to transfer large values when we’re looking into HASHes.

>>> conn.hexists('hash-key2', 'num')

False

We can also check the existence of specific keys.

>>> conn.hincrby('hash-key2', 'num')

1L

>>> conn.hexists('hash-key2', 'num')

True

Incrementing a previously nonexistent key in a hash behaves just like on strings; Redis operates as though the value had been 0.

As we described earlier, when confronted with a large value in a HASH, we can fetch the keys and only fetch values that we’re interested in to reduce the amount of data that’s transferred. We can also perform key checks, as we could perform member checks on SETs with SISMEMBER. And back in chapter 1, we used HINCRBY to keep track of the number of votes an article had received, which we just revisited.

Let’s look at a structure that we’ll be using fairly often in the remaining chapters: sorted sets.

## Sorted sets

ZSETs offer the ability to store a mapping of members to scores (similar to the keys and values of HASHes). These mappings allow us to manipulate the numeric scores,2 and fetch and scan over both members and scores based on the sorted order of the scores. In chapter 1, we showed a brief example that used ZSETs as a way of sorting submitted articles based on time and how many up-votes they had received, and in chapter 2, we had an example that used ZSETs as a way of handling the expiration of old cookies.

In this section, we’ll talk about commands that operate on ZSETs. You’ll learn how to add and update items in ZSETs, as well as how to use the ZSET intersection and union commands. When finished with this section, you’ll have a much clearer understanding about how ZSETs work, which will help you to better understand what we did with them in chapter 1, and how we’ll use them in chapters 5, 6, and 7.

Let’s look at some commonly used ZSET commands in table 3.9.

Table 3.9 Some common ZSET commands

|  |  |
| --- | --- |
| **Command** | **Example use and description** |
| ZADD | ZADD key-name score member [score member …] — Adds members with the given scores to the ZSET |
| ZREM | ZREM key-name member [member …] — Removes the members from the ZSET, returning the number of members that were removed |
| ZCARD | ZCARD key-name — Returns the number of members in the ZSET |
| ZINCRBY | ZINCRBY key-name increment member — Increments the member in the ZSET |
| ZCOUNT | ZCOUNT key-name min max — Returns the number of members with scores between the provided minimum and maximum |
| ZRANK | ZRANK key-name member — Returns the position of the given member in the ZSET |
| ZSCORE | ZSCORE key-name member — Returns the score of the member in the ZSET |
| ZRANGE | ZRANGE key-name start stop [WITHSCORES] — Returns the members and optionally the scores for the members with ranks between start and stop |

We’ve used some of these commands in chapters 1 and 2, so they should already be familiar to you. Let’s quickly revisit the use of some of our commands.

Listing 3.9 A sample interaction showing some common ZSET commands in Redis

>>> conn.zadd('zset-key', 'a', 3, 'b', 2, 'c', 1)

3

Adding members to ZSETs in Python has the arguments reversed compared to standard Redis, which makes the order the same as HASHes.

>>> conn.zcard('zset-key')

3

Knowing how large a ZSET is can tell us in some cases if it’s necessary to trim our ZSET.

>>> conn.zincrby('zset-key', 'c', 3)

4.0

We can also increment members like we can with STRING and HASH values.

>>> conn.zscore('zset-key', 'b')

2.0

Fetching scores of individual members can be useful if we’ve been keeping counters or toplists.

>>> conn.zrank('zset-key', 'c')

2

By fetching the 0-indexed position of a member, we can then later use ZRANGE to fetch a range of the values easily.

>>> conn.zcount('zset-key', 0, 3)

2L

Counting the number of items with a given range of scores can be quite useful for some tasks.

>>> conn.zrem('zset-key', 'b')

True

Removing members is as easy as adding them.

>>> conn.zrange('zset-key', 0, -1, withscores=True)

[('a', 3.0), ('c', 4.0)]

For debugging, we usually fetch the entire ZSET with this ZRANGE call, but real use cases will usually fetch items a relatively small group at a time.

You’ll likely remember our use of ZADD, ZREM, ZINCRBY, ZSCORE, and ZRANGE from chapters 1 and 2, so their semantics should come as no surprise. The ZCOUNT command is a little different than the others, primarily meant to let you discover the number of values whose scores are between the provided minimum and maximum scores.

Table 3.10 shows several more ZSET commands in Redis that you’ll find useful.

Table 3.10 Commands for fetching and deleting ranges of data from ZSETs and offering SET-like intersections

|  |  |
| --- | --- |
| **Command** | **Example use and description** |
| ZREVRANK | ZREVRANK key-name member — Returns the position of the member in the ZSET, with members ordered in reverse |
| ZREVRANGE | ZREVRANGE key-name start stop [WITHSCORES] — Fetches the given members from the ZSET by rank, with members in reverse order |
| ZRANGEBYSCORE | ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count] — Fetches the members between min and max |
| ZREVRANGEBYSCORE | ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count] — Fetches the members in reverse order between min and max |
| ZREMRANGEBYRANK | ZREMRANGEBYRANK key-name start stop — Removes the items from the ZSET with ranks between start and stop |
| ZREMRANGEBYSCORE | ZREMRANGEBYSCORE key-name min max — Removes the items from the ZSET with scores between min and max |
| ZINTERSTORE | ZINTERSTORE dest-key key-count key [key …] [WEIGHTS weight [weight …]] [AGGREGATE SUM|MIN|MAX] — Performs a SET-like intersection of the provided ZSETs |
| ZUNIONSTORE | ZUNIONSTORE dest-key key-count key [key …] [WEIGHTS weight [weight …]] [AGGREGATE SUM|MIN|MAX] — Performs a SET-like union of the provided ZSETs |

This is the first time that you’ve seen a few of these commands. If some of the ZREV\* commands are confusing, remember that they work the same as their nonreversed counterparts, except that the ZSETbehaves as if it were in reverse order (sorted by score from high to low). You can see a few examples of their use in the next listing.

Listing 3.10 A sample interaction showing ZINTERSTORE and ZUNIONSTORE

>>> conn.zadd('zset-1', 'a', 1, 'b', 2, 'c', 3)

3

>>> conn.zadd('zset-2', 'b', 4, 'c', 1, 'd', 0)

3

We’ll start out by creating a couple of ZSETs.

>>> conn.zinterstore('zset-i', ['zset-1', 'zset-2'])

2L

>>> conn.zrange('zset-i', 0, -1, withscores=True)

[('c', 4.0), ('b', 6.0)]

When performing ZINTERSTORE or ZUNIONSTORE, our default aggregate is sum, so scores of items that are in multiple ZSETs are added.

>>> conn.zunionstore('zset-u', ['zset-1', 'zset-2'], aggregate='min')

4L

>>> conn.zrange('zset-u', 0, -1, withscores=True)

[('d', 0.0), ('a', 1.0), ('c', 1.0), ('b', 2.0)]

It’s easy to provide different aggregates, though we’re limited to sum, min, and max.

>>> conn.sadd('set-1', 'a', 'd') 2

>>> conn.zunionstore('zset-u2', ['zset-1', 'zset-2', 'set-1']) 4L

>>> conn.zrange('zset-u2', 0, -1, withscores=True)

[('d', 1.0), ('a', 2.0), ('c', 4.0), ('b', 6.0)]

We can also pass SETs as inputs to ZINTERSTORE and ZUNIONSTORE; they behave as though they were ZSETs with all scores equal to 1.

ZSET union and intersection can be difficult to understand at first glance, so let’s look at some figures that show what happens during the processes of both intersection and union. Figure 3.1 shows the intersection of the two ZSETs and the final ZSET result. In this case, our aggregate is the default of sum, so scores are added.

Unlike intersection, when we perform a union operation, items that exist in at least one of the input ZSETs are included in the output. Figure 3.2 shows the result of performing a union operation with a different aggregate function, min, which takes the minimum score if a member is in multiple input ZSETs.

In chapter 1, we used the fact that we can include SETs as part of ZSET union and intersection operations. This feature allowed us to easily add and remove articles from groups without needing to propagate scoring and insertion times into additional ZSETs. Figure 3.3 shows a ZUNIONSTORE call that combines two ZSETs with one SET to produce a final ZSET.

Figure 3.1 What happens when calling conn.zinterstore(‘zset-i’, [‘zset-1’, ‘zset-2’]); elements that exist in both zset-1 and zset-2 are added together to get zset-iFigure 3.2 What happens when calling conn.zunionstore(‘zset-u’, [‘zset-1’, ‘zset-2′], aggregate=’min’); elements that exist in either zset-1 or zset-2 are combined with the minimum function to get zset-uFigure 3.3 What happens when calling conn.zunionstore(‘zset-u2’, [‘zset-1’, ‘zset-2’, ‘set-1’]); elements that exist in any of zset-1, zset-2, or set-1 are combined via addition to get zset-u2

In chapter 7, we’ll use ZINTERSTORE and ZUNIONSTORE as parts of a few different types of search. We’ll also talk about a few different ways to combine ZSET scores with the optional WEIGHTS parameter to further extend the types of problems that can be solved with SETs and ZSETs.

As you’re developing applications, you may have come upon a pattern known as publish/subscribe, also referred to as pub/sub. Redis includes this functionality, which we’ll cover next.

# Cache Abstraction

## Introduction

Since version 3.1, Spring Framework provides support for transparently adding caching into an existing Spring application. Similar to the [transaction](https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#transaction) support, the caching abstraction allows consistent use of various caching solutions with minimal impact on the code.

As from Spring 4.1, the cache abstraction has been significantly improved with the support of [JSR-107 annotations](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cache-jsr-107) and more customization options.

## Understanding the cache abstraction

Cache vs Buffer

The terms "buffer" and "cache" tend to be used interchangeably; note however they represent different things. A buffer is used traditionally as an intermediate temporary store for data between a fast and a slow entity. As one party would have to wait for the other affecting performance, the buffer alleviates this by allowing entire blocks of data to move at once rather then in small chunks. The data is written and read only once from the buffer. Furthermore, the buffers are visible to at least one party which is aware of it.

A cache on the other hand by definition is hidden and neither party is aware that caching occurs. It as well improves performance but does that by allowing the same data to be read multiple times in a fast fashion.

A further explanation of the differences between two can be found [here](https://en.wikipedia.org/wiki/Cache_(computing)#The_difference_between_buffer_and_cache).

At its core, the abstraction applies caching to Java methods, reducing thus the number of executions based on the information available in the cache. That is, each time a targeted method is invoked, the abstraction will apply a caching behavior checking whether the method has been already executed for the given arguments. If it has, then the cached result is returned without having to execute the actual method; if it has not, then method is executed, the result cached and returned to the user so that, the next time the method is invoked, the cached result is returned. This way, expensive methods (whether CPU or IO bound) can be executed only once for a given set of parameters and the result reused without having to actually execute the method again. The caching logic is applied transparently without any interference to the invoker.

Other cache-related operations are provided by the abstraction such as the ability to update the content of the cache or remove one of all entries. These are useful if the cache deals with data that can change during the course of the application.

Just like other services in the Spring Framework, the caching service is an abstraction (not a cache implementation) and requires the use of an actual storage to store the cache data - that is, the abstraction frees the developer from having to write the caching logic but does not provide the actual stores. This abstraction is materialized by the org.springframework.cache.Cache and org.springframework.cache.CacheManager interfaces.

There are [a few implementations](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cache-store-configuration) of that abstraction available out of the box: JDK java.util.concurrent.ConcurrentMap based caches, [Ehcache 2.x](http://ehcache.org/), Gemfire cache, [Caffeine](https://github.com/ben-manes/caffeine/wiki) and JSR-107 compliant caches (e.g. Ehcache 3.x). See [Plugging-in different back-end caches](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cache-plug) for more information on plugging in other cache stores/providers.

If you have a multi-process environment (i.e. an application deployed on several nodes), you will need to configure your cache provider accordingly. Depending on your use cases, a copy of the same data on several nodes may be enough but if you change the data during the course of the application, you may need to enable other propagation mechanisms.

Caching a particular item is a direct equivalent of the typical get-if-not-found-then- proceed-and-put-eventually code blocks found with programmatic cache interaction: no locks are applied and several threads may try to load the same item concurrently. The same applies to eviction: if several threads are trying to update or evict data concurrently, you may use stale data. Certain cache providers offer advanced features in that area, refer to the documentation of the cache provider that you are using for more details.

To use the cache abstraction, the developer needs to take care of two aspects:

* caching declaration - identify the methods that need to be cached and their policy
* cache configuration - the backing cache where the data is stored and read from

## Declarative annotation-based caching

For caching declaration, the abstraction provides a set of Java annotations:

* @Cacheable triggers cache population
* @CacheEvict triggers cache eviction
* @CachePut updates the cache without interfering with the method execution
* @Caching regroups multiple cache operations to be applied on a method
* @CacheConfig shares some common cache-related settings at class-level

Let us take a closer look at each annotation:

### @Cacheable annotation

As the name implies, @Cacheable is used to demarcate methods that are cacheable - that is, methods for whom the result is stored into the cache so on subsequent invocations (with the same arguments), the value in the cache is returned without having to actually execute the method. In its simplest form, the annotation declaration requires the name of the cache associated with the annotated method:

@Cacheable("books")

**public** Book findBook(ISBN isbn) {...}

In the snippet above, the method findBook is associated with the cache named books. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated. While in most cases, only one cache is declared, the annotation allows multiple names to be specified so that more than one cache are being used. In this case, each of the caches will be checked before executing the method - if at least one cache is hit, then the associated value will be returned:

@Cacheable({"books", "isbns"})

**public** Book findBook(ISBN isbn) {...}

#### Default Key Generation

Since caches are essentially key-value stores, each invocation of a cached method needs to be translated into a suitable key for cache access. Out of the box, the caching abstraction uses a simple KeyGenerator based on the following algorithm:

* If no params are given, return SimpleKey.EMPTY.
* If only one param is given, return that instance.
* If more the one param is given, return a SimpleKey containing all parameters.

This approach works well for most use-cases; As long as parameters have natural keys and implement valid hashCode()and equals() methods. If that is not the case then the strategy needs to be changed.

To provide a different default key generator, one needs to implement the org.springframework.cache.interceptor.KeyGenerator interface.

|  |
| --- |
| The default key generation strategy changed with the release of Spring 4.0. Earlier versions of Spring used a key generation strategy that, for multiple key parameters, only considered the hashCode() of parameters and not equals(); this could cause unexpected key collisions (see [SPR-10237](https://jira.spring.io/browse/SPR-10237) for background). The new 'SimpleKeyGenerator' uses a compound key for such scenarios.  If you want to keep using the previous key strategy, you can configure the deprecated org.springframework.cache.interceptor.DefaultKeyGenerator class or create a custom hash-based 'KeyGenerator' implementation. |

#### Custom Key Generation Declaration

Since caching is generic, it is quite likely the target methods have various signatures that cannot be simply mapped on top of the cache structure. This tends to become obvious when the target method has multiple arguments out of which only some are suitable for caching (while the rest are used only by the method logic). For example:

@Cacheable("books")

**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

At first glance, while the two boolean arguments influence the way the book is found, they are no use for the cache. Further more what if only one of the two is important while the other is not?

For such cases, the @Cacheable annotation allows the user to specify how the key is generated through its key attribute. The developer can use [SpEL](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions) to pick the arguments of interest (or their nested properties), perform operations or even invoke arbitrary methods without having to write any code or implement any interface. This is the recommended approach over the [default generator](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cache-annotations-cacheable-default-key) since methods tend to be quite different in signatures as the code base grows; while the default strategy might work for some methods, it rarely does for all methods.

Below are some examples of various SpEL declarations - if you are not familiar with it, do yourself a favor and read [Spring Expression Language](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions):

@Cacheable(cacheNames="books", **key="#isbn"**)

**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

@Cacheable(cacheNames="books", **key="#isbn.rawNumber"**)

**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

@Cacheable(cacheNames="books", **key="T(someType).hash(#isbn)"**)

**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

The snippets above show how easy it is to select a certain argument, one of its properties or even an arbitrary (static) method.

If the algorithm responsible to generate the key is too specific or if it needs to be shared, you may define a custom keyGenerator on the operation. To do this, specify the name of the KeyGenerator bean implementation to use:

@Cacheable(cacheNames="books", **keyGenerator="myKeyGenerator"**)

**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

#### Default Cache Resolution

Out of the box, the caching abstraction uses a simple CacheResolver that retrieves the cache(s) defined at the operation level using the configured CacheManager.

To provide a different default cache resolver, one needs to implement the org.springframework.cache.interceptor.CacheResolver interface.

#### Custom cache resolution

The default cache resolution fits well for applications working with a single CacheManager and with no complex cache resolution requirements.

For applications working with several cache managers, it is possible to set the cacheManager to use per operation:

@Cacheable(cacheNames="books",**cacheManager="anotherCacheManager”**)

**public** Book findBook(ISBN isbn) {...}

It is also possible to replace the CacheResolver entirely in a similar fashion as for [key generation](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cache-annotations-cacheable-key). The resolution is requested for every cache operation, giving a chance to the implementation to actually resolve the cache(s) to use based on runtime arguments:

@Cacheable(**cacheResolver="runtimeCacheResolver"**)

**public** Book findBook(ISBN isbn) {...}

#### Synchronized caching

In a multi-threaded environment, certain operations might be concurrently invoked for the same argument (typically on startup). By default, the cache abstraction does not lock anything and the same value may be computed several times, defeating the purpose of caching.

For those particular cases, the sync attribute can be used to instruct the underlying cache provider to lock the cache entry while the value is being computed. As a result, only one thread will be busy computing the value while the others are blocked until the entry is updated in the cache.

@Cacheable(cacheNames="foos", **sync=true**)

**public** Foo executeExpensiveOperation(String id) {...}

#### Conditional caching

Sometimes, a method might not be suitable for caching all the time (for example, it might depend on the given arguments). The cache annotations support such functionality through the condition parameter which takes a SpELexpression that is evaluated to either true or false. If true, the method is cached - if not, it behaves as if the method is not cached, that is executed every time no matter what values are in the cache or what arguments are used. A quick example - the following method will be cached only if the argument name has a length shorter than 32:

@Cacheable(cacheNames="book", **condition="#name.length() < 32"**)

**public** Book findBook(String name)

In addition the condition parameter, the unless parameter can be used to veto the adding of a value to the cache. Unlike condition, unless expressions are evaluated after the method has been called. Expanding on the previous example - perhaps we only want to cache paperback books:

@Cacheable(cacheNames="book", condition="#name.length() < 32", **unless="#result.hardback"**)

**public** Book findBook(String name)

The cache abstraction supports java.util.Optional, using its content as cached value only if it present. #result always refers to the business entity and never on a supported wrapper so the previous example can be rewritten as follows:

@Cacheable(cacheNames="book", condition="#name.length() < 32", **unless="#result?.hardback"**)

**public** Optional<Book> findBook(String name)

Note that result still refers to Book and not Optional. As it might be null, we should use the safe navigation operator.

#### Available caching SpEL evaluation context

Each SpEL expression evaluates again a dedicated [context](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions-language-ref). In addition to the build in parameters, the framework provides dedicated caching related metadata such as the argument names. The next table lists the items made available to the context so one can use them for key and conditional computations:

| Table 11. Cache SpEL available metadata | | | |
| --- | --- | --- | --- |
| Name | Location | Description | Example |
| methodName | root object | The name of the method being invoked | #root.methodName |
| method | root object | The method being invoked | #root.method.name |
| target | root object | The target object being invoked | #root.target |
| targetClass | root object | The class of the target being invoked | #root.targetClass |
| args | root object | The arguments (as array) used for invoking the target | #root.args[0] |
| caches | root object | Collection of caches against which the current method is executed | #root.caches[0].name |
| argument name | evaluation context | Name of any of the method arguments. If for some reason the names are not available (e.g. no debug information), the argument names are also available under the #a<#arg> where #argstands for the argument index (starting from 0). | #iban or #a0 (one can also use #p0 or #p<#arg>notation as an alias). |
| result | evaluation context | The result of the method call (the value to be cached). Only available in unless expressions, cache put expressions (to compute the key), or cache evict expressions (when beforeInvocation is false). For supported wrappers such asOptional, #result refers to the actual object, not the wrapper. | #result |

### @CachePut annotation

For cases where the cache needs to be updated without interfering with the method execution, one can use the @CachePut annotation. That is, the method will always be executed and its result placed into the cache (according to the @CachePut options). It supports the same options as @Cacheable and should be used for cache population rather than method flow optimization:

@CachePut(cacheNames="book", key="#isbn")

**public** Book updateBook(ISBN isbn, BookDescriptor descriptor)

|  |  |
| --- | --- |
|  | Note that using @CachePut and @Cacheable annotations on the same method is generally strongly discouraged because they have different behaviors. While the latter causes the method execution to be skipped by using the cache, the former forces the execution in order to execute a cache update. This leads to unexpected behavior and with the exception of specific corner-cases (such as annotations having conditions that exclude them from each other), such declaration should be avoided. Note also that such condition should not rely on the result object (i.e. the #result variable) as these are validated upfront to confirm the exclusion. |

### @CacheEvict annotation

The cache abstraction allows not just population of a cache store but also eviction. This process is useful for removing stale or unused data from the cache. Opposed to @Cacheable, annotation @CacheEvict demarcates methods that perform cache eviction, that is methods that act as triggers for removing data from the cache. Just like its sibling, @CacheEvict requires specifying one (or multiple) caches that are affected by the action, allows a custom cache and key resolution or a condition to be specified but in addition, features an extra parameter allEntries which indicates whether a cache-wide eviction needs to be performed rather then just an entry one (based on the key):

@CacheEvict(cacheNames="books", **allEntries=true**)

**public** **void** loadBooks(InputStream batch)

This option comes in handy when an entire cache region needs to be cleared out - rather then evicting each entry (which would take a long time since it is inefficient), all the entries are removed in one operation as shown above. Note that the framework will ignore any key specified in this scenario as it does not apply (the entire cache is evicted not just one entry).

One can also indicate whether the eviction should occur after (the default) or before the method executes through the beforeInvocation attribute. The former provides the same semantics as the rest of the annotations - once the method completes successfully, an action (in this case eviction) on the cache is executed. If the method does not execute (as it might be cached) or an exception is thrown, the eviction does not occur. The latter ( beforeInvocation=true) causes the eviction to occur always, before the method is invoked - this is useful in cases where the eviction does not need to be tied to the method outcome.

It is important to note that void methods can be used with @CacheEvict - as the methods act as triggers, the return values are ignored (as they don’t interact with the cache) - this is not the case with @Cacheable which adds/updates data into the cache and thus requires a result.

### @Caching annotation

There are cases when multiple annotations of the same type, such as @CacheEvict or @CachePut need to be specified, for example because the condition or the key expression is different between different caches. @Caching allows multiple nested @Cacheable, @CachePut and @CacheEvict to be used on the same method:

@Caching(evict = { @CacheEvict("primary"), @CacheEvict(cacheNames="secondary", key="#p0") })

**public** Book importBooks(String deposit, Date date)

### @CacheConfig annotation

So far we have seen that caching operations offered many customization options and these can be set on an operation basis. However, some of the customization options can be tedious to configure if they apply to all operations of the class. For instance, specifying the name of the cache to use for every cache operation of the class could be replaced by a single class-level definition. This is where @CacheConfig comes into play.

**@CacheConfig("books")**

**public** **class** **BookRepositoryImpl** **implements** BookRepository {

@Cacheable

**public** Book findBook(ISBN isbn) {...}

}

@CacheConfig is a class-level annotation that allows to share the cache names, the custom KeyGenerator, the custom CacheManager and finally the custom CacheResolver. Placing this annotation on the class does not turn on any caching operation.

An operation-level customization will always override a customization set on @CacheConfig. This gives therefore three levels of customizations per cache operation:

* Globally configured, available for CacheManager, KeyGenerator
* At class level, using @CacheConfig
* At the operation level

### Enable caching annotations

It is important to note that even though declaring the cache annotations does not automatically trigger their actions - like many things in Spring, the feature has to be declaratively enabled (which means if you ever suspect caching is to blame, you can disable it by removing only one configuration line rather than all the annotations in your code).

To enable caching annotations add the annotation @EnableCaching to one of your @Configuration classes:

@Configuration

@EnableCaching

**public** **class** **AppConfig** {

}

Alternatively for XML configuration use the cache:annotation-driven element:

<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"

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">

<cache:annotation-driven/>

</beans>

Both the cache:annotation-driven element and @EnableCaching annotation allow various options to be specified that influence the way the caching behavior is added to the application through AOP. The configuration is intentionally similar with that of [@Transactional](https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#tx-annotation-driven-settings):

|  |
| --- |
| The default advice mode for processing caching annotations is "proxy" which allows for interception of calls through the proxy only; local calls within the same class cannot get intercepted that way. For a more advanced mode of interception, consider switching to "aspectj" mode in combination with compile-time or load-time weaving. |
| Advanced customizations using Java config require to implement CachingConfigurer: Please refer to [the javadoc for more details](https://docs.spring.io/spring-framework/docs/5.0.8.RELEASE/javadoc-api/org/springframework/cache/annotation/CachingConfigurer.html). |

| Table 12. Cache annotation settings | | | |
| --- | --- | --- | --- |
| XML Attribute | Annotation Attribute | Default | Description |
| cache-manager | N/A (See CachingConfigurerjavadocs) | cacheManager | Name of cache manager to use. A default CacheResolver will be initialized behind the scenes with this cache manager (or `cacheManager`if not set). For more fine-grained management of the cache resolution, consider setting the 'cache-resolver' attribute. |
| cache-resolver | N/A (See CachingConfigurerjavadocs) | A SimpleCacheResolver using the configured cacheManager. | The bean name of the CacheResolver that is to be used to resolve the backing caches. This attribute is not required, and only needs to be specified as an alternative to the 'cache-manager' attribute. |
| key-generator | N/A (See CachingConfigurerjavadocs) | SimpleKeyGenerator | Name of the custom key generator to use. |
| error-handler | N/A (See CachingConfigurerjavadocs) | SimpleCacheErrorHandler | Name of the custom cache error handler to use. By default, any exception throw during a cache related operations are thrown back at the client. |
| mode | mode | proxy | The default mode "proxy" processes annotated beans to be proxied using Spring’s AOP framework (following proxy semantics, as discussed above, applying to method calls coming in through the proxy only). The alternative mode "aspectj" instead weaves the affected classes with Spring’s AspectJ caching aspect, modifying the target class byte code to apply to any kind of method call. AspectJ weaving requires spring-aspects.jar in the classpath as well as load-time weaving (or compile-time weaving) enabled. (See[Spring configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw-spring) for details on how to set up load-time weaving.) |
| proxy-target-class | proxyTargetClass | false | Applies to proxy mode only. Controls what type of caching proxies are created for classes annotated with the @Cacheable or @CacheEvict annotations. If the proxy-target-class attribute is set to true, then class-based proxies are created. If proxy-target-class is false or if the attribute is omitted, then standard JDK interface-based proxies are created. (See [Proxying mechanisms](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-proxying) for a detailed examination of the different proxy types.) |
| order | order | Ordered.LOWEST\_PRECEDENCE | Defines the order of the cache advice that is applied to beans annotated with@Cacheable or @CacheEvict. (For more information about the rules related to ordering of AOP advice, see [Advice ordering](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj-advice-ordering).) No specified ordering means that the AOP subsystem determines the order of the advice. |

|  |  |
| --- | --- |
|  | <cache:annotation-driven/> only looks for @Cacheable/@CachePut/@CacheEvict/@Caching on beans in the same application context it is defined in. This means that, if you put <cache:annotation-driven/> in a WebApplicationContext for a DispatcherServlet, it only checks for beans in your controllers, and not your services. See [the MVC section](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-servlet) for more information. |

Method visibility and cache annotations

When using proxies, you should apply the cache annotations only to methods with public visibility. If you do annotate protected, private or package-visible methods with these annotations, no error is raised, but the annotated method does not exhibit the configured caching settings. Consider the use of AspectJ (see below) if you need to annotate non-public methods as it changes the bytecode itself.

|  |  |
| --- | --- |
|  | Spring recommends that you only annotate concrete classes (and methods of concrete classes) with the @Cache\* annotation, as opposed to annotating interfaces. You certainly can place the @Cache\*annotation on an interface (or an interface method), but this works only as you would expect it to if you are using interface-based proxies. The fact that Java annotations are not inherited from interfaces means that if you are using class-based proxies ( proxy-target-class="true") or the weaving-based aspect ( mode="aspectj"), then the caching settings are not recognized by the proxying and weaving infrastructure, and the object will not be wrapped in a caching proxy, which would be decidedly bad. |
|  | In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation, in effect, a method within the target object calling another method of the target object, will not lead to an actual caching at runtime even if the invoked method is marked with @Cacheable - considering using the aspectj mode in this case. Also, the proxy must be fully initialized to provide the expected behaviour so you should not rely on this feature in your initialization code, i.e. @PostConstruct. |

### Using custom annotations

Custom annotation and AspectJ

This feature only works out-of-the-box with the proxy-based approach but can be enabled with a bit of extra effort using AspectJ.

The spring-aspects module defines an aspect for the standard annotations only. If you have defined your own annotations, you also need to define an aspect for those. Check AnnotationCacheAspect for an example.

The caching abstraction allows you to use your own annotations to identify what method triggers cache population or eviction. This is quite handy as a template mechanism as it eliminates the need to duplicate cache annotation declarations (especially useful if the key or condition are specified) or if the foreign imports (org.springframework) are not allowed in your code base. Similar to the rest of the [stereotype](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-stereotype-annotations) annotations, @Cacheable, @CachePut, @CacheEvictand @CacheConfig can be used as [meta-annotations](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-meta-annotations), that is annotations that can annotate other annotations. To wit, let us replace a common @Cacheable declaration with our own, custom annotation:

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.METHOD})

@Cacheable(cacheNames="books", key="#isbn")

**public** @interface SlowService {

}

Above, we have defined our own SlowService annotation which itself is annotated with @Cacheable - now we can replace the following code:

@Cacheable(cacheNames="books", key="#isbn")

**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

with:

@SlowService

**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

Even though @SlowService is not a Spring annotation, the container automatically picks up its declaration at runtime and understands its meaning. Note that as mentioned [above](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cache-annotation-enable), the annotation-driven behavior needs to be enabled.

## JCache (JSR-107) annotations

Since the Spring Framework 4.1, the caching abstraction fully supports the JCache standard annotations:

these are @CacheResult, @CachePut, @CacheRemove and @CacheRemoveAll as well as the @CacheDefaults, @CacheKey and @CacheValue companions.

These annotations can be used right the way without migrating your cache store to JSR-107: the internal implementation uses Spring’s caching abstraction and provides default CacheResolver and KeyGenerator implementations that are compliant with the specification. In other words, if you are already using Spring’s caching abstraction, you can switch to these standard annotations without changing your cache storage (or configuration, for that matter).

### Feature summary

For those who are familiar with Spring’s caching annotations, the following table describes the main differences between the Spring annotations and the JSR-107 counterpart:

| Table 13. Spring vs. JSR-107 caching annotations | | |
| --- | --- | --- |
| Spring | JSR-107 | Remark |
| @Cacheable | @CacheResult | Fairly similar. @CacheResult can cache specific exceptions and force the execution of the method regardless of the content of the cache. |
| @CachePut | @CachePut | While Spring updates the cache with the result of the method invocation, JCache requires to pass it as an argument that is annotated with @CacheValue. Due to this difference, JCache allows to update the cache before or after the actual method invocation. |
| @CacheEvict | @CacheRemove | Fairly similar. @CacheRemove supports a conditional evict in case the method invocation results in an exception. |
| @CacheEvict(allEntries=true) | @CacheRemoveAll | See @CacheRemove. |
| @CacheConfig | @CacheDefaults | Allows to configure the same concepts, in a similar fashion. |

JCache has the notion of javax.cache.annotation.CacheResolver that is identical to the Spring’s CacheResolverinterface, except that JCache only supports a single cache. By default, a simple implementation retrieves the cache to use based on the name declared on the annotation. It should be noted that if no cache name is specified on the annotation, a default is automatically generated, check the javadoc of @CacheResult#cacheName() for more information.

CacheResolver instances are retrieved by a CacheResolverFactory. It is possible to customize the factory per cache operation:

@CacheResult(cacheNames="books", **cacheResolverFactory=MyCacheResolverFactory.class**)

**public** Book findBook(ISBN isbn)

|  |  |
| --- | --- |
|  | For all referenced classes, Spring tries to locate a bean with the given type. If more than one match exists, a new instance is created and can use the regular bean lifecycle callbacks such as dependency injection. |

Keys are generated by a javax.cache.annotation.CacheKeyGenerator that serves the same purpose as Spring’s KeyGenerator. By default, all method arguments are taken into account unless at least one parameter is annotated with @CacheKey. This is similar to Spring’s [custom key generation declaration](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cache-annotations-cacheable-key). For instance these are identical operations, one using Spring’s abstraction and the other with JCache:

@Cacheable(cacheNames="books", **key="#isbn"**)

**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

@CacheResult(cacheName="books")

**public** Book findBook(**@CacheKey** ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

The CacheKeyResolver to use can also be specified on the operation, in a similar fashion as the CacheResolverFactory.

JCache can manage exceptions thrown by annotated methods: this can prevent an update of the cache but it can also cache the exception as an indicator of the failure instead of calling the method again. Let’s assume that InvalidIsbnNotFoundException is thrown if the structure of the ISBN is invalid. This is a permanent failure, no book could ever be retrieved with such parameter. The following caches the exception so that further calls with the same, invalid ISBN, throws the cached exception directly instead of invoking the method again.

@CacheResult(cacheName="books", **exceptionCacheName="failures"**

**cachedExceptions = InvalidIsbnNotFoundException.class**)

**public** Book findBook(ISBN isbn)

### Enabling JSR-107 support

Nothing specific needs to be done to enable the JSR-107 support alongside Spring’s declarative annotation support. Both @EnableCaching and the cache:annotation-driven element will enable automatically the JCache support if both the JSR-107 API and the spring-context-support module are present in the classpath.

## Declarative XML-based caching

If annotations are not an option (no access to the sources or no external code), one can use XML for declarative caching. So instead of annotating the methods for caching, one specifies the target method and the caching directives externally (similar to the declarative transaction management [advice](https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#transaction-declarative-first-example)). The previous example can be translated into:

*<!-- the service we want to make cacheable -->*

<bean id="bookService" class="x.y.service.DefaultBookService"/>

*<!-- cache definitions -->*

<cache:advice id="cacheAdvice" cache-manager="cacheManager">

<cache:caching cache="books">

<cache:cacheable method="findBook" key="#isbn"/>

<cache:cache-evict method="loadBooks" all-entries="true"/>

</cache:caching>

</cache:advice>

*<!-- apply the cacheable behavior to all BookService interfaces -->*

<aop:config>

<aop:advisor advice-ref="cacheAdvice" pointcut="execution(\* x.y.BookService.\*(..))"/>

</aop:config>

*<!-- cache manager definition omitted -->*

In the configuration above, the bookService is made cacheable. The caching semantics to apply are encapsulated in the cache:advice definition which instructs method findBooks to be used for putting data into the cache while method loadBooks for evicting data. Both definitions are working against the books cache.

The aop:config definition applies the cache advice to the appropriate points in the program by using the AspectJ pointcut expression (more information is available in [Aspect Oriented Programming with Spring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop)). In the example above, all methods from the BookService are considered and the cache advice applied to them.

The declarative XML caching supports all of the annotation-based model so moving between the two should be fairly easy - further more both can be used inside the same application. The XML based approach does not touch the target code however it is inherently more verbose; when dealing with classes with overloaded methods that are targeted for caching, identifying the proper methods does take an extra effort since the method argument is not a good discriminator - in these cases, the AspectJ pointcut can be used to cherry pick the target methods and apply the appropriate caching functionality. However through XML, it is easier to apply a package/group/interface-wide caching (again due to the AspectJ pointcut) and to create template-like definitions (as we did in the example above by defining the target cache through the cache:definitions cache attribute).

## Configuring the cache storage

Out of the box, the cache abstraction provides several storage integration. To use them, one needs to simply declare an appropriate CacheManager - an entity that controls and manages Caches and can be used to retrieve these for storage.

### JDK ConcurrentMap-based Cache

The JDK-based Cache implementation resides under org.springframework.cache.concurrent package. It allows one to use ConcurrentHashMap as a backing Cache store.

*<!-- simple cache manager -->*

<bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">

<property name="caches">

<set>

<bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name="default"/>

<bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name="books"/>

</set>

</property>

</bean>

The snippet above uses the SimpleCacheManager to create a CacheManager for the two nested ConcurrentMapCacheinstances named default and books. Note that the names are configured directly for each cache.

As the cache is created by the application, it is bound to its lifecycle, making it suitable for basic use cases, tests or simple applications. The cache scales well and is very fast but it does not provide any management or persistence capabilities nor eviction contracts.

### Ehcache-based Cache

|  |  |
| --- | --- |
|  | Ehcache 3.x is fully JSR-107 compliant and no dedicated support is required for it. |

The Ehcache 2.x implementation is located under org.springframework.cache.ehcache package. Again, to use it, one simply needs to declare the appropriate CacheManager:

<bean id="cacheManager"

class="org.springframework.cache.ehcache.EhCacheCacheManager" p:cache-manager-ref="ehcache"/>

*<!-- EhCache library setup -->*

<bean id="ehcache"

class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean" p:config-location="ehcache.xml"/>

This setup bootstraps the ehcache library inside Spring IoC (through the ehcache bean) which is then wired into the dedicated CacheManager implementation. Note the entire ehcache-specific configuration is read from ehcache.xml.

### Caffeine Cache

Caffeine is a Java 8 rewrite of Guava’s cache and its implementation is located underorg.springframework.cache.caffeine package and provides access to several features of Caffeine.

Configuring a CacheManager that creates the cache on demand is straightforward:

<bean id="cacheManager"

class="org.springframework.cache.caffeine.CaffeineCacheManager"/>

It is also possible to provide the caches to use explicitly. In that case, only those will be made available by the manager:

<bean id="cacheManager" class="org.springframework.cache.caffeine.CaffeineCacheManager">

<property name="caches">

<set>

<value>default</value>

<value>books</value>

</set>

</property>

</bean>

The Caffeine CacheManager also supports customs Caffeine and CacheLoader. See the [Caffeine documentation](https://github.com/ben-manes/caffeine/wiki) for more information about those.

### GemFire-based Cache

GemFire is a memory-oriented/disk-backed, elastically scalable, continuously available, active (with built-in pattern-based subscription notifications), globally replicated database and provides fully-featured edge caching. For further information on how to use GemFire as a CacheManager (and more), please refer to the [Spring Data GemFire reference documentation](https://docs.spring.io/spring-gemfire/docs/current/reference/html/).

### JSR-107 Cache

JSR-107 compliant caches can also be used by Spring’s caching abstraction. The JCache implementation is located under org.springframework.cache.jcache package.

Again, to use it, one simply needs to declare the appropriate CacheManager:

<bean id="cacheManager"

class="org.springframework.cache.jcache.JCacheCacheManager"

p:cache-manager-ref="jCacheManager"/>

*<!-- JSR-107 cache manager setup -->*

<bean id="jCacheManager" .../>

#### 8.6.6. Dealing with caches without a backing store

Sometimes when switching environments or doing testing, one might have cache declarations without an actual backing cache configured. As this is an invalid configuration, at runtime an exception will be thrown since the caching infrastructure is unable to find a suitable store. In situations like this, rather then removing the cache declarations (which can prove tedious), one can wire in a simple, dummy cache that performs no caching - that is, forces the cached methods to be executed every time:

<bean id="cacheManager" class="org.springframework.cache.support.CompositeCacheManager">

<property name="cacheManagers">

<list>

<ref bean="jdkCache"/>

<ref bean="gemfireCache"/>

</list>

</property>

<property name="fallbackToNoOpCache" value="true"/>

</bean>

The CompositeCacheManager above chains multiple CacheManagers and additionally, through the fallbackToNoOpCacheflag, adds a no op cache that for all the definitions not handled by the configured cache managers. That is, every cache definition not found in either jdkCache or gemfireCache (configured above) will be handled by the no op cache, which will not store any information causing the target method to be executed every time.

## Plugging-in different back-end caches

Clearly there are plenty of caching products out there that can be used as a backing store. To plug them in, one needs to provide a CacheManager and Cache implementation since unfortunately there is no available standard that we can use instead. This may sound harder than it is since in practice, the classes tend to be simple [adapter](https://en.wikipedia.org/wiki/Adapter_pattern)s that map the caching abstraction framework on top of the storage API as the ehcache classes can show. Most CacheManager classes can use the classes in org.springframework.cache.support package, such as AbstractCacheManager which takes care of the boiler-plate code leaving only the actual mapping to be completed. We hope that in time, the libraries that provide integration with Spring can fill in this small configuration gap.

## How can I set the TTL/TTI/Eviction policy/XXX feature?

Directly through your cache provider. The cache abstraction is…​ well, an abstraction not a cache implementation. The solution you are using might support various data policies and different topologies which other solutions do not (take for example the JDK ConcurrentHashMap) - exposing that in the cache abstraction would be useless simply because there would no backing support. Such functionality should be controlled directly through the backing cache, when configuring it or through its native API.