Caching with Spring Ehcache

**What is a Caching?**

A cache is itself can be imagined as a key-value based store. that stores data so future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation, or the duplicate of data stored elsewhere.

A cache hit occurs when the requested data can be found in a cache, while a cache miss occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store.

**Caching in Spring [ EhCache ]**

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.

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**

Caching Annotations

1. @EnableCaching : Configuration level annotation, enables Caching
2. @Cacheable : Triggers cache population
3. @CachePut : Updates the cache, without interfering with the method execution
4. @CacheEvict : Triggers cache eviction[removing items from cache]
5. @Caching : Regroups multiple cache operations to be applied on a method
6. @CacheConfig : Shares some common cache-related settings at class-level
7. @EnableCaching

**How to Enable Caching?**

@EnableCaching annotation, usually applied on a @Configuration class, triggers a post processor that inspects every Spring bean for the presence of caching annotations [@Cacheable, @CacheEvict, @CachePut..] on public methods. If such an annotation is found, a proxy is automatically created to intercept the method call and handle the caching behavior accordingly.

org.springframework.cache.CacheManager is the common Cache abstraction provided by spring to handle all caching related activities. Several options are available in market: JDK java.util.concurrent.ConcurrentMap based caches, EhCache, Gemfire cache, Caffeine, Guava caches and JSR-107 (JCache) compliant caches.

1. @Cacheable

Used for Cache-population. @Cacheable annotation indicates that the result of invoking a method (or all methods in a class) can be cached. Almost anything [object, array, list, ..] can be cached.

First time a method annotated with @Cacheable gets called, it gets executed and it’s return value is stored in Cache using a key[method parameter for instance, ]. Next time, if the method gets called using same key[same parameter for instance], the result is returned directly from Cache, without executing the method.

Attributes :

cacheNames, values : - Names of the caches in which method invocation results are stored.

cacheManager : - Bean Name of custom CacheManager.

key : - Used to specify the key for cache storage. SpEL also can be used. If you missed to provide the ‘key’ attribute, Spring may generate the key based on method argument itself.

condition : - Conditional Caching. Item will be cached only if the condition mentioned in ‘condition’ met. Note that condition applies to method argument and evaluated before method execution.

unless : - Conditional Caching, applies to return value of method. Item will be cached, unless the condition mentioned in ‘unless’ met. Note that condition applies to return value of method. #result refers to method return value.

keyGenerator : - The bean name of the custom KeyGenerator to use. Mutually exclusive with the key attribute.

1. @CachePut

Used for Cache-update operation. Method annotated with @CachePut are always gets executed and their result gets stored in the cache, eventually overriding any entry with same key in cache. @CachePut, like @Cacheable, supports several attributes, having similar functionality as described above.

1. @CacheEvict

Used for Cache-removal /cache-cleanup operation. @CacheEvict annotation indicates that a method (or all methods on a class) triggers a cache evict operation, removing specific [or all] items from cache.

Attributes apart from others discussed earlier :

allEntries : - Whether all the entries inside the cache(s) are removed.

beforeInvocation : - Whether the eviction should occur before the method is invoked.

1. @Caching

@Caching annotation comes handy when you want to specify multiple annotations of the same type, such as @CacheEvict or @CachePut or @Cacheable on same method. Because, it is not allowed by the language itself to have two annotations of the same time on same element.

Attributes are : *evict, put, cacheable*

Example :

@Caching(evict = { @CacheEvict( "primary" ),

@CacheEvict( cacheNames="secondary", key="#p0") })

@Caching ( put = {@CachePut( value="listcache", keyGenerator = "customkey"),

@CachePut( value="customCache",key = "#root.targetClass")})

@Caching(cacheable= {@Cacheable( value= "customCache", key = "#userID")

@Cacheable( value= "saveUserCache", key = "#userID")})

1. @CacheConfig

@CacheConfig is a class-level annotation which can be used to specify the common caching related settings directly on class level, thus freeing user from duplicating them on each method level.

Attributes are : cacheManager, KeyGenerator

**Custom Key Generator**

After enabling the caching spring used the org.springframework.cache.interceptor.DefaultKeyGenerator to generate keys if there is no specific key is provided. However, in case default key may not satisfy our requirement to manage various caches based on key.

So, we can implement own custom key generator by implementing org.springframework.cache.interceptor.KeyGenerator interface & overriding generate() method to acquire custom keys as per the application requirements.

Example :

**public** **class** CustomKeyGenerator **implements** KeyGenerator {

@Override

**public** Object generate(Object target, Method method, Object... params) {

**/\*** Generate a key for the given method and its parameters.

**\*** key generating logic

**\*** goes here

**\*/**

**return** obj;

}

}

**EhCache Configuration [ XML ]**

CacheManager can be configured in EhCache in XML,

Properties **ehcache.xml**

name : - Name of the cache to be used.

maxEntriesLocalHeap : - Sets the maximum objects to be held in memory (0 = no limit).

maxEntriesLocalDisk : - Builder which sets the maximum number elements on Disk.

eternal : - Sets whether elements are eternal. If eternal, timeouts are ignored and the element is never expired. False by default.

overflowToDisk : - Sets whether elements can overflow to disk when the in-memory cache has reached the set limit.

timeToLiveSeconds : - Sets the time to idle for an element before it expires. Is only used if the element is not eternal.

timeToIdleSeconds : - the time to idle for an element before it expires. Is only used if the element is not eternal. A value of 0 means do not check for idling.

transactionalMode : - Sets the transaction mode [available modes are - OFF, LOCAL, XA, XA\_STRICT]

Example

<ehcache xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:noNamespaceSchemaLocation=*"http://ehcache.org/ehcache.xsd"*>

<defaultCache eternal=*"false"*

maxElementsInMemory=*"1000"*

overflowToDisk=*"false"*

timeToLiveSeconds=*"600"*/>

<cache name=*"customCache"*

maxElementsInMemory=*"1000"*

eternal=*"false"*

overflowToDisk=*"false"*

timeToLiveSeconds=*"600"*/>

</ehcache>

**JCache (JSR-107) annotations**

Spring cache was introduced in Spring 3.1, and brought annotations such as @CachePut to define uses of caches in a Spring application; by default, Spring cache uses a plain Map, but you can configure it to use any popular caching framework, including EhCache, Caffeine, Guava caches.

Since the caching service is an abstraction (not a cache implementation) and requires the use of an actual storage to store the cache data. Since Spring 4.1, Spring cache supports JSR-107, the standard for caching on the JVM.

What that means, is that you can add JSR-107 caching annotations, and then choose your caching library (EhCache 2 or 3 / guava / caffeine / etc.) : you're not tied to any caching vendor, even not tied to Spring cache annotations !

Ehcache is a JVM caching library, famous for being used as the default 2nd level cache for the Hibernate ORM.

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

|  |  |
| --- | --- |
| SPRING | JSR-107 |
| @Cacheable | @CacheResult |
| @CachePut | @CachePut |
| @CacheEvict | @CacheRemove |
| @CacheEvict(allEntries=true) | @CacheRemoveAll |
| @CacheConfig | @CacheDefaults |

**JCache, why and how?**

Most caching solutions are based on map like data structures and JCache API tries to standardize the most common use cases. If you have advanced needs, you probably have to use some implementation specific features, like with JPA, but the standard will definitely make it easier to swap between caching libraries in the future. And it also makes it easier for developers to move from a project to another, which are probably using different caching libraries.

For References :

<https://vaadin.com/blog/-/blogs/jcache-why-and-how->

<https://spring.io/blog/2014/04/14/cache-abstraction-jcache-jsr-107-annotations-support>

<https://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html#cache-jsr-107>

<http://www.ehcache.org/blog/2016/05/18/ehcache3_jsr107_spring.html>