**Saturday, 8 September 2012**

**Spring 3.1 Caching and @Cacheable**

Caches have been around in the software world for long time. They’re one of those really useful things that once you start using them you wonder how on earth you got along without them so, it seems a little strange that the Guys at Spring only got around to adding a caching implementation to Spring core in version 3.1. I’m guessing that previously it wasn’t seen as a priority and besides, before the introduction of Java annotations, one of the difficulties of caching was the coupling of caching code with your business code, which could often become pretty messy.

Spring 3.1 M1 is out with some very [useful features](http://blog.springsource.com/2011/02/11/spring-framework-3-1-m1-released/). One of the coolest feature in the latest release is comprehensive Caching support!

Spring Framework provides support for transparently adding caching into an existing Spring application. Similar to the transaction support, the caching abstraction allows consistent use of various caching solutions with minimal impact on the code.

The cache is applied to Java methods, reducing the number of executions based on the information available. Spring checks if the given method is already executed for given set of parameters. If the method is already executed, Spring uses the cache value and returns it to caller instead of calling the method again. This is a write through cache. 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.

However, the Guys at Spring have now devised a simple to use caching system based around a couple of annotations: @Cacheable and @CacheEvict.  
  
The idea of the @**Cacheable** annotation is that you use it to mark the method return values that will be stored in the cache.   
The @**Cacheable** annotation can be applied either at method or type level. When applied at method level, then the annotated method’s return value is cached. When applied at type level, then the return value of every method is cached.  
  
The code below demonstrates how to apply @Cacheable at type level:

@Cacheable(value = "employee")  
public class EmployeeDAO {  
  
  public Person findEmployee(String firstName, String surname, int age) {  
  
    return new Person(firstName, surname, age);  
  }  
  
  public Person findAnotherEmployee(String firstName, String surname, int age) {  
  
    return new Person(firstName, surname, age);  
  }  
}

The Cacheable annotation takes three arguments: value, which is mandatory, together with key and condition. The first of these, value, is used to specify the name of the cache (or caches) in which the a method’s return value is stored.

  @Cacheable(value = "employee")  
  public Person findEmployee(String firstName, String surname, int age) {  
    return new Person(firstName, surname, age);  
  }

The code above ensures that the new Person object is stored in the “employee” cache.  
  
Any data stored in a cache requires a key for its speedy retrieval. Spring, by default, creates caching keys using the annotated method’s signature as demonstrated by the code above. You can override this using @Cacheable’s second parameter: key. To define a custom key you use a SpEL expression.

  @Cacheable(value = "employee", key = "#surname")  
  public Person findEmployeeBySurname(String firstName, String surname, int age) {  
  
    return new Person(firstName, surname, age);  
  }

In the ***findEmployeeBySurname***(...) code, the ‘#surname’ string is a SpEL expression that means ‘go and create a key using the surname argument of the findEmployeeBySurname(...) method’.  
  
The final @Cacheable argument is the optional condition argument. Again, this references a SpEL expression, but this time it’s specifies a condition that’s used to determine whether or not your method’s return value is added to the cache.

  @Cacheable(value = "employee", condition = "#age < 25")  
  public Person findEmployeeByAge(String firstName, String surname, int age) {  
  
    return new Person(firstName, surname, age);  
  }

In the code above, I’ve applied the ludicrous business rule of only caching Person objects if the employee is less than 25 years old.  
  
Having quickly demonstrated how to apply some caching, the next thing to do is to take a look at what it all means.

  @Test  
  public void testCache() {  
  
    Person employee1 = instance.findEmployee("John", "Smith", 22);  
    Person employee2 = instance.findEmployee("John", "Smith", 22);  
  
    assertEquals(employee1, employee2);  
  }

The above test demonstrates caching at its simplest. The first call to findEmployee(...), the result isn’t yet cached so my code will be called and Spring will store its return value in the cache. In the second call to findEmployee(...) my code isn’t called and Spring returns the cached value; hence the local variable employee1 refers to the same object reference as employee2, which means that the following is true:

 assertEquals(employee1, employee2);

But, things aren’t always so clear cut. Remember that in findEmployeeBySurname I’ve modified the caching key so that the surname argument is used to create the key and the thing to watch out for when creating your own keying algorithm is to ensure that any key refers to a unique object.

  @Test  
  public void testCacheOnSurnameAsKey() {  
  
    Person employee1 = instance.findEmployeeBySurname("John", "Smith", 22);  
    Person employee2 = instance.findEmployeeBySurname("Jack", "Smith", 55);  
  
    assertEquals(employee1, employee2);  
  }

The code above finds two Person instances which are clearly refer to different employees; however, because I’m caching on surname only, Spring will return a reference to the object that’s created during my first call to findEmployeeBySurname(...). This isn’t a problem with Spring, but with my poor cache key definition.  
  
Similar care has to be taken when referring to objects created by methods that have a condition applied to the @Cachable annotation. In my sample code I’ve applied the arbitrary condition of only caching Person instances where the employee is under 25 years old.

  @Test  
  public void testCacheWithAgeAsCondition() {  
  
    Person employee1 = instance.findEmployeeByAge("John", "Smith", 22);  
    Person employee2 = instance.findEmployeeByAge("John", "Smith", 22);  
  
    assertEquals(employee1, employee2);  
  }

In the above code, the references to employee1 and employee2 are equal because in the second call to findEmployeeByAge(...) Spring returns its cached instance.

  @Test  
  public void testCacheWithAgeAsCondition2() {  
  
    Person employee1 = instance.findEmployeeByAge("John", "Smith", 30);  
    Person employee2 = instance.findEmployeeByAge("John", "Smith", 30);  
  
    assertFalse(employee1 == employee2);  
  }

Similarly, in the unit test code above, the references to employee1 and employee2 refer to different objects as, in this case, John Smith is over 25.  
  
That just about covers @Cacheable, but what about @CacheEvict and clearing items form the cache? Also, there’s the question adding caching to your Spring config and choosing a suitable caching implementation. However, more on that later....

Like @Cacheable, @CacheEvict has value, key and condition attributes. These work in exactly the same way as those supported by @Cacheable

@CacheEvict supports two additional attributes: allEntries and beforeInvocation. If I were a gambling man I'd put money on the most popular of these being allEntries. allEntries is used to completely clear the contents of a cache defined by @CacheEvict's mandatory value argument. The method below demonstrates how to apply allEntries:

@CacheEvict(value = "employee", allEntries = true)

public void resetAllEntries() {

  // Intentionally blank

}

The second attribute is beforeInvocation. This determines whether or not a data item(s) is cleared from the cache before or after your method is invoked.  
  
The code below is pretty nonsensical; however, it does demonstrate that you can apply both @CacheEvict and @Cacheable simultaneously to a method.



@CacheEvict(value = "employee", beforeInvocation = true)

@Cacheable(value = "employee")

public Person evictAndFindEmployee(String firstName, String surname, int age)

  return new Person(firstName, surname, age);

}

In the code above, @CacheEvict deletes any entries in the cache with a matching key before @Cacheable searches the cache. As @Cacheable won’t find any entries it’ll call my code storing the result in the cache. The subsequent call to my method will invoke @CacheEvict which will delete any appropriate entries with the result that in the JUnit test below the variable employee1 will never reference the same object as employee2:

@Test

public void testBeforeInvocation() {

  Person employee1 = instance.evictAndFindEmployee("John", "Smith", 22);

  Person employee2 = instance.evictAndFindEmployee("John", "Smith", 22);

  assertNotSame(employee1, employee2);

}

As I said above, evictAndFindEmployee(...) seems somewhat nonsensical as I’m applying both @Cacheable and @CacheEvict to the same method. But, it’s more that that, it makes the code unclear and breaks the Single Responsibility Principle; hence, I’d recommend creating separate cacheable and cache-evict methods. For example, if you have a cacheing method such as:



@Cacheable(value = "employee", key = "#surname")

public Person findEmployeeBySurname(String firstName, String surname, int age) {

  return new Person(firstName, surname, age);

}

then, assuming you need finer cache control than a simple ‘clear-all’, you can easily define its counterpart:

@CacheEvict(value = "employee", key = "#surname")

public void resetOnSurname(String surname) {

  // Intentionally blank

}

This is a simple blank marker method that uses the same SpEL expression that’s been applied to @Cacheable to evict all Person instances from the cache where the key matches the ‘surname’ argument.

@Test

public void testCacheResetOnSurname() {

  Person employee1 = instance.findEmployeeBySurname("John", "Smith", 22);

  instance.resetOnSurname("Smith");

  Person employee2 = instance.findEmployeeBySurname("John", "Smith", 22);

  assertNotSame(employee1, employee2);

}

In the above code the first call to findEmployeeBySurname(...) creates a Person object, which Spring stores in the “employee” cache with a key defined as: “Smith”. The call to resetOnSurname(...) clears all entries from the “employee” cache with a surname of “Smith” and finally the second call to findEmployeeBySurname(...) creates a new Person object, which Spring again stores in the “employee” cache with the key of “Smith”. Hence, the variables employee1, and employee2 do not reference the same object.

**Currently supported libraries**

There are probably hundreds of cache libraries available which can be used in your JEE project. For now the Spring framework supports following implementations:

1. JDK ConcurrentMap based Cache
2. Ehcache based Cache

**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.

|  |
| --- |
| <!-- generic cache manager -->  <bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">    <property name="caches">      <set>        <bean class="org.springframework.cache.concurrent.ConcurrentCacheFactoryBean" p:name="default"/>        <bean class="org.springframework.cache.concurrent.ConcurrentCacheFactoryBean" p:name="persons"/>      </set>    </property>  </bean> |

In above code snippet, we use SimpleCacheManager class to create a CacheManager. Note that we have created two caches in our application, one is default and second is persons.

**Ehcache based Cache**

The Ehcache 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="ehcache"/>    <!-- Ehcache library setup -->  <bean id="ehcache" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean" p:config-location="ehcache.xml"/> |

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

**Using Default key**

The cache is nothing but a key-value store which stores the data based on certain key. In Spring framework based caching, the method arguments of cached method acts as the source of Key generation. Every key is essentially the Hash-code of these arguments. This approach works well for objects with natural keys as long as the hashCode() reflects that. If that is not the case then for distributed or persistent environments, the strategy needs to be changed as the objects hashCode is not preserved. In fact, depending on the JVM implementation or running conditions, the same hashCode can be reused for different objects, in the same VM instance.

To provide a different default key generator, one needs to implement the org.springframework.cache.KeyGenerator interface. Once configured, the generator will be used for each declaration that does not specify its own key generation strategy.

By default, all the method arguments are used in Key generation logic. In practice not all methods have only one argument or, worse yet, the parameters are not suitable as cache keys – take for example a variation of the method above:

|  |
| --- |
| @Cacheable(value="persons", key="personId")  public Person profile(Long personId, Long groundId) { ... } |

Reference

<http://java.dzone.com/articles/spring-31-caching-and-0>

<http://viralpatel.net/blogs/cache-support-spring-3-1-m1/>

<http://blog.springsource.com/2011/02/11/spring-framework-3-1-m1-released/>

<http://stackoverflow.com/questions/13263392/spring-mvc-3-caching-example>

<https://github.com/kenyattaclark/spring-cache-example>

<http://techannotation.wordpress.com/2012/05/24/5-minutes-with-spring-cache/>

<http://www.captaindebug.com/2012/09/spring-31-caching-and-cacheable.html#.UUtqPjdvAqc>