Cache Abstraction: JCache (JSR-107) Annotations Support

Spring's caching abstraction is available as from Spring 3.1 and it was about time to show it some more love. In this post, I want to walk you through the major improvement in that area which is the JCache (JSR-107) annotations support.

As you may have heard, JSR-107 went final after all, 13 years after the initial proposal. For those who are familiar with Spring's caching annotations, the following table describes the mapping between the Spring annotations and the JSR-107 counterpart:

Spring	JSR-107
`@Cacheable`	`@CacheResult`
`@CachePut`	`@CachePut`
`@CacheEvict`	`@CacheRemove`
`@CacheEvict(allEntries=true)`	`@CacheRemoveAll`

JCache annotations

Let's first look at each annotation and describe how they

can be used. This will be a chance to better understand what they support with regards to what you've been used to with the Spring annotations and more importantly the **new** features that these annotations bring.

@CacheResult

`@CacheResult` is fairly similar to `@Cacheable`, the following rewrites the original example using the `@CacheResult` annotation:

```
@CacheResult(cacheName = "books")
public Book findBook(ISBN isbn) {...}
```

COPY

Keys generation can be customized using the `CacheKeyGenerator` interface. If no specific implementation is specified, the default implementation, *per* spec, takes all parameters unless one or more parameters are annotated with the `@CacheKey` annotation, in which case only those are used. Assuming that the method above needs now an extra attribute that should not be part of the key, this is how we would write it with JCache:

```
@CacheResult(cacheName = "book")
public Book findBook(@CacheKey ISBN isbn,
boolean checkWarehouse) { ... }
```

COPY

`@CacheResult` brings the concept of *exception* cache: whenever a method execution failed, it is possible to *cache* the exception that was raised to prevent 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(@CacheKey ISBN isbn) { ...
}
```

COPY

Of course, blindly throwing a cached exception might be very confusing as the call stack may not match with the current invocation context. We do our best to make sure the stacktrace matches by copying the exception with a consistent call stack.

JCache has this cool notion of `CacheResolver` that permits to resolve the cache to use at runtime. Because JCache supports regular caches and exception caches, the `CacheResolver` instances to use are determined by a `CacheResolverFactory`. The obvious default is to resolve the cache to use based on the `cacheName` and `exceptionCacheName` attributes, respectively. However, it is also possible to customize the factory to use per operation.

```
@CacheResult(cacheName = "books",
cacheResolverFactory = MyFactory.class)
public Book findBook(@CacheKey ISBN isbn) { ...
}
```

COPY

Finally, `@CacheResult` has a `skipGet` attribute that can be enabled to *always* invoke the method regardless of the status of the cache. This is actually quite similar to our own use of `@CachePut`.

@CachePut

While the annotations have the same name, the semantic in JCache is fairly different. A simple update for our book would be written like this

```
@CachePut(value = "books", key = "#p0")
public Book update(ISBN isbn, Book updatedBook)
{ ... }
```

COPY

While JCache would require you to write it like this

```
@CachePut(cacheName = "books")
public void update(ISBN isbn, @CacheValue Book
updatedBook) { ... }
```

COPY

Note that even though `updatedBook` should not be part of the key, we didn't have to add a `@CacheKey` to the first argument. This is because the parameter annotated with `@CacheValue` is automatically excluded from the key generation.

As for `@CacheResult`, `@CachePut` allows to manage any exception that is thrown while executing the method, preventing the put operation to happen if the thrown exception matches the filter specified on the annotation.

Finally, it is possible to control if the cache is updated before or after the invocation of the annotated method. Of course, if it is updated before, no exception handling takes place.

@CacheRemove and @CacheRemoveAll

These are really similar to `@CacheEvict` and

`@CacheEvict(allEntries = true)` respectively. `@CacheRemove` has a special exception handling to prevent the eviction if the annotated method throws an exception that matches the filter specified on the annotation.

Other features

CacheDefaults

`@CacheDefaults` is a class-level annotation that allows you to *share* common settings on any caching operation defined on the class. These are:

- The name of the cache
- The custom `CacheResolverFactory`
- The custom `CacheKeyGenerator`

In the sample below, any cache-related operation would use the `books` cache:

```
@CacheDefaults(cacheName = "books")
public class BookRepositoryImpl implements
BookRepository {

    @CacheResult
    public Book findBook(@CacheKey ISBN isbn) {
... }
}
```

COPY

Enabling JSR-107 support

The implementation of the JCache support uses our own `Cache` and `CacheManager` abstraction which means that you can use your existing `CacheManager` infrastructure, and yet use standard annotations!

To enable the support of Spring caching annotations, you

are used to either `@EnableCaching` or the `<cache:annotation-driven/>` xml element, for instance something like:

```
@Configuration
@EnableCaching
public class AppConfig {
    @Bean
    public CacheManager cacheManager() { ...}

    ...
}
```

COPY

So, what does it take to bring the support of standard annotations into the mix? Well, not much. Just add the JCache API and the `spring-context-support` module in your classpath if you haven't already and you'll be set.

The existing infrastructure actually looks for the presence of the JCache API and when found alongside Spring's JCache support, it will also configure the necessary infrastructure to support the standard annotations.

Wrapping up

Long story short, if you are already using Spring's caching abstraction and you'd like to try the standard annotations, adding two more dependencies to your project is all that would be needed to get started.

Want to give it a try? Grab a nightly SNAPSHOT build of Spring 4.1 and add the `javax.cache:cache-api:1.0.0` and `org.springframework:spring-context-support:4.1.0.BUILD-SNAPSHOT` dependencies to your project. The documentation has also been updated in case you need more details.

In a next post, I'll cover how supporting JSR-107

annotations affected our own support as well as some other cache-related improvements.

Comments Community Privacy Policy 1 Login

Recommend 1 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Ken Krueger • 6 years ago

Does the Spring team have a recommendation on whether new development efforts should use JSR-107 or Spring annotations?

2 • Reply • Share ›

Stéphane Nicoll ↗ Ken Krueger • 6 years ago

I'd say the choice is up to you basically. You have to understand what the standard annotations can and cannot do. For instance, only one annotation is allowed per method and only one cache can be specified. If your use cases fit with those restrictions and you are preferring the use of standard annotations, JCache sounds an obvious choice.

If you absolutely rely on several caches and/or operations per method, you don't really have a choice either as JCache does not support that.

• Reply • Share ›

Francisco Lozano ↗ Stéphane Nicoll • 6 years ago

What about the inverse? JCache seems to support exception caches, whereas spring native annotations don't (as of 4.0)

• Reply • Share ›

Stéphane Nicoll ↗ Francisco

Lozano • 6 years ago

That's a good point and I realize now I should have been more complete in my answer. There are indeed other features that only JCache supports now and exception caching is one of them.

In a next blog post, I'll explain what we have added to our own caching support but some JCache features are strongly linked to the fact that only one cache operation is allowed per method invocation. As this is not the case for our own support, these features will be hard to integrate.

Reply • Share ›

NK • 2 years ago

I am using @CacheRemoveAll provide a cache refresh feature at run time. But calling method with this annotation cache is not cleared. But whe try to clear programatically it gets cleared. beanJCacheCacheManager.getCache("xxx").clear() How to make @CacheRemoveAll work

Reply • Share ›

Stéphane Nicoll NK • 2 years ago

This blog post is not the best place to get support. Please ask on StackOverflow with the `spring-cache` and additional details about your setup and we can take it from there. Thank you

1 Reply • Share ›

Marcel Stör • 3 years ago • edited

As for CacheDefaults you're saying:

> In the sample below, any cache-related operation would use the books cache:

That would work well if the class had a let's say `findByISBN(ISBN):Book` and a

`findByAuthor(String):Set<book>` method, right? The cache itself is basically a generic key-value store and you can reuse the same cache even if the methods use different cache keys (ISBN vs. String) and different cache values (Book vs. Set<book>). Correct?

Reply • Share ›

Stéphane Nicoll Marcel Stör • 3 years ago

That works well if the class works on a single cache. That way you don't have to repeat those settings over and over again. I personally don't see a cache (in the abstraction or in a library specific arrangement) as a generic key-value store. I like to see the abstraction as a way to avoid you handing the cache API directly.

In other words, I wouldn't mix several things in the same cache and I'd be quite careful as how the key is generated; typically I would expect `cache.get(id)` to work with a consistent ID type. So, no, I wouldn't mix `String` and `ISBN` in the same cache.

Of course, if you feel differently, the abstraction won't prevent you to do that.

Having said all of that, I am not sure I understand your question. That specific feature is meant to share settings that apply to a given type. If that applies for your use cases, then please give that a try.

Reply • Share ›

vinay • 6 years ago

See JSR107 example using EhCache as the implementation @ https://github.com/vinaynai...

Reply • Share ›

Petar Tahchiev • 6 years ago

Hi guys,

I would really appreciate it if you make a blog post on how to setup caching in Spring. In my project I have a

sample method with 1 logging statement and caching enabled, and also an integration test-case that calls the method always with the same CacheKey, but I see multiple logs being printed. I use the EHCache-JCache integration (1.0-SNAPSHOT) and I think I have set it up right, but it still goes through my method and calls the logging. Is there a sample application somewhere on github that I can have a look at?

 • Reply • Share ›

Stéphane Nicoll Petar Tahchiev
• 6 years ago • edited

Bear in mind that you do *not* need a JSR-107 cache implementation to use the standard JSR-107 annotations with Spring. You can just use