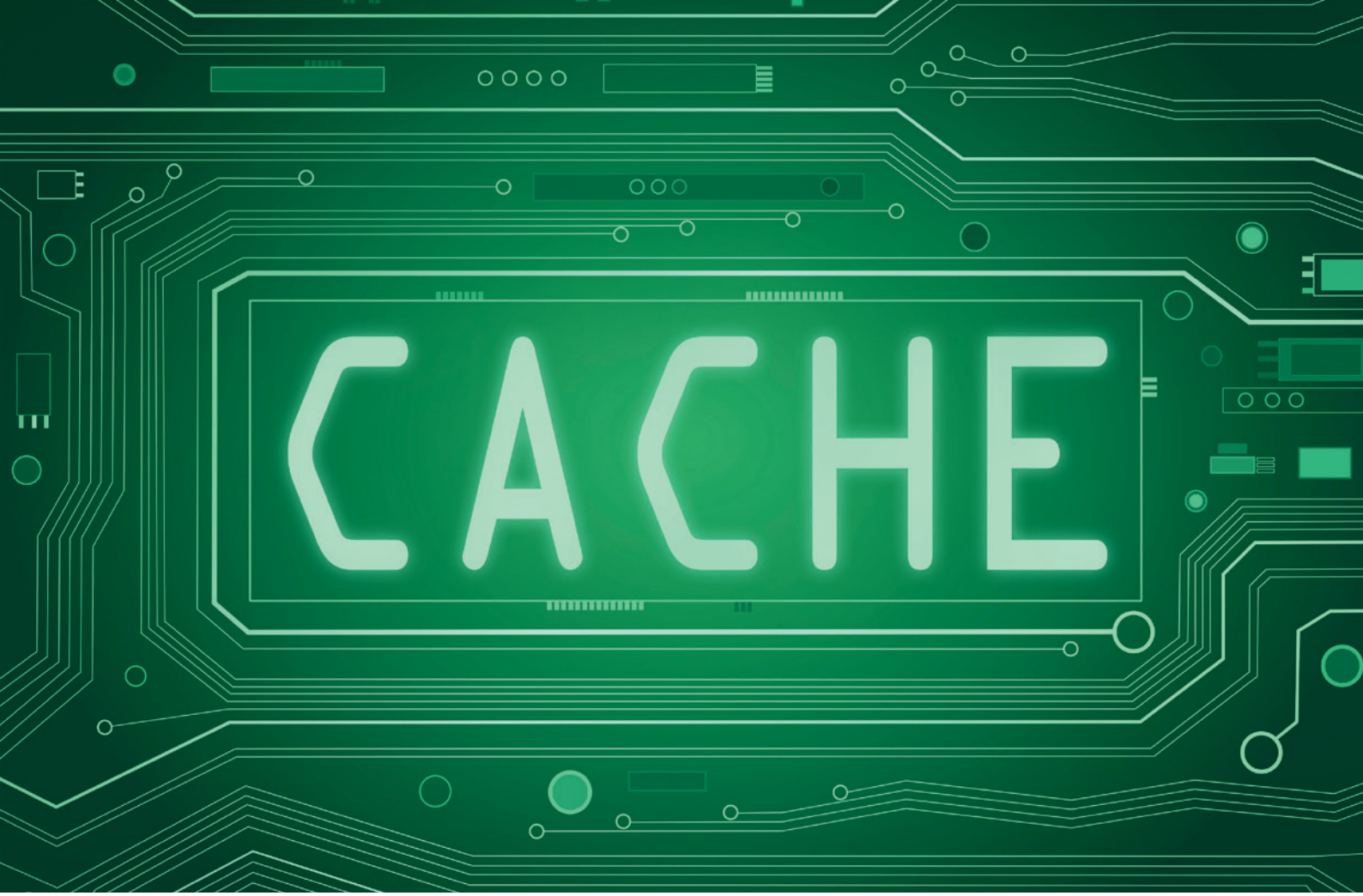

Caching mit Spring über JCache hinaus

Andreas Wirth, Bitech AG

Der Artikel zeigt anhand eines durchgängigen Beispiels die Spring-Cache-Abstraktion analog zu JCache (JSR-107). Zudem wird auf die über JCache hinausgehenden Möglichkeiten des Spring-Cachings eingegangen und demonstriert, wie der Spring-Context innerhalb eines Aspekts verwendet werden kann.

Im Beispiel kommen vor allem die Spring-Framework-Komponenten „spring-context-support" und „spring-test" zum Einsatz. Anschließend sind die Ziele des JSR-107 klar und man kann JCache in einer JEE-Umgebung nutzen. Hinzu kommen einige über JCache hinausgehende Caching-Möglichkeiten, die in der Spring-Cache-Abstraktion geboten werden und in einer Spring-Anwendung verwendet werden können. Die Leser wissen den Spring-Context zielgerichtet innerhalb ihrer eigenen Aspekte zu nutzen, eine über Caching hinausgehende interessante Option.

Caching ist ein altes Thema, aber gerade heutzutage wieder aktuell. In (Micro-) Service-Landschaften ist eine Vielzahl von Remote-Aufrufen notwendig und bei jedem Aufruf stellt sich die Frage, ob hier Caching helfen kann, zu lange Wartezeiten zu vermeiden. Umso erfreulicher, dass mit JCache (JSR-107) das Caching standardisiert ist und Teil von JEE 8 (JSR-366) sein wird. Dabei lohnt sich der standardisierte Ansatz auch für leichtgewichtige Implementierungen.

Definition JCache

JCache definiert Caching so: „In the context of application design it is often used to describe the technique whereby application developers utilize a separate in-memory or low-latency data-structure, a Cache, to temporarily store, or cache, a copy of or reference to information that an application may reuse at some later point in time, this alleviating the cost to re-access or re-create it" *[1]*.

Ebenso wird in der Spezifikation bereits beschrieben, wo JCache eingesetzt werden kann: „Fundamentally any information that is ex-

pensive or time consuming to produce or access can be stored in a cache" *[1]*. Damit spielt Caching sowohl auf Service-Consumer- als auch auf Service-Provider-Seite eine Rolle.

Das 5-Kern-Konzept

JCache basiert auf fünf Kernkomponenten: Über einen CachingProvider wird in der Regel ein CacheManager angefordert, der n Caches verwalten kann. Die jeweiligen Cache-Entries bestehen immer aus einem Key-Value-Paar *(siehe Abbildung 1)*.

Sofern nur ein JCache-konformer CachingProvider im Classpath vorhanden ist, kann dieser über den DefaultClassLoader geladen werden. Der Zugriff auf die anderen vier Kernkomponenten erfolgt entsprechend der Hierarchie *(siehe Listing 1)*.

Die Grundoperationen bezüglich der Cache-Entries können programmatisch oder per Annotation erfolgen *(siehe Tabelle 1)*. Werden Annotations verwendet, muss dafür ein geeigneter Context zur Verfügung stehen. Geeignet sind alle JEE-8-Container (wie GlassFish 5), per Extension können auch CDI-Container-JCache-Annotations berücksichtigt werden *[3]* oder einfach ein Spring-Context – wie später gezeigt. Darüber hinaus definiert JCache weitere Covenient-Methoden *[4]* wie zum Beispiel:

- replace
- containsKey
- unwrap

Weitere Konzepte

Neben den fünf Kernkomponenten definiert JCache einige weitere Standards. Besonders hilfreich findet der Autor folgende:

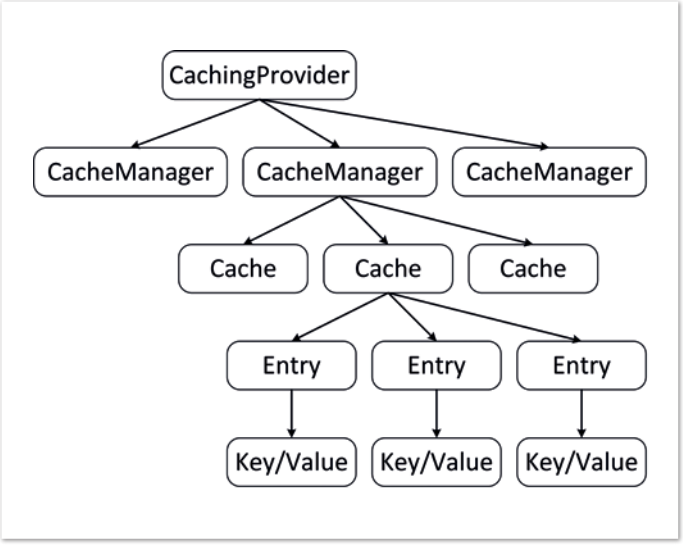

Abbildung 1: JCache Core Concepts [2]

- **Configuration**
  Die wichtigsten Konfigurationen lassen sich standardisiert durchführen. Speziellere implementierungsspezifische Features lassen sich beispielsweise durch Ableitung von „MutableConfiguration" konfigurieren. Wird eine spezielle Konfiguration nur einmalig benötigt, ermöglicht „unwrap" (siehe oben) einen einfachen Zugang zur implementierungsspezifischen Konfiguration.
- **CacheEntryListener**
  Für die „CacheEntry"-Events „create", „expire", „remove" und „update" können Listener definiert werden *(siehe Listing 2)*. Gerade für Logging-Belange ist dies hilfreich.

```
CachingProvider cachingProvider = Caching.getCachingProvider();
CacheManager cacheManager = cachingProvider.getCacheManager();
MutableConfiguration<Integer, List> conf = new MutableConfiguration<>();
Duration duration = new Duration(TimeUnit.SECONDS, 3L);
conf.setExpiryPolicyFactory(CreatedExpiryPolicy.factoryOf(duration));
Cache<Integer, List> cache = cacheManager.createCache("myCache", conf);
cache.put(key1, val1);
List val2 = cache.get(key2);
```

Listing 1

```
MutableConfiguration<Integer, List> conf = new MutableConfiguration<>();
CacheEntryListenerConfiguration celc = new MutableCacheEntryListenerConfiguration(
  FactoryBuilder.factoryOf(MyCacheEntryListener.class), null, false, false);
conf.addCacheEntryListenerConfiguration(celc);
```

Listing 2

Programmatisch	Annotation	Beschreibung
cache.put	@CachePut	Fügt einen Entry dem Cache hinzu
cache.get	@CacheResult	Falls Key vorhanden, liest einen Entry aus dem Cache; falls Key nicht vorhanden, dann führt „@CacheResult" die Methode aus und fügt das Ergebnis dem Cache hinzu
cache.remove	@CacheRemove	Entfernt einen Entry aus dem Cache
cache.removeAll	@CacheRemoveAll	Entfernt alle Entries aus dem Cache

Tabelle 1: Die Grundoperationen bezüglich der Cache-Entries

JCache	Spring
@CachePut	@CachePut
@CacheResult	@Cacheable
@CacheRemove	@CacheEvict
@CacheRemoveAll	@CacheEvict(allEntries=true)
@CacheDefaults	@CacheConfig

Tabelle 2

- **EntryProcessor**
  EnryProcessor ermöglichen durch Locks, Operationen atomar durchzuführen.

JCache-Implementierungen

Es gibt eine weite Auswahl von JCache-Implementierungen, darunter EhCache3, Infinispan, Hazelcast, Oracle Coherence und Apache Ignite. Alle Listings und einige Features werden in einem durchgängigen Beispiel demonstriert *[5]*. Darin kommen beispielsweise EhCache3 und Infinispan zum Einsatz.

Bevor wir zur Spring-Cache-Abstraktion kommen, ein Blick auf den zeitlichen Ablauf: Der Prozess zu JCache (JSR-107) wurde im Jahr 2001 begonnen und erst im Jahr 2014 abgeschlossen. *Abbildung 2* stellt die Ereignisse von JCache, Java Platform Enterprise Edition (JEE) und Spring-Framework gegenüber.

Spring-Cache-Abstraktion

JCache ist im Jahr 2014 verabschiedet worden und wird im Sommer 2017 in JEE 8 aufgenommen. Mit der Version 3.1 (Ende 2011) wurde im Spring-Framework ein ähnliches Konzept realisiert. Seit der Spring-Version 4.1 (Ende 2015) wird auch JCache in Spring unterstützt. Dabei schlägt „JCacheCacheManager" die Brücke von JCache zur Spring-Cache-Abstraktion *(siehe Listings 3 und 4)*. Mit „<cache:annotation-driven/>" oder „@EnableCaching" lässt sich das Caching per Annotations aktivieren. Analog zu den JCache-Annotations können auch die Spring-Cache-Annotations verwendet werden *(siehe Tabelle 2)*.

Der Mehrwert von Spring

Das Spring-Framework bietet dabei einige über JCache hinausgehende Möglichkeiten *[6]:*

- **synchronized caching „@Cacheable(sync=true)"**
  Auch in Multithreading-Umgebungen ist damit sichergestellt, dass die Methode nur einmalig pro Key ausgeführt wird
- **conditional caching „@Cacheable(condition=SpEL, unless=SpEL)"**
  Je nach Input-Parameter („condition") oder Output-Parameter („unless") kann entschieden werden, ob gecacht werden soll
- **key generator „@Cacheable(keyGenerator=REF)"**
  Keys können programmatisch definiert sein
- Adapter für Nicht-JCache-konforme Implementierungen wie EhCache2, Caffeine etc.

Das ist zu berücksichtigen

Wie in Spring üblich, wird Caching per Proxy realisiert. Dabei ist der Standard-Modus JDK-basiert („<cache:annotation-driven mode="proxy"/"> beziehungsweise „EnableCaching(mode=AdviceMode.PROXY)"). Dementsprechend lassen sich nur „public"-Methoden annotieren. Sofern mehrere Advices an demselben

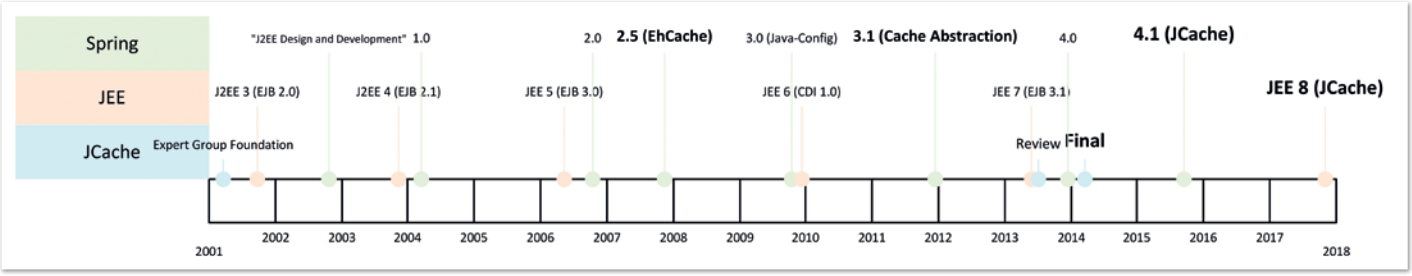

Abbildung 2: JCache-Timeline

```
<bean id="cacheManager" class="org.springframework.cache.jcache.JCacheCacheManager">
  <constructor-arg ref="jCacheManager"/>
</bean>
<bean id="jCacheManager" factory-bean="jCacheFactoryBean" factory-method="buildJCacheCacheManager"/>
```

Listing 3

```
public CacheManager buildJCacheCacheManager() {
  CachingProvider cachingProvider = Caching.getCachingProvider();
  CacheManager ret = cachingProvider.getCacheManager();
  MutableConfiguration<String, List<Integer>> configuration = new MutableConfiguration<>();
  configuration.setExpiryPolicyFactory(AccessedExpiryPolicy.factoryOf(new Duration(TimeUnit.HOURS, 1L)));
  ret.createCache(CACHE_NAME, configuration);
  return ret;
}
```

Listing 4

```
@Autowired
@Qualifier("cacheManager")
private CacheManager cacheManager;

private Cache myCache;

@PostConstruct
public void init() {
  this.myCache = this.cacheManager.getCache("myCache");
}

@Pointcut("execution(* *.*(..))")
public void allExecutions() {
  // NOP
}

  @Pointcut("@annotation(de.bitech.javaday.spring.cache.aspect.MyCache)")
public void myCache() {
  // NOP
}

@Pointcut("allExecutions() && myCache()")
public void execMyCache() {
  // NOP
}

@Around("execMyCache()")
public Object aroundExecMyCache(ProceedingJoinPoint pjp) throws Throwable {
  Object ret = null;
  Object[] params = pjp.getArgs();

  // condition
  int number = (int) params[0];
  if (number <= 1000000) {
     ret = pjp.proceed();
  } else {
     ValueWrapper valueWrapper = this.myCache.get(number);
     if (valueWrapper == null) {
        ret = pjp.proceed();
        this.myCache.put(number, ret);
     } else {
        ret = valueWrapper.get();
     }
  }
  return ret;
}
```

Listing 5

```
@MyCache
private List<Integer> fractionize(int pNumber) {
  [...]
```

Listing 6

```
<bean class="de.bitech.javaday.spring.cache.aspect.CacheAspect" factory-method="aspectOf"/>
```

Listing 7

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/de/bitech/javaday/spring/context-cache.xml")
@DirtiesContext
public class MyTest {
```

Listing 8

Joinpoint definiert sind, muss gegebenenfalls die Ausführ-Reihenfolge definiert sein.

Falls die eingebauten Möglichkeiten („@EnableCaching(order=Ordered.LOWEST_PRECEDENCE)“) nicht ausreichen, kann der Proxy-Modus auf AspectJ geändert werden („@EnableCaching(mode=AdviceMode.ASPECTJ)“). Nun können auch „private“-Methoden annotiert sein, sodass die Aufruf-Reihenfolge in Form der Reihenfolge der Methoden festgelegt ist. Als positiver Nebeneffekt wird der Quellcode damit auch leichter lesbar. Allerdings muss „aop“-typische Instrumentation aktiviert, also „spring-instrument“ als Dependency dem Classpath hinzugefügt und per „<context:load-time-weaver aspectj-weaving="on"/>“ AspectJ-Weaver aktiviert sein.

Eigener Caching-Aspekt

Für Annotations an privaten Methoden benötigt man Bytecode-Manipulation, etwa durch AspectJ. Anstelle von Load-Time-Weaving hätte der Autor lieber Compile-Time-Weaving, damit entfällt Instrumentation. Also kann man einen eigenen Aspekt schreiben, der innerhalb seines Advice das Caching programmatisch ausführt und die Joinpoints per Annotation deklariert *(siehe Listings 5 und 6)*.
Der Aspekt lebt in der „aspectj-runtime“, aber wie kann er auf den Spring-Context zugreifen? Dazu deklariert man mit der „factory-method aspectOf“ den Aspekt als „factory-bean“ und Dependency Injection à la Spring funktioniert im Aspekt ebenfalls *(siehe Listing 7)*.

Ein letztes Problem muss noch gelöst werden: Tests, die den Spring-Context benötigen, erzeugen per Default nur einmalig den Spring-Context. AspectJ erzeugt aber die Aspekte für jeden Durchlauf neu. Mit „@DirtiesContext“ kann man auch Spring dazu veranlassen, in jedem Durchlauf einen frischen Context zu erzeugen *(siehe Listing 8)*.

Fazit

JCache bietet mit seinen fünf Kern-Konzepten ein einheitliches und einfaches Caching-API sowohl programmatisch als auch per Annotation an. Ebenso vereinheitlicht ist die Basiskonfiguration der JCache-Manager. Speziellere Konfigurationen können mit den herstellerspezifischen APIs umgesetzt werden und trotzdem kann der Spring „JCacheCacheManager“ verwendet werden.

Die Caching-Strategie ist nicht Bestandteil von JCache und in jedem Fall zu berücksichtigen. Abhängig von Anzahl und Größe der zu cachenden Objekte ist zu entscheiden, ob der Cache „embedded“ oder „standalone“ realisiert wird – je nach Applikation, also je nachdem, ob der Cache verteilt oder repliziert ausgeprägt sein sollte.

Die Spring-Cache-Abstraktion kann zu 100 Prozent JCache-konform eingesetzt werden. Über die Spring-spezifischen Annotations kommen einige kleine, aber feine zusätzliche Funktionen hinzu. Insbesondere „synchronized“ und „conditional“ Caching bieten in der Praxis einen Mehrwert. Auch die feingranularere Steuerung über AOP-Proxies kann hilfreich sein.

Die meisten Möglichkeiten bietet ein eigener Aspekt. Dieser Artikel zeigt, wie man den Spring-Context einfach innerhalb eines Aspekts verwenden kann, sodass der Mehraufwand dieser Lösung überschaubar bleibt. Gerade auch in Szenarien mit „distributed“ oder „standalone“ Caches bietet ein eigener Aspekt ein einheitliches API innerhalb der Applikation und eine zentrale Stelle für die Caching-Strategie.

Weitere Informationen

[1] *Greg Luck, Brian Oliver: JSR107FinalSpecification 1.2: https://jcp.org/en/jsr/detail?id=107*
[2] *Abhishek Gupta: https://dzone.com/refcardz/java-caching*
[3] *Jonathan Gallimore: http://www.tomitribe.com/blog/2015/06/using-jcache-with-cdi*
[4] *http://static.javadoc.io/javax.cache/cache-api/1.0.0/index.html*
[5] *https://github.com/awi72/spring-jcache*
[6] *Spring Framework Reference Documentation (4.3.6.Release): http://docs.spring.io/spring/docs/current/spring-framework-reference*
[7] *http://hazelcast.com/content/Using-JCache-with-Hazelcast-Slides.pdf*
[8] *http://www.ehcache.org/documentation/3.0/107.html*
[9] *http://www.ehcache.org/blog/2016/05/18/ehcache3_jsr107_spring.html*
[10] *https://www.slideshare.net/SpringCentral/spring-one2gx-caching-with-spring?qid=8fd5ecbc-f5e7-4c84-b984-ce4aa5a9fd4e&v*

Andreas Wirth
andreas.wirth@koeln.bitech.de

Andreas Wirth arbeitet als IT-Berater bei der Bitech AG in Köln. Er unterstützt seine Kunden als Entwickler, Architekt oder Coach in der Java-Enterprise-Entwicklung. Darüber hinaus setzt er sich in seinen Projekten für „clean code“ ein.