JCache in Microprofile
Discussion Paper

Authors:
Greg Luck
Ondrej Mihályi

Table of Contents

Purpose

The purpose of this discussion paper is to be living document capturing the thoughts of those interested in seeing JCache be added to Microprofile.

Microprofile is a new Enterprise Java standard for Microservices.

Motivation

Caching is used to speed up application performance, to scale up in a single instance and to scale out where an application has multiple instances.

It works well where the number of reads of a given piece of data, within its lifetime, are high. Also where the ratio of reads to writes is high.

The motivation here is to bake it into Microprofile, so that other services can rely on its presence. Also, so that developers naturally add caching at the design stage and not as a hasty afterthought when production problems are experienced.

Community Support

Looking at the voting results as they stood at JavaOne, (https://docs.google.com/viewer?a=v&pid=forums&srcid=MDYzNzM2MDU2MjAwNjA2ODYwNTkBMTgwNjg4ODU4ODQwOTU2NDY1MzkBYVJIbHlRQnlCUUFKATAuMQEBdjI) JCache inclusion is a popular choice with about 430 votes so far. 36% of voters voted for this feature.

Vibrant JCache Ecosystem

We now have 10 implementations of JCache covering most of the IMDGs and in-process caches. Jens Wilke maintains the JCache Test Zoo which tests all the available implementations.

All of these are open source with the exception of Oracle Coherence. Most vendors, with the exception of IBM already have a distributed JCache implementation they are already using in their stacks. And IBM are apparently working on it.

So this is a well-implemented spec which should not be difficult for vendors to add.

Spring thought it was a good idea

Spring moved quickly to add JCache support. And Spring Boot, their analogue to Microprofile also has it. They also allow use of the Caching annotations.

Relationship with other Standards

Java EE 8/9

Oracle have announced Java EE 8 due end of 2017. They are currently assessing community support for JCache inclusion, a process that wraps up 10 October. After that we should see if they will include JCache. Two years ago, in an earlier survey, JCache was the second most requested feature.

If JCache becomes included in EE 8, it will require Java 8 as the minimum language level so it would require JCache 2.0. Much should get resolved in the next month.

Sample Application - microservice-schedule

To test out ideas in this paper, the microservice-schedule application is being used.

Payara integrated Hazelcast as a JCache provider over a year ago so we can readily try out ideas and even go back to them with patches to change the way they work.

The Microprofile owners will not permit code to be added to the main repository until a feature is agreed across all, so this is being maintained at https://github.com/gregrluck/microprofile-conference.

The application shows the storage of schedule data in a cache rather than a Map. A more realistic example would use a persistent data store with the same data cached.

Java and JCache Versions

JCache 1.0 which is the released version uses Java 6 and higher.

JCache 1.1 which is near to release, also uses Java 6.

Because JCache 1.1 can include some changes and enhancements if indeed any are required for Microprofile, it is the minimum version.

JCache 2.0 is being discussed. It will require Java 8. The rest of Microprofile is Java 8 so this would be beneficial and would allow for example smooth integration with java.util.stream.

Specification Notes

Substitutability

A microprofile microservice should be able to swap out the caching implementation with no code changes.

The JCache specification does not define cache configuration which is usually declaratively configured in a vendor specific configuration file on the classpath.

Distributed Caching by Default

Distributed cache makes it easy to scale up and replicate the data. Once a node is down, data is provided by other nodes, avoiding data losses. Although JCache does not require that caches must be distributed, Microprofile can add this as a requirement.

Microservices are intended to have multiple and sometimes many instances providing the service. Because of the N* problem, overall cache efficiency declines linearly as new instances are added where in-process caches are used.

JCache allows for both in-process and distributed implementations and therefore does not mandate objects to be Serializable. However distributed caches must have Serializable keys and values. For that reason, all cache providers for Microprofile must be distributed caches and objects to be cached must be `Serializable`.

Single Caching Provider

JCache has the concept of a caching provider. The spec supports discovery of multiple providers. However in a single Microservice, that is overkill.

Microprofile should support resolving a single CachingProvider via Caching:

```
/**
* Obtain the {@link CachingProvider} that is implemented by the specified
* fully qualified class name using the {@link #getDefaultClassLoader()}.
* Should this {@link CachingProvider} already be loaded it is simply returned,
* otherwise an attempt will be made to load and instantiate the specified
* class (using a no-args constructor).
*
* @param fullyQualifiedClassName the fully qualified class name of the
*                                {@link CachingProvider}
* @return the {@link CachingProvider}
* @throws CacheException    if the {@link CachingProvider} cannot be created
* @throws SecurityException when the operation could not be performed
*                           due to the current security settings
*/
public static CachingProvider getCachingProvider(String
fullyQualifiedClassName) {
 return CACHING_PROVIDERS.getCachingProvider(fullyQualifiedClassName);
}
```

To use Hazelcast as an example, a Microprofile vendor would use:

Caching.getCachingProvider(“`com.hazelcast.cache.HazelcastCachingProvider`“);

Because the vendor uses the fully qualified provider resolution, other providers can exist in classpath without conflict, should the application wish to use them.

Singleton Cache Manager

One again, though the spec allows for multiple CacheManagers per application, that is overkill and simpler to have one.

The default CacheManager is obtained using:

CachingProvider.getCacheManager();

```
/**
```

```
* Requests a {@link CacheManager} configured according to the
* {@link #getDefaultURI()} and {@link #getDefaultProperties()} be made
* available that using the {@link #getDefaultClassLoader()} for loading
* underlying classes.
* <p>
* Multiple calls to this method must return the same {@link CacheManager}
* instance, except if a previously returned {@link CacheManager} has been
* closed.
*
* @throws SecurityException when the operation could not be performed
*                           due to the current security settings
*/
CacheManager getCacheManager();
```

Injection of the Singleton CacheManager

With the above simplifications, we now have a singleton CacheManager from a single provider.

A microprofile service must allow the CacheManager to be injected which follows the rules above.

It is recommended that the CacheManager not be created until it is first used by the application to encourage fast startup.

```
@Inject
private CacheManager cacheManager;
```

Example. See usage in CacheProducer.

Shutdown of the Singleton CacheManager

It is important that the cache manager is shutdown when the microservice instances is shutdown.

A Microprofile service must shutdown the CacheManager on Microservice shut down.

This is done by calling cacheManager.close();

```
/**
* Closes the {@link CacheManager}.
* <p>
* For each {@link Cache} managed by the {@link CacheManager}, the
* {@link Cache#close()} method will be invoked, in no guaranteed order.
* <p>
* If a {@link Cache#close()} call throws an exception, the exception will be
```

```
     * ignored.
     * <p>
     * After executing this method, the {@link #isClosed()} method will return
     * <code>true</code>.
     * <p>
     * All attempts to close a previously closed {@link CacheManager} will be
     * ignored.
     *
     * Closing a CacheManager does not necessarily destroy the contents of the
     * Caches in the CacheManager.
     * <p>
     * It simply signals that the CacheManager is no longer required by the application
     * and that future uses of a specific CacheManager instance should not be permitted.
     * <p>
     * Depending on the implementation and Cache topology,
     * (e.g. a storage-backed or distributed cache), the contents of closed Caches
     * previously referenced by the CacheManager, may still be available and accessible
     * by other applications.
     *
     * @throws SecurityException when the operation could not be performed due to the
     *         current security settings
     */
    void close();
```

Cache Creation and Injection

Provided a Microprofile implementation makes a CacheManager available via injection, the rest is up to the developer.

Example: CacheProducer

In the following example we create and configure a cache called “schedule” using init with a `@PostConstruct` annotation and then make it available for injection with the annotation `@ScheduleCache`.

```
/**
 * @author mike
 */
@ApplicationScoped
public class CacheProducer {

 @Inject
 private CacheManager cm;

 private Cache<LongKey, Schedule> scheduleCache;
```

```
 @PostConstruct
 public void init() {
   scheduleCache = cm.createCache("schedule", new MutableConfiguration<LongKey, Schedule>());
 }

 @Produces
 @ApplicationScoped
 @ScheduleCache
 public Cache<LongKey, Schedule> getCache() {
    return scheduleCache;
 }

}
```

This is however all done by the end user. They could equally declare the cache in the Caching Provider's configuration file. They can inject it or they can get it imperatively using cacheManager.getCache("scheduleCache").

Imperative Cache Use

If someone wants to do caching imperatively , once they have a reference to a Cache they just do it in the ordinary way. See https://github.com/jsr107/jsr107spec/blob/master/src/main/java/javax/cache/Cache.java

Example: ScheduleDAO

CDI JCache Annotations Cache Use

JCache has defined Caching annotations. See https://github.com/jsr107/jsr107spec/tree/master/src/main/java/javax/cache/annotation

CDI is already in Microprofile. In JCache we defined standard POJO annotations and have already provided a sample implementation for CDI. See https://github.com/jsr107/RI/tree/master/cache-annotations-ri/cache-annotations-ri-cdi.

A Microprofile implementation must provide annotations processing for JCache annotations.

So we add caching annotations to Microprofile in the same way that Spring, Spring Boot, Payara and TomEE already have.

Example: ScheduleDAO

In this example we show how a schedule is removed from the cache. The body of this method would then normally remove it from the persistent store.

```
   @CacheRemove(cacheName="schedule", cacheKeyGenerator = LongKeyGenerator.class)
   public void deleteSchedule(Long scheduleId) {
//         if (scheduleId != null) {
//             scheduleCache.remove(new MyKey(scheduleId));
//         }
   }
```

Appendix 1: Further Caching Use Case Ideas

JCache is Not an IMDG

JCache is not a full In-Memory Data Grid API. It is a cache API. Hazelcast gets used heavily in the plumbing of microservices for clustering and async message passing. See https://hazelcast.com/resources/microservices-with-hazelcast/ for typically how this is done. However these usages do not use the JCache API.

There are many ways to solve these problems with IDMG as only one way.

There are no standard APIs for these additional things. So I think this should be left out of Microprofile.

However if something can be built on top of JCache's API then it could be included.

Caching Provider Interoperability

We could also discuss specification of a wired protocol to connect multiple cache providers into the same cluster, so that microservices based on different runtimes can share the same data (e.g. distribute data across Hazelcast and EHCache together)

Greg: It is an anti-pattern to share data bases/caches between separate Microservices

Lightweight service registry/discovery - Ondrej

Cache can be shared by multiple different applications, and each of them can store an object that defines how to access it (exposed REST services, node name, etc.) All other members of the cluster can list the data in the cache to discover other services.

Lightweight load balancing - Ondrej

JCache provides cache entry listeners that are ensured to be spawn only on a single node. We can easily achieve that requests are distributed to multiple nodes randomly. It is easy to scale up and provide additional nodes without any change in the application. When we define that cache expires immediately, putting a value into the cache works as a message, that is picked up by exactly one node.

Message passing - Ondrej

The above also provides base to implement message passing - yes, it is quite far away from the initial intent of caching data, but once the cache is distributed, all members in the cluster are connected and can communicate with each other. We just need to add asynchronous behavior to JCache and provide convenient API. (Note Asynchronous behaviour is coming in JCache 2.0).

Hazelcast is already a base for distributed CDI event bus available in Payara Micro and we can extend that to provide what is necessary.