
hazelcast
OPERATIONAL IN-MEMORY COMPUTING. SIMPLIFIED.
Distributed Caching
Distributed Computing
Distributed Messaging
Learn more at hazelcast.com | hazelcast.org

Get More Refcardz! Visit DZone.com/refcardz

CONTENTS

Java Caching: Strategies and the JCache API

By Abhishek Gupta

INTRODUCTION

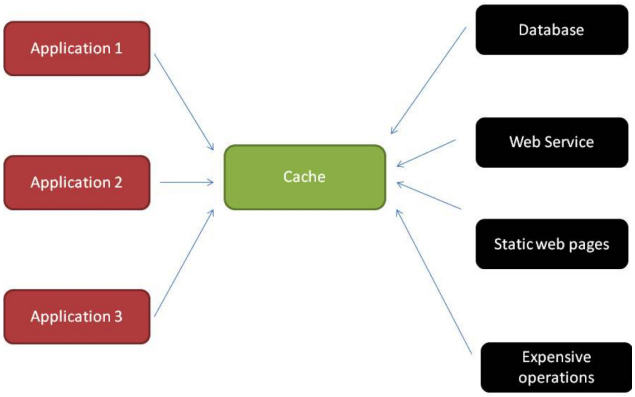
WHAT IS CACHING?

Generally speaking, caching is a technique wherein objects in your application are stored in a temporary storage area known as a cache. The cache itself can be thought of as an in-memory data structure. Cached objects could be anything from the results of expensive and time-consuming operations, to static webpages or the contents of your backend database. Caching insulates your application from the requirement of fetching the same information (from any data source) or executing the same expensive calculation repeatedly, by storing to and fetching from an intermediate cache when required.

Caching provides several benefits, which is why it is heavily leveraged in use cases where fast access to data is required. Proper caching can improve performance, decrease resource usage, and increase responsiveness.

Typically, in an enterprise application, the cache acts as an intermediate, low-latency data source and sits between the application and the backend systems.

Note: *This is a rather simplistic representation to get you started. We will cover more involved architectural topologies in detail later.*



JAVA CACHING LANDSCAPE

The Java ecosystem has had solid support for caching solutions with several products and frameworks available to choose from. Some widely used solutions are:

- Hazelcast
- Oracle Coherence
- Infinispan
- Ehcache
- Apache Ignite
- GridGain
- GemFire

In order to create a standard set of APIs to abstract over these heterogeneous caching solutions, *JSR 107: Java Temporary Caching API* was initiated.

JCACHE API (JSR 107)

WHAT IS JCACHE?

JCache is a Java API for caching. It provides a set of common interfaces and classes that can be used to temporarily store Java objects in memory. It is a JCP standard represented by JSR 107. It is one of the longest-running JSRs in the history of the JCP—starting with its inception in 2001 and ending with its final release in March 2014.

The JCache API is implemented by different vendors (a.k.a. caching providers), but the manner in which client code accesses it is via a standard set of interfaces and classes that it exposes.

WHY JCACHE?

Caching is not a new concept in the Java world—or even in general. As already mentioned, there are several enterprise-grade caching solutions already available—so why would you choose JCache over a specific vendor/product that you are already using in your applications?

There are two primary benefits:

Portability: JCache allows you to use a standard API in your client applications, decoupling them from the actual vendor implementation. This in turn makes your application portable between various JCache implementations.

Developer Productivity: Developers can leverage the common set of JCache APIs and interfaces without concerning themselves with vendor-specific details. This ensures that the learning curve is smaller since it is restricted to the knowledge

of JCache as a standard and not the specifics of a vendor implementation.

WHAT ABOUT VENDOR-SPECIFIC FEATURES?

JCache allows the use of equivalent vendor APIs corresponding to a specific JCache interface/class. This will be discussed in detail in the upcoming sections.

JCACHE DEEP DIVE

BASIC BUILDING BLOCKS

Here is a simple code snippet to get you started. The details will be explored in upcoming sections.

```
CachingProvider cachingProvider = Caching.
getCachingProvider();
CacheManager cacheManager = cachingProvider.
getCacheManager();
MutableConfiguration<String, String> config = new
MutableConfiguration();
Cache<String, String> cache = cacheManager.
createCache("JDKCodeNames",config);
cache.put("JDK1.5","Tiger");
cache.put("JDK1.6","Mustang");
cache.put("JDK1.7","Dolphin");
String jdk7CodeName = cache.get("JDK1.7");
```

CACHE

A *javax.cache.Cache* is a type-safe data structure that allows applications to store data temporarily (as key-value pairs). Its semantics are similar to a *java.util.Map* object, but it is not exactly the same as a Map.

The Cache interface exposes several features via its API, but some of the basic ones are listed in the following table:

METHOD	DESCRIPTION
put , putAll	Adds entries (key-value pairs) to a cache
containsKey	Checks if an entry with a key exists in the cache
get , getAll	Gets value(s) corresponding to key(s)
replace	Substitutes an existing value for a key with another value
remove , removeAll	Removes one or all entries from a cache

ENTRY

As its name indicates, the *javax.cache.Entry* represents a key-value pair in a cache. As conveyed earlier, there can be multiple such entries in a cache. An entry is a simple entity that exposes bare minimum operations, such as fetching the key and value via the *getKey* and *getValue* methods respectively.

CACHEMANAGER

The *javax.cache.CacheManager* interface helps deal with cache objects and executes operations like cache creation, destruction, and introspection (fetching relevant details about itself). Let's look at some of the common operations:

METHOD	DESCRIPTION
createCache	Create a cache
destroyCache	Destroy a cache
getCache (and its overloaded forms)	Search for a (possibly) existing cache
getProperties, getURI, isClosed, getClassLoader, etc.	Provide information about the cache

CACHINGPROVIDER

Part of the JCache SPI, the *javax.cache.spi.CachingProvider* provides methods that allow applications to manage CacheManager instances.

METHOD	DESCRIPTION
getCacheManager (and its overloaded versions)	Create CacheManager instance
close (and its overloaded versions)	Destroy/close CacheManager instance
getDefaultClassLoader, getDefaultProperties, getDefaultURI, isSupported	Obtain information about the CachingProvider

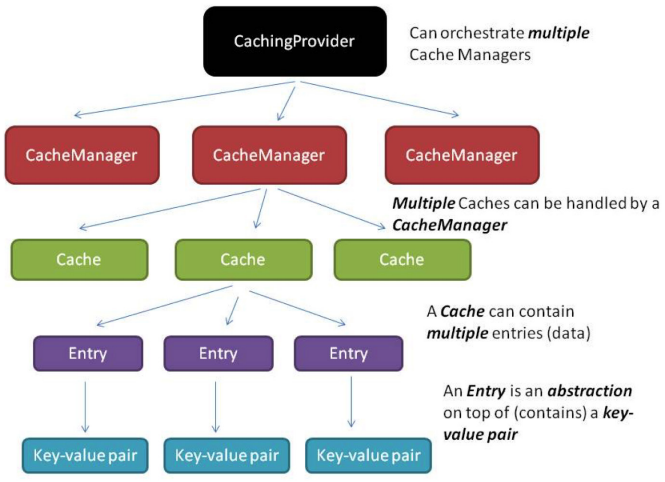
CACHING

This class is used to obtain a handle to a *javax.cache.CacheProvider* object via *getCachingProvider* and its overloaded versions.

Note: *There are other ways in which the CachingProvider might be obtained.*

Apart from this, the Caching class also provides other capabilities as noted in the following table:

METHOD	DESCRIPTION
getCachingProviders	Provides a list of all *CachingProvider* instances
getCache	Allows the client code to search for a *javax.cache.Cache* object using its name



API OVERVIEW

It's good to think of JCache as a set of modules, with each module delivering a specific feature. The JCache API is so split up that it's easy and intuitive to explore it in this manner.

JCACHE MODULE/FEATURE	API PACKAGE
Core	javax.cache
Configuration	javax.cache.configuration
Expiration Policies	javax.cache.expiry
Listeners and Filters	javax.cache.event
Processing	javax.cache.processing
External Resource Integration	javax.cache.integration
Annotations	javax.cache.annotation
Management	javax.cache.management
SPI/Extensions	javax.cache.spi

CORE

The APIs introduced in the **Basic Building Blocks** section form the bulk of what we can call the *core* of JCache, since these classes/interfaces (and their respective provider implementations) are fundamental to the API.

CONFIGURATION

The JCache API offers a standard set of interfaces and implementations with which one can configure a cache programmatically. The *javax.cache.configuration.MutableConfiguration* and its builder-like API aids in the configuration.

The following cache characteristics can be configured:

METHOD	JCACHE CONFIGURATION ARTIFACT
setTypes	Data types to be stored in the Cache
setStoreByValue	Store by reference or value
setExpiryPolicyFactory	Expiration Policy
setReadThrough, setWriteThrough	Read through and Write through policies
setCacheLoaderFactory, setCacheWriterFactory	Loader and Writer implementation
addCacheEntryListenerConfiguration	Entry Listener implementation
setManagementEnabled, setStatisticsEnabled	Management and statistics activation

```
MutableConfiguration<String,String> config = new
MutableConfiguration();
config.setReadThrough(true);
```

EXPIRATION POLICIES

As the name suggests, an expiry policy can be enforced on a cache, which would determine auto-eviction or removal of entries from your cache as per the policy semantics.

Note: *By default, the entries in a* javax.cache.Cache *do not expire.*

The expiry policies provided by the JCache API revolve around the *javax.cache.expiry.ExpiryPolicy* interface and its ready-to-use implementations.

EXPIRATION POLICY IMPLEMENTATION CLASS	DESCRIPTION
AccessedExpiryPolicy	Based on time of last access
CreatedExpiryPolicy	Based on creation time
EternalExpiryPolicy	Ensures the cache entries never expire (default expiry policy)
ModifiedExpiryPolicy	Based on time of last update
TouchedExpiryPolicy	Based on time of last access OR update

```
MutableConfiguration<String,String> config = new
MutableConfiguration();
config.setExpiryPolicyFactory(CreatedExpiryPolicy.
factoryOf(Duration.ONE_MINUTE));
```

LISTENERS AND FILTERS

Cache event listeners allow client code to register callbacks against the cache events they might be interested in. *javax.cache.event.CacheEntryListener* acts as a (parent) marker interface for other interfaces that provide a contract (method) which can be implemented in order to react to a specific event in the cache. These are typically *single abstract methods*, which makes them perfectly suitable for Java 8 lambda-style implementation.

METHOD	EVENT TYPE	LISTENER INTERFACE
onCreated	CacheEntryEvent.CREATED	CacheEntryCreatedListener
onUpdated	CacheEntryEvent.UPDATED	CacheEntryUpdatedListener
onExpired	CacheEntryEvent.EXPIRED	CacheEntryExpiredListener
onRemoved	CacheEntryEvent.REMOVED	CacheEntryRemovedListener

JCache also has filters, which help determine whether or not to call the cache listeners. This comes in handy when you want to selectively dispatch calls to cache listeners based on certain conditions.

```
CacheEntryCreatedListener<Long,TicketDetails>
newTicketCreationListener = (cacheEntries) -> {
          for(CacheEntryEvent ticketCreatedEvent :
cacheEntries){
               System.out.println("Ticket ID: " +
ticketCreatedEvent.getKey());
               System.out.println("Ticket Details: " +
ticketCreatedEvent.getValue().toString());
          }
};
CacheEntryEventFilter<Long,TicketDetails>
entryEventFilter = (event) -> event.getSource().getName().
equals("TicketsCache");
```

EXTERNAL RESOURCE INTEGRATION

The JCache API supports cache loaders and cache writers, which help integrate the cache with external resources. A read-through operation is accomplished with the help of a *javax.cache.CacheLoader* implementation (which is automatically invoked if a key is not found in the cache) that retrieves the value for the corresponding key from an external source. Similarly, a *javax.cache.CacheWriter* implementation synchronizes an external source in response to updates and removal of entries in the cache.

INTEGRATION FEATURE	INTERFACE AND METHODS	PRE-REQUISITE CONFIGURATION
Read-through	CacheLoader [load, loadAll]	setReadThrough(true)
Write-through	CacheWriter [delete, deleteAll, write, writeAll]	setWriteThrough(true)

```
public class TicketCacheLoaderWriter implements
CacheLoader<Long, TicketDetails>, CacheWriter<Long,
TicketDetails>{

        @Override
        public TicketDetails load(Long ticketID) throws
CacheLoaderException {
            return getTicketDetails(ticketID);
        }
        @Override
        public Map<Long, TicketDetails> loadAll(Iterable<?
extends Long> ticketIDs) throws CacheLoaderException {
            Map<Long, TicketDetails> tickets = new
HashMap<>();
            for(Long ticketID : ticketIDs){
                tickets.put(ticketID,
getTicketDetails(ticketID));
            }

            return Collections.unmodifiableMap(tickets);
        }

        private TicketDetails getTicketDetails(Long
ticketID){
            TicketDetails details = null;
            //business logic to fetch ticket information
            return details;
        }

        @Override
        public void write(Cache.Entry<? extends
Long, ? extends TicketDetails> ticketEntry) throws
CacheWriterException {
            writeTicketDetails(ticketEntry.getKey(),
ticketEntry.getValue());
        }
        @Override
        public void writeAll(Collection<Cache.Entry<?
extends Long, ? extends TicketDetails>> ticketEntries)
throws CacheWriterException {
            for(Cache.Entry ticketEntry : ticketEntries){
                writeTicketDetails((Long) ticketEntry.
getKey(), (TicketDetails) ticketEntry.getValue());
            }
        }
        @Override
        public void delete(Object ticketID) throws
CacheWriterException {
            deleteTicket((Long) ticketID);
        }
        @Override
        public void deleteAll(Collection<?> ticketIds)
throws CacheWriterException {
            for(Object ticketID : ticketIds){
                deleteTicket((Long) ticketID);
            }
        }

        private void writeTicketDetails(Long ticketID,
TicketDetails ticketDetails){
            //business logic to delete ticket information
        }

        private void deleteTicket(Long ticketID){
            //business logic to delete ticket information
        }
}
```

CACHE ENTRY PROCESSING

An entry processor is particularly useful when your cache is distributed (which is quite often the case) over multiple nodes (JVMs). In order to update an existing entry in the cache, one might follow the default set of steps:

- Get the value from the cache
- Mutate/update it
- Put the updated value back into the cache

While this is perfectly normal, it is not efficient in terms of performance (especially when the cache values are large). The caching provider has to de-serialize the cache value from one of the many nodes to the client and then send the updated (and serialized) version back to the cache. The problem is magnified if multiple such calls are made in succession.

Entry processors allow the client to *apply a transformation* on the cache entry by sending it over to the cache node rather than fetching the entry from the cache and then mutating it locally. This significantly reduces the network traffic as well as serialization/de-serialization expenses. All you need to do is define/implement the entry processor represented by the *javax.cache.EntryProcessor* interface and specify the same during the *Cache.invoke* or *Cache.invokeAll* methods.

METHOD	API	DESCRIPTION
process	javax.cache. EntryProcessor	Invoked as a result of calling *Cache.invoke* or *Cache.invokeAll*
get	javax.cache. EntryProcessorResult	Part of the Map returned by the *Cache.invokeAll* method (one for each key). It wraps the result returned by the entry processor

```
EntryProcessor<String,String,String> processor = (e,p) ->
{
            String key = e.getKey();
            String oldValue = e.getValue();
            Function<String,String> transformation =
(Function<String,String>) p[0];
            return transformation.apply(oldValue);
};
```

ANNOTATIONS

JCache annotations (in the *javax.cache.annotation* package) help you treat caching operations as aspects (from an aspect-oriented paradigm perspective). These annotations allow the client code to specify caching requirements in a declarative fashion. Note that the efficacy of these annotations is dependent on an external framework (like CDI, Guice, etc.), which can process these and execute the required operations.

By default, JCache exposes a limited number of caching operations via annotations, but they are useful nonetheless.

ANNOTATION	DESCRIPTION
@CacheResult	Searches the cache for a key (the method parameter), invokes the method if the value cannot be found, and caches the same for future invocations
@CachePut	Executes a cache put with key and value in the method parameters
@CacheRemove	Removes a cache entry by using the key specified in the method parameter
@CacheRemoveAll	Removes ALL cache entries

Note: *The above-mentioned annotations are applicable on a class (which essentially enforces them for all the methods in that class) or for one or more method(s).*

There are also three auxiliary annotations worth mentioning:

ANNOTATION	DESCRIPTION
@CacheDefaults	Helps specify default configurations for aforementioned annotations. Applicable for a class
@CacheKey	Used to explicitly specify a method parameter as a cache key
@CacheValue	Used to explicitly specify a method parameter as a cache value when using the @CachePut annotation

```
@CacheDefaults(cacheName=”TicketsCache”)
public class TicketingService{
@CachePut
public void persistTicket(long ticketID, @CacheValue
TicketDetails details){
   //domain logic to persist ticket information
}
@CacheResult
public TicketDetails findTicket(@CacheKey long ticketID){
        TicketDetails details = null;
        //execute domain logic to find ticket information
        return details;
}
@CacheRemove
public void deleteTicket(@CacheKey long ticketID){
        //domain logic to delete ticket information
}
@CacheRemoveAll
public void deleteAllTickets(){
        //domain logic to delete ticket information
}
}
```

MANAGEMENT

JCache provides MBean interfaces whose implementations expose cache configuration and runtime performance monitoring related statistics. These statistics can be tracked through any JMX client or through the *javax.management.MBeanServer* API (programmatic).

MXBEAN	OPERATIONS	PRE-REQUISITES
CacheMXBean	getKeyType, getValueType, isManagementEnabled, isReadThrough, isStatisticsEnabled, isStoreByValue, isWriteThrough	setReadThrough(true)
CacheStatistics MXBean	Clear, getAverageGetTime, getAveragePutTime, getAverageRemoveTime, getCacheEvictions, getCacheGets, getCacheHitPercentage, getCacheHits, getCacheMisses, getCacheMissPercentage, getCachePuts, getCacheRemovals	setWriteThrough(true)

```
MutableConfiguration<String,String> config = new
MutableConfiguration();
config.setManagementEnabled(true);
config.setStatisticsEnabled(true);
```

SPI/EXTENSIONS

The *javax.cache.spi* package consists of a single interface: *CachingProvider*. We have already looked at the specific details of this class, but let's understand it from a JCache vendor perspective.

A JCache provider implements this interface, and for it to be auto-discoverable, the full qualified class name of the concrete class is declared in *META-INF/services/javax.cache.spi.CachingProvider*—it must be available in the class path. Generally, this is packaged in the vendor implementation JAR itself. So you can think of this interface as the gateway to your JCache provider.

```
CacheManager cacheManager = Caching.
getCachingProvider(“com.hazelcast.cache.impl.
HazelcastCachingProvider”).getCacheManager();
CacheManager cacheManager = Caching.
getCachingProvider(“com.tangosol.coherence.jcache.
CoherenceBasedCachingProvider”).getCacheManager();
```

BEST OF BOTH WORLDS: USING VENDOR-SPECIFIC FEATURES ALONG WITH JCACHE

By now I am sure you understand that JCache (JSR 107) is just a standard set of APIs that are implemented by different vendors. The standard JCache APIs provide you with a hook to tap into the concrete vendor-specific implementation itself—you can do so using the **unwrap** method. Let's look at the details in the table below.

Note: *This is obviously not recommended if true CachingProvider portability is what you are looking for since your application would be directly coupled to vendor-specific APIs.*

METHOD	AVAILABLE IN CLASS/INTERFACE	DESCRIPTION
unwrap	javax.cache. CacheManager	Get a handle to the CacheManager implementation of a provider
	javax.cache.Cache	Get a handle to the Cache implementation of a provider
	javax.cache.Cache.Entry	Get a handle to the Cache.Entry implementation of a provider
	javax.cache.annotation. CacheInvocationContext	Get a handle to the CacheInvocationContext implementation of a provider

```
//Hazelcast specific example
ICache<String,String> hazelcastICache = cache.
unwrap(ICache.class);
```

The Hazelcast ICache extension for JCache provides lots of value added features. It's not possible to discuss all of them in detail, but here are some of the important ones:

Asynchronous Operations: The ICache extension exposes asynchronous equivalents for most of the JCache operations like get, put, putIfAbsent, etc.

Near Cache: This feature allows Hazelcast clients (via explicit configuration) to store data locally rather than reaching out to remote Hazelcast cluster, thus reducing latency and network traffic.

CACHING STRATEGIES

Caching strategies are methodologies one might adopt while implementing a caching solution as part of an application. The

drivers/use cases behind the requirement of a caching layer vary based on the application's requirements.

Let's look at some of the commonly adopted caching strategies and how JCache fits into the picture. The following table provides a quick snapshot followed by some details:

FACTOR	STRATEGY
Cache topology	Standalone, Distributed, Replicated
Cache modes	Embedded cache or Isolated cache
Transparent cache access	Read-Through and Write-Through caching
Cache data quality	Expiry / Eviction policy fine tuning

CACHE TOPOLOGY

Which caching topology/setup is best suited for your application? Does your application need a single-node cache or a collaborative cache with multiple nodes?

STRATEGIES/OPTIONS

From a specification perspective, JCache does not include any details or presumptions with regards to the cache topology.

Standalone: As the name suggests, this setup consists of a single node containing all the cached data. It's equivalent to a single-node cluster and does not collaborate with other running instances.

Distributed: Data is spread across multiple nodes in a cache such that only a single node is responsible for fetching a particular entry. This is possible by distributing/partitioning the cluster in a balanced manner (i.e., all the nodes have the same number of entries and are hence load balanced). Failover is handled via configurable backups on each node.

Replicated: Data is spread across multiple nodes in a cache such that each node consists of the complete cache data, since each cluster node contains **all** the data; failover is not a concern.

CACHE MODES

Do you want the cache to run in the same process as your application, or would you want it to exist independently (*as-a-service* style) and execute in a client-server mode?

STRATEGIES/OPTIONS

JCache does not mandate anything specific as far as cache modes are concerned. It embraces these principles by providing flexible APIs that are designed in a cache-mode agnostic manner.

The following modes are common across caches in general:

Embedded mode: When the cache and the application co-exist within the same JVM, the cache can be said to be operating in embedded mode. The cache lives and dies with the application JVM. This strategy should be used when:

- Tight coupling between your application and the cache is not a concern
- The application host has enough capacity (memory) to accommodate the demands of the cache

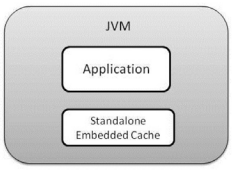
Client-Server mode: In this setup, the application acts as the client to a standalone (remote) caching layer. This should be leveraged when:

- The caching infrastructure and application need to evolve independently
- Multiple applications use a unified caching layer which can be scaled up without affecting client applications

MULTIPLE COMBINATIONS TO CHOOSE FROM

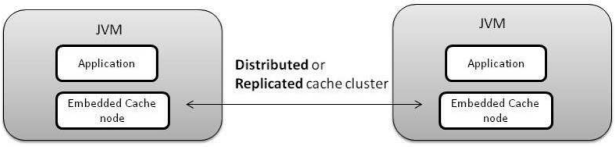
Different cache modes and topologies make it possible to have multiple options to choose from, depending upon specific requirements.

Standalone Embedded Cache: A single cache node within the same JVM as the application itself



Distributed Embedded Cache: Multiple cache (clustered) nodes, each of which is co-located within the application JVM and is responsible for a specific cache entry only

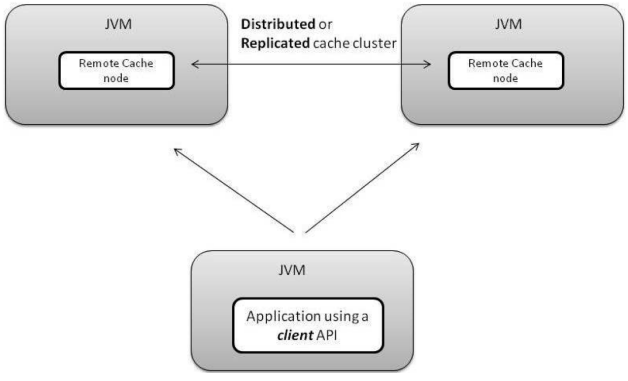
Replicated Embedded Cache: Multiple cache (clustered) nodes, each of which is co-located within the application JVM; here the cached data is replicated to all the nodes



Standalone Client-Server Cache: A single cache node running as a separate process than that of the application

Distributed Client-Server Cache: Multiple cache (clustered) nodes, collaborating in a distributed fashion and executing in isolation from the client application

Replicated Client-Server Cache: Multiple cache (clustered) nodes, where the entire cache data copy is present on each node, and the cache itself is run as a separate process than that of the application



TRANSPARENT CACHE ACCESS

You are designing a generic caching layer for heterogeneous applications. Ideally, you would want your application to access the cache transparently without polluting its core logic with the specifics of the caching implementation.

STRATEGIES/OPTIONS

As already stated in the **JCache Deep Dive** section, integration to external systems—like databases, file stores, LDAP, etc.—is already abstracted via the CacheLoader and CacheWriter mechanisms, which help you implement Read-Through and Write-Through strategies respectively.

Read-Through: A process by which a missing cache entry is fetched from the integrated backend store.

Write-Through: A process by which changes to a cache entry (create, update, delete) are pushed into the backend data store.

It is important to note that the business logic for *Read-Through* and *Write-Through* operations for a specific cache are confined within the caching layer itself. Hence, your application remains insulated from the specifics of the cache and its backing system-of-record.

OTHER STRATEGIES (NON-JCACHE)

Write-Behind: This strategy leverages a more efficient approach wherein the cache updates are batched (queued) and *asynchronously* written to the backend store instead of an eager and synchronous policy adopted by the Write-Through strategy. For example, Hazelcast supports the Write-Behind strategy via its *com.hazelcast.core.MapStore* interface when the *write-delay-seconds* configuration property is greater than 0. *Please note that this is purely a Hazelcast Map feature and is not supported through ICache extension.*

Refresh-Ahead: This is another interesting strategy where a caching implementation allows you to refresh the cache data from the backend store depending upon a specific factor, which can be expressed in terms of an entry's expiration time. The reload process is asynchronous in nature. For example, Oracle Coherence supports this strategy, which is driven by a configuration element known as the *refresh-ahead factor*, which is a percentage of the expiration time of a cache entry.

CACHE DATA QUALITY

Cache expiry/eviction policies are vital from a cache data quality perspective. If a cache is not tuned to expire its contents, then it will not get a chance to refresh/reload/sync up with its master repo/system-of-record and might end up returning stale or outdated data.

You need to ensure that the caching implementation takes into account the data volatility in the backing data store (behind the cache) and effectively tune your cache to maintain quality data.

STRATEGIES/OPTIONS

In the **JCache Deep Dive** section, you came across the default expiry policies available in JCache—AccessedExpiryPolicy, CreatedExpiryPolicy, EternalExpiryPolicy, ModifiedExpiryPolicy, and TouchedExpiryPolicy. In addition to these policies, JCache allows you to implement custom eviction policies by implementing the *javax.cache.expiry.ExpiryPolicy* interface.

FLEXI-EVICTION (NON-JCACHE)

The JCache API allows you to enforce expiry policies on a specific cache; as a result, it is applicable to **all** the entries in that cache. With the Hazelcast JCache implementation, you can fine-tune this further by providing the capability to specify the ExpiryPolicy per entry in a cache. This is powered by the *com.hazelcast.cache.ICache* interface.

```
ICache<String,String> hazelcastICache = cache.
unwrap(ICache.class);
hazelcastICache.put("key2", "value2", new
TouchedExpiryPolicy(Duration.FIVE_MINUTES));
```

ABOUT THE AUTHOR

Abhishek Gupta is a consultant and developer specializing in the Oracle Identity Governance suite, as well as a Java EE developer and architect. His areas of interests include Java EE, object oriented design, scalable architecture and (lately) distributed computing. He loves writing about and evangelizing Java EE technologies and frequently blogs at abhirockzz.wordpress.com.

CREDITS:

Editor: **G. Ryan Spain** | Designer: **Yassee Mohebbi** | Production: **Chris Smith** | Sponsor Relations: **Chris Brumfield** | Marketing: **Chelsea Bosworth**

BROWSE OUR COLLECTION OF 250+ FREE RESOURCES, INCLUDING:

RESEARCH GUIDES: Unbiased insight from leading tech experts

REFCARDZ: Library of 200+ reference cards covering the latest tech topics

COMMUNITIES: Share links, author articles, and engage with other tech experts

JOIN NOW

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

Copyright © 2015 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

VERSION 1.0 **$7.95**