Introduction

============

Ehcache is a pure Java, in-process cache with the following features:

1. Fast and Light Weight

1.1 Fast

1.2 Simple

1.3 Small foot print

1.4 Minimal dependencies

2. Scalable

2.1 Provides Memory and Disk stores for scalabilty into gigabytes

2.4 Scalable to hundreds of caches

2.3 Tuned for high concurrent load on large multi-cpu servers

2.4 Multiple CacheManagers per virtual machine

3. Flexible

3.1 Supports Object or Serializable caching

3.2 Support cache-wide or Element-based expiry policies

3.3 Provides LRU, LFU and FIFO cache eviction policies

3.4 Provides Memory and Disk stores

3.5 Distributed

3.6 Dynamic, Runtime Configuration of Caches

4. Standards Based

4.1 Full implementation of JSR107 JCACHE API

5. Extensible

5.1 Listeners may be plugged in

5.2 Peer Discovery, Replicators and Listeners may be plugged in

5.3 Cache Extensions may be plugged in

5.4 Cache Loaders may be plugged in

5.5 Cache Exception Handlers may be plugged in

6. Application Persistence

6.1 Persistent disk store which stores data between VM restarts

6.2 Flush to disk on demand

7. Listeners

7.1 CacheManager listeners

7.2 Cache event listeners

8. JMX Enabled

9. Distributed Caching

10. Support for replication via RMI or JGroups

10.1 Peer Discovery

10.2 Reliable Delivery

10.3 Synchronous Or Asynchronous Replication

10.4 Copy Or Invalidate Replication

10.5 Transparent Replication

10.6 Extensible

10.7 Bootstrapping from Peers

11. Cache Server

11.1 RESTful cache server

11.2 SOAP cache server

11.3 comes as a WAR or as a complete server

12. Java EE and Applied Caching

12.1 Blocking Cache to avoid duplicate processing for concurrent operations

12.2 SelfPopulating Cache for pull through caching of expensive operations

12.3 Java EE Gzipping Servlet Filter

12.4 Cacheable Commands

12.5 Works with Hibernate

12.6 Works with Google App Engine

12.7 Transactional support through JTA

13. High Quality

13.1 High Test Coverage

13.2 Automated Load, Limit and Performance System Tests

13.3 Specific Concurrency Testing

13.4 Production tested

13.5 Fully documented

13.6 Trusted by Popular Frameworks

13.7 Conservative Commit policy

13.8 Full public information on the history of every bug

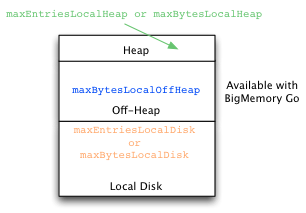
13.9 Responsiveness to serious bugs

14. Open Source Licensing

14.1 Apache 2.0 license

Ehcache has three storage tiers, as explained below:

1. **Memory store**
2. It is the Top tier as Heap memory that holds a copy of the hottest subset of data.
3. Its overall size is taken from the Java heap size.
4. It does not require any special configuration.
5. Since it exists in the heap, it is limited by Java GC(Garbage Collection) constraints.
6. Fastest storage option.
7. **Off-heap store** – Limited in size only by available RAM. Not subject to Java GC. Can store serialized data only. Provides overflow capacity to the memory store.
8. **Disk store** – Backs up in-memory data and provides overflow capacity to the other tiers. Can store serialized data only.



1. **Memory Store**

**Configuring Memory Store**

* It is always enabled and exists in heap memory.
* Allot as much heap memory as possible without triggering garbage collection (GC) pauses for better performance.

**Memory Use, Spooling, and Expiry Strategy in the Memory Store**

* When an element is added to a cache and it goes beyond its maximum memory size, an existing element is either deleted, if overflow is not enabled, or evaluated for spooling to another tier, if overflow is enabled.
* The overflow options are overflowToOffHeap and <persistence> (disk store).
* If overflow is enabled, a check for expiry is carried out. If it is expired it is deleted; if not it is spooled.
* The eviction of an item from the memory store is based on the optional MemoryStoreEvictionPolicy attribute specified in the configuration file.

**Memory Store Eviction Policy has some specific values which are:**

* **Least Recently Used (LRU)**

1. LRU is the default setting.
2. The last-used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.

* **Least Frequently Used (LFU)**

1. For each get call on the element the number of hits is updated.
2. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element with least number of hits, the Less Frequently Used element, is evicted.

* **First In First Out (FIFO)**

1. Elements are evicted in the same order as they come in.
2. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

Storage Tiers

You can divide a cache or in-memory data set across the following storage areas, referred to as tiers:

\*MemoryStore – On-heap memory used to hold cache elements. This tier is subject to Java garbage collection.

\*OffHeapStore – Provides overflow capacity to the MemoryStore. Limited in size only by available RAM. Not subject to Java garbage collection (GC). Available only with Terracotta BigMemory products.

\*DiskStore – Backs up in-memory cache elements and provides overflow capacity to the other tiers.

MemoryStore

The memory store is always enabled and exists in heap memory. It has the following characteristics:

\*It accepts all data, whether serializable or not.

\*It is the fastest storage option.

\*Is thread safe for use by multiple concurrent threads.

If you use OffHeapStore (available with the BigMemory products only), MemoryStore holds a copy of the hottest subset of data from the OffHeapStore.

All caches specify their maximum in-memory size, in terms of the number of elements, at configuration time.

When an element is added to a cache and it goes beyond its maximum memory size, an existing element is either deleted, if overflow is not enabled, or evaluated for spooling to another tier, if overflow is enabled.

If overflow is enabled, a check for expiry is carried out. If it is expired it is deleted; if not it is spooled.

For information about sizing and configuring the MemoryStore, see “Configuring Memory Store” in the Ehcache Configuration Guide .

OffHeapStore

The OffHeapStore extends a cache to memory outside the of the Java heap. This store, which is not subject to Java garbage collection (GC), is limited only by the amount of RAM available. Using OffHeapStore, you can create extremely large local caches. OffHeapStore is only available with the Terracotta BigMemory products.

Because off-heap data is stored in bytes, only data that is Serializable is suitable for the OffHeapStore. Any non serializable data overflowing to the OffHeapMemoryStore is simply removed, and a WARNING level log message is emitted.

Since serialization and deserialization take place on putting and getting from the off-heap store, it is theoretically slower than the MemoryStore. This difference, however, is mitigated when garbage collection associated with larger heaps is taken into account.

For the best performance, you should allocate to a cache as much heap memory as possible without triggering GC pauses. Then, use the OffHeapStore to hold the data that cannot fit in heap (without causing GC pauses).

For information about sizing and configuring OffHeapStore, see “Configuring OffHeapStore” in the Configuration Guide for your BigMemory product.

DiskStore

The DiskStore provides a thread-safe disk-spooling facility that can be used for either additional storage or persisting data through system restarts.

Note:

The DiskStore tier is available only for local (standalone) instances of cache. When you use a distributed cache (available only in BigMemory Max), a Terracotta Server Array is used instead of a disk tier.

Only data that is Serializable can be placed in the DiskStore. Writes to and from the disk use ObjectInputStream and the Java serialization mechanism. Any non-serializable data overflowing to the disk store is removed and a NotSerializableException is thrown.Be aware that serialization speed is affected by the size of the objects being serialized and their type. For example, it has been shown that:

\*The serialization time for a Java object consisting of a large Map of String arrays was 126ms, where the serialized size was 349,225 bytes.

\*The serialization time for a byte[] was 7ms, where the serialized size was 310,232 bytes.

Byte arrays are 20 times faster to serialize, making them a better choice for increasing disk-store performance.

Configuring a disk store is optional. If all caches use only memory and off-heap stores, then there is no need to configure a disk store. This simplifies configuration, and uses fewer threads.

CacheManager Configuration

==========================

An ehcache.xml corresponds to a single CacheManager.

See instructions below or the ehcache schema (ehcache.xsd) on how to configure.

System property tokens can be specified in this file which are replaced when the configuration

is loaded. For example multicastGroupPort=${multicastGroupPort} can be replaced with the

System property either from an environment variable or a system property specified with a

command line switch such as -DmulticastGroupPort=4446. Another example, useful for Terracotta

server based deployments is <terracottaConfig url="${serverAndPort}"/ and specify a command line

switch of -Dserver36:9510

The attributes of <ehcache> are:

\* name - an optional name for the CacheManager. The name is optional and primarily used

for documentation or to distinguish Terracotta clustered cache state. With Terracotta

clustered caches, a combination of CacheManager name and cache name uniquely identify a

particular cache store in the Terracotta clustered memory. There is a restriction on characters

in the name in case MBeans are used. See the section describing restrictions for unquoted values

in the javax.management.ObjectName javadoc.

\* updateCheck - an optional boolean flag specifying whether this CacheManager should check

for new versions of Ehcache over the Internet. If not specified, updateCheck="true".

\* dynamicConfig - an optional setting that can be used to disable dynamic configuration of caches

associated with this CacheManager. By default this is set to true - i.e. dynamic configuration

is enabled. Dynamically configurable caches can have their TTI, TTL and maximum disk and

in-memory capacity changed at runtime through the cache's configuration object.

\* monitoring - an optional setting that determines whether the CacheManager should

automatically register the SampledCacheMBean with the system MBean server.

Currently, this monitoring is only useful when using Terracotta clustering and using the

Terracotta Developer Console. With the "autodetect" value, the presence of Terracotta clustering

will be detected and monitoring, via the Developer Console, will be enabled. Other allowed values

are "on" and "off". The default is "autodetect". This setting does not perform any function when

used with JMX monitors.

\* maxBytesLocalHeap - optional setting that constraints the memory usage of the Caches managed by the CacheManager

to use at most the specified number of bytes of the local VM's heap.

\* maxBytesLocalOffHeap - optional setting that constraints the offHeap usage of the Caches managed by the CacheManager

to use at most the specified number of bytes of the local VM's offHeap memory.

\* maxBytesLocalDisk - optional setting that constraints the disk usage of the Caches managed by the CacheManager

to use at most the specified number of bytes of the local disk.

These settings let you define "resource pools", caches will share. For instance setting maxBytesLocalHeap to 100M, will result in

all caches sharing 100 MegaBytes of ram. The CacheManager will balance these 100 MB across all caches based on their respective usage

patterns. You can allocate a precise amount of bytes to a particular cache by setting the appropriate maxBytes\* attribute for that cache.

That amount will be subtracted from the CacheManager pools, so that if a cache a specified 30M requirement, the other caches will share

the remaining 70M.

Also, specifying a maxBytesLocalOffHeap at the CacheManager level will result in overflowToOffHeap to be true by default. If you don't want

a specific cache to overflow to off heap, you'll have to set overflowToOffHeap="false" explicitly

Here is an example of CacheManager level resource tuning, which will use up to 400M of heap and 2G of offHeap:

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="ehcache.xsd"

updateCheck="true" monitoring="autodetect"

dynamicConfig="true" maxBytesLocalHeap="400M" maxBytesLocalOffHeap="2G">

-->

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="ehcache.xsd"

updateCheck="true" monitoring="autodetect"

dynamicConfig="true">

<!--

Management Rest Service configuration

=====================================

The managementRESTService element is optional. By default the REST service that exposes monitoring and

management features for the caches within the cache manager is disabled. Enabling this feature will

affect cache performance.

The 'bind' attribute defaults to "0.0.0.0:9888" and sets the IP Address and Port to bind the web service

to. "0.0.0.0" binds to all local addresses / network interfaces.

If you provide the 'securityServiceLocation' attribute, this will also enable authentication and other

security measures on the REST service - which are only available for the enterprise-edition of the

service. The location should be the URL to the Terracotta Management Server that is being used to

manage the ehcache instance. Enabling security requires that the management REST service be provided with

a terracotta keychain in the default location ${user.home}/.tc/mgmt/keychain or as defined by the system property

com.tc.management.keychain.file. The keychain is expected to hold a secret shared by the management client

and keyed with this REST service's URI.

Related to the the enterprise-edition security setup is the 'securityServiceTimeout' attribute. Setting this

value will allow adjustment of the connection timeout to the security service location. The default value is

5000 millis.

If the 'sslEnabled' attribute is set to true, this will enable a non-blocking ssl connection to the management

REST service. Turning this ssl connection on requires an identity store be provided at the default location

${user.home}/.tc/mgmt/keystore and that the JKS passphrase be included in the REST service keychain, keyed with

the identity store file URI, or that the keystore and passphrase be identified with the ssl system properties

javax.net.ssl.keyStore and javax.net.ssl.keyStorePassword.

The 'needClientAuth' attribute requires ssl client certificate authorization if the 'sslEnabled' attribute has been

set to true. Otherwise, it will be ignored. Setting this attribute to true will require that the client's

identity is imported as trusted into a truststore which is provided in the default location

${user.home}/.tc/mgmt/keystore and that the JKS passphrase be included in the REST service keychain, keyed with

the trust store file URI, or that the truststore and passphrase be identified with the ssl system properties

javax.net.ssl.trustStore and javax.net.ssl.trustStorePassword.

Finally, several attributes exist to configure sampling history.

- 'sampleHistorySize' allows the configuration of how many statistical samples will be kept in memory for

each cache. The default value is set to 30.

- 'sampleIntervalSeconds' allows the configuration of how often cache statistics will be obtained in seconds.

The default value is set to 1 second.

- 'sampleSearchIntervalSeconds' allows the configuration of how often cache seach statistics will be obtained in

seconds. The default value is set to 10 seconds.

examples:

<managementRESTService enabled="true" bind="0.0.0.0:9888" />

<managementRESTService enabled="true" securityServiceLocation="http://localhost:9889/tmc/api/assertIdentity" />

-->

<!--

DiskStore configuration

=======================

The diskStore element is optional. To turn off disk store path creation, comment out the diskStore

element below.

Configure it if you have disk persistence enabled for any cache or if you use

unclustered indexed search.

If it is not configured, and a cache is created which requires a disk store, a warning will be

issued and java.io.tmpdir will automatically be used.

diskStore has only one attribute - "path". It is the path to the directory where

any required disk files will be created.

If the path is one of the following Java System Property it is replaced by its value in the

running VM. For backward compatibility these should be specified without being enclosed in the ${token}

replacement syntax.

The following properties are translated:

\* user.home - User's home directory

\* user.dir - User's current working directory

\* java.io.tmpdir - Default temp file path

\* ehcache.disk.store.dir - A system property you would normally specify on the command line

e.g. java -Dehcache.disk.store.dir=/u01/myapp/diskdir ...

Subdirectories can be specified below the property e.g. java.io.tmpdir/one

-->

<diskStore path="java.io.tmpdir"/>

<!--

TransactionManagerLookup configuration

======================================

This class is used by ehcache to lookup the JTA TransactionManager use in the application

using an XA enabled ehcache. If no class is specified then DefaultTransactionManagerLookup

will find the TransactionManager in the following order

\*GenericJNDI (i.e. jboss, where the property jndiName controls the name of the

TransactionManager object to look up)

\*Bitronix

\*Atomikos

You can provide you own lookup class that implements the

net.sf.ehcache.transaction.manager.TransactionManagerLookup interface.

-->

<transactionManagerLookup class="net.sf.ehcache.transaction.manager.DefaultTransactionManagerLookup"

properties="jndiName=java:/TransactionManager" propertySeparator=";"/>

<!--

CacheManagerEventListener

=========================

Specifies a CacheManagerEventListenerFactory which is notified when Caches are added

or removed from the CacheManager.

The attributes of CacheManagerEventListenerFactory are:

\* class - a fully qualified factory class name

\* properties - comma separated properties having meaning only to the factory.

Sets the fully qualified class name to be registered as the CacheManager event listener.

The events include:

\* adding a Cache

\* removing a Cache

Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility

of the implementer to safely handle the potential performance and thread safety issues

depending on what their listener is doing.

If no class is specified, no listener is created. There is no default.

-->

<cacheManagerEventListenerFactory class="" properties=""/>

<!--

CacheManagerPeerProvider

========================

(For distributed operation)

Specifies a CacheManagerPeerProviderFactory which will be used to create a

CacheManagerPeerProvider, which discovers other CacheManagers in the cluster.

One or more providers can be configured. The first one in the ehcache.xml is the default, which is used

for replication and bootstrapping.

The attributes of cacheManagerPeerProviderFactory are:

\* class - a fully qualified factory class name

\* properties - comma separated properties having meaning only to the factory.

Providers are available for RMI, JGroups and JMS as shown following.

RMICacheManagerPeerProvider

+++++++++++++++++++++++++++

Ehcache comes with a built-in RMI-based distribution system with two means of discovery of

CacheManager peers participating in the cluster:

\* automatic, using a multicast group. This one automatically discovers peers and detects

changes such as peers entering and leaving the group

\* manual, using manual rmiURL configuration. A hardcoded list of peers is provided at

configuration time.

Configuring Automatic Discovery:

Automatic discovery is configured as per the following example:

<cacheManagerPeerProviderFactory

class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="hostName=fully\_qualified\_hostname\_or\_ip,

peerDiscovery=automatic, multicastGroupAddress=230.0.0.1,

multicastGroupPort=4446, timeToLive=32"/>

Valid properties are:

\* peerDiscovery (mandatory) - specify "automatic"

\* multicastGroupAddress (mandatory) - specify a valid multicast group address

\* multicastGroupPort (mandatory) - specify a dedicated port for the multicast heartbeat

traffic

\* timeToLive - specify a value between 0 and 255 which determines how far the packets will

propagate.

By convention, the restrictions are:

0 - the same host

1 - the same subnet

32 - the same site

64 - the same region

128 - the same continent

255 - unrestricted

\* hostName - the hostname or IP of the interface to be used for sending and receiving multicast

packets (relevant to multi-homed hosts only)

Configuring Manual Discovery:

Manual discovery requires a unique configuration per host. It is contains a list of rmiURLs for

the peers, other than itself. So, if we have server1, server2 and server3 the configuration will

be:

In server1's configuration:

<cacheManagerPeerProviderFactory class=

"net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=manual,

rmiUrls=//server2:40000/sampleCache1|//server3:40000/sampleCache1

| //server2:40000/sampleCache2|//server3:40000/sampleCache2"

propertySeparator="," />

In server2's configuration:

<cacheManagerPeerProviderFactory class=

"net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=manual,

rmiUrls=//server1:40000/sampleCache1|//server3:40000/sampleCache1

| //server1:40000/sampleCache2|//server3:40000/sampleCache2"

propertySeparator="," />

In server3's configuration:

<cacheManagerPeerProviderFactory class=

"net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=manual,

rmiUrls=//server1:40000/sampleCache1|//server2:40000/sampleCache1

| //server1:40000/sampleCache2|//server2:40000/sampleCache2"

propertySeparator="," />

Valid properties are:

\* peerDiscovery (mandatory) - specify "manual"

\* rmiUrls (mandatory) - specify a pipe separated list of rmiUrls, in the form

//hostname:port

\* hostname (optional) - the hostname is the hostname of the remote CacheManager peer. The port is the listening

port of the RMICacheManagerPeerListener of the remote CacheManager peer.

JGroupsCacheManagerPeerProvider

+++++++++++++++++++++++++++++++

<cacheManagerPeerProviderFactory

class="net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPeerProviderFactory"

properties="channel=ehcache^connect=UDP(mcast\_addr=231.12.21.132;mcast\_port=45566;ip\_ttl=32;

mcast\_send\_buf\_size=150000;mcast\_recv\_buf\_size=80000):

PING(timeout=2000;num\_initial\_members=6):

MERGE2(min\_interval=5000;max\_interval=10000):

FD\_SOCK:VERIFY\_SUSPECT(timeout=1500):

pbcast.NAKACK(gc\_lag=10;retransmit\_timeout=3000):

UNICAST(timeout=5000):

pbcast.STABLE(desired\_avg\_gossip=20000):

FRAG:

pbcast.GMS(join\_timeout=5000;join\_retry\_timeout=2000;shun=false;print\_local\_addr=false)"

propertySeparator="^"

/>

JGroups configuration is done by providing a connect string using connect= as in the above example which uses

multicast, or since version 1.4, a file= to specify the location of a JGroups configuration file.

If neither a connect or file property is specified, the default JGroups JChannel will be used.

Multiple JGroups clusters may be run on the same network by specifying a different CacheManager name. The name

is used as the cluster name.

Since version 1.4 you can specify a channelName to avoid conflicts.

JMSCacheManagerPeerProviderFactory

++++++++++++++++++++++++++++++++++

<cacheManagerPeerProviderFactory

class="net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory"

properties="..."

propertySeparator=","

/>

The JMS PeerProviderFactory uses JNDI to maintain message queue independence. Refer to the manual for full configuration

examples using ActiveMQ and Open Message Queue.

Valid properties are:

\* initialContextFactoryName (mandatory) - the name of the factory used to create the message queue initial context.

\* providerURL (mandatory) - the JNDI configuration information for the service provider to use.

\* topicConnectionFactoryBindingName (mandatory) - the JNDI binding name for the TopicConnectionFactory

\* topicBindingName (mandatory) - the JNDI binding name for the topic name

\* getQueueBindingName (mandatory only if using jmsCacheLoader) - the JNDI binding name for the queue name

\* securityPrincipalName - the JNDI java.naming.security.principal

\* securityCredentials - the JNDI java.naming.security.credentials

\* urlPkgPrefixes - the JNDI java.naming.factory.url.pkgs

\* userName - the user name to use when creating the TopicConnection to the Message Queue

\* password - the password to use when creating the TopicConnection to the Message Queue

\* acknowledgementMode - the JMS Acknowledgement mode for both publisher and subscriber. The available choices are

AUTO\_ACKNOWLEDGE, DUPS\_OK\_ACKNOWLEDGE and SESSION\_TRANSACTED. The default is AUTO\_ACKNOWLEDGE.

-->

<cacheManagerPeerProviderFactory

class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=automatic,

multicastGroupAddress=230.0.0.1,

multicastGroupPort=4446, timeToLive=1"

propertySeparator=","

/>

<!--

CacheManagerPeerListener

========================

(Enable for distributed operation)

Specifies a CacheManagerPeerListenerFactory which will be used to create a

CacheManagerPeerListener, which listens for messages from cache replicators participating in the cluster.

The attributes of cacheManagerPeerListenerFactory are:

class - a fully qualified factory class name

properties - comma separated properties having meaning only to the factory.

Ehcache comes with a built-in RMI-based distribution system. The listener component is

RMICacheManagerPeerListener which is configured using

RMICacheManagerPeerListenerFactory. It is configured as per the following example:

<cacheManagerPeerListenerFactory

class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"

properties="hostName=fully\_qualified\_hostname\_or\_ip,

port=40001,

remoteObjectPort=40002,

socketTimeoutMillis=120000"

propertySeparator="," />

All properties are optional. They are:

\* hostName - the hostName of the host the listener is running on. Specify

where the host is multihomed and you want to control the interface over which cluster

messages are received. Defaults to the host name of the default interface if not

specified.

\* port - the port the RMI Registry listener listens on. This defaults to a free port if not specified.

\* remoteObjectPort - the port number on which the remote objects bound in the registry receive calls.

This defaults to a free port if not specified.

\* socketTimeoutMillis - the number of ms client sockets will stay open when sending

messages to the listener. This should be long enough for the slowest message.

If not specified it defaults to 120000ms.

-->

<cacheManagerPeerListenerFactory

class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"/>

<!--

TerracottaConfig

========================

(Enable for Terracotta clustered operation)

Note: You need to install and run one or more Terracotta servers to use Terracotta clustering.

See http://www.terracotta.org/web/display/orgsite/Download.

Specifies a TerracottaConfig which will be used to configure the Terracotta

runtime for this CacheManager.

Configuration can be specified in two main ways: by reference to a source of

configuration or by use of an embedded Terracotta configuration file.

To specify a reference to a source (or sources) of configuration, use the url

attribute. The url attribute must contain a comma-separated list of:

\* path to Terracotta configuration file (usually named tc-config.xml)

\* URL to Terracotta configuration file

\* <server host>:<port> of running Terracotta Server instance

Simplest example for pointing to a Terracotta server on this machine:

<terracottaConfig url="localhost:9510"/>

This element has one attribute "rejoin" which can take values of either "true" or "false":

<terracottaConfig rejoin="true" url="localhost:9510" />

By default, this attribute is false.

Without rejoin, if the Terracotta Server is restarted the client cannot connect back to the

server. When enabled, this allows the client to connect to the new cluster without the

need to restart the node.

Example using a path to Terracotta configuration file:

<terracottaConfig url="/app/config/tc-config.xml"/>

Example using a URL to a Terracotta configuration file:

<terracottaConfig url="http://internal/ehcache/app/tc-config.xml"/>

Example using multiple Terracotta server instance URLs (for fault tolerance):

<terracottaConfig url="host1:9510,host2:9510,host3:9510"/>

To embed a Terracotta configuration file within the ehcache configuration, simply

place a normal Terracotta XML config within the <terracottaConfig> element.

Example:

<terracottaConfig>

<tc-config>

<servers>

<server host="server1" name="s1"/>

<server host="server2" name="s2"/>

</servers>

<clients>

<logs>app/logs-%i</logs>

</clients>

</tc-config>

</terracottaConfig>

For more information on the Terracotta configuration, see the Terracotta documentation.

-->

<!--

Cache configuration

===================

The following attributes are required.

name:

Sets the name of the cache. This is used to identify the cache. It must be unique.

There is a restriction on characters in the name in case MBeans are used.

See the section describing restrictions for unquoted values in the javax.management.ObjectName javadoc.

The following attributes and elements are optional.

maxEntriesLocalHeap:

Sets the maximum number of objects that will be held on heap memory. 0 = no limit.

maxEntriesLocalDisk:

Sets the maximum number of objects that will be maintained in the DiskStore

The default value is zero, meaning unlimited.

eternal:

Sets whether elements are eternal. If eternal, timeouts are ignored and the

element is never expired.

maxEntriesInCache:

This feature is applicable only to Terracotta distributed caches.

Sets the maximum number of entries that can be stored in the cluster. 0 = no limit.

Note that clustered cache will still perform eviction if resource usage requires it.

This property can be modified dynamically while the cache is operating.

overflowToOffHeap:

(boolean) This feature is available only in enterprise versions of Ehcache.

When set to true, enables the cache to utilize off-heap memory

storage to improve performance. Off-heap memory is not subject to Java

GC. The default value is false.

maxBytesLocalHeap:

Defines how many bytes the cache may use from the VM's heap. If a CacheManager

maxBytesLocalHeap has been defined, this Cache's specified amount will be

subtracted from the CacheManager. Other caches will share the remainder.

This attribute's values are given as <number>k|K|m|M|g|G for

kilobytes (k|K), megabytes (m|M), or gigabytes (g|G).

For example, maxBytesLocalHeap="2g" allots 2 gigabytes of heap memory.

If you specify a maxBytesLocalHeap, you can't use the maxEntriesLocalHeap attribute.

maxEntriesLocalHeap can't be used if a CacheManager maxBytesLocalHeap is set.

Elements put into the cache will be measured in size using net.sf.ehcache.pool.sizeof.SizeOf

If you wish to ignore some part of the object graph, see net.sf.ehcache.pool.sizeof.annotations.IgnoreSizeOf

maxBytesLocalOffHeap:

This feature is available only in enterprise versions of Ehcache.

Sets the amount of off-heap memory this cache can use, and will reserve.

This setting will set overflowToOffHeap to true. Set explicitly to false to disable overflow behavior.

Note that it is recommended to set maxEntriesLocalHeap to at least 100 elements

when using an off-heap store, otherwise performance will be seriously degraded,

and a warning will be logged.

The minimum amount that can be allocated is 128MB. There is no maximum.

maxBytesLocalDisk:

As for maxBytesLocalHeap, but specifies the limit of disk storage this cache will ever use.

timeToIdleSeconds:

Sets the time to idle for an element before it expires.

i.e. The maximum amount of time between accesses before an element expires

Is only used if the element is not eternal.

Optional attribute. A value of 0 means that an Element can idle for infinity.

The default value is 0.

timeToLiveSeconds:

Sets the time to live for an element before it expires.

i.e. The maximum time between creation time and when an element expires.

Is only used if the element is not eternal.

Optional attribute. A value of 0 means that and Element can live for infinity.

The default value is 0.

diskExpiryThreadIntervalSeconds:

The number of seconds between runs of the disk expiry thread. The default value

is 120 seconds.

diskSpoolBufferSizeMB:

This is the size to allocate the DiskStore for a spool buffer. Writes are made

to this area and then asynchronously written to disk. The default size is 30MB.

Each spool buffer is used only by its cache. If you get OutOfMemory errors consider

lowering this value. To improve DiskStore performance consider increasing it. Trace level

logging in the DiskStore will show if put back ups are occurring.

clearOnFlush:

whether the MemoryStore should be cleared when flush() is called on the cache.

By default, this is true i.e. the MemoryStore is cleared.

memoryStoreEvictionPolicy:

Policy would be enforced upon reaching the maxEntriesLocalHeap limit. Default

policy is Least Recently Used (specified as LRU). Other policies available -

First In First Out (specified as FIFO) and Less Frequently Used

(specified as LFU)

copyOnRead:

Whether an Element is copied when being read from a cache.

By default this is false.

copyOnWrite:

Whether an Element is copied when being added to the cache.

By default this is false.

Cache persistence is configured through the persistence sub-element. The attributes of the

persistence element are:

strategy:

Configures the type of persistence provided by the configured cache. This must be one of the

following values:

\* localRestartable - Enables the RestartStore and copies all cache entries (on-heap and/or off-heap)

to disk. This option provides fast restartability with fault tolerant cache persistence on disk.

It is available for Enterprise Ehcache users only.

\* localTempSwap - Swaps cache entries (on-heap and/or off-heap) to disk when the cache is full.

"localTempSwap" is not persistent.

\* none - Does not persist cache entries.

\* distributed - Defers to the <terracotta> configuration for persistence settings. This option

is not applicable for standalone.

synchronousWrites:

When set to true write operations on the cache do not return until after the operations data has been

successfully flushed to the disk storage. This option is only valid when used with the "localRestartable"

strategy, and defaults to false.

The following example configuration shows a cache configured for localTempSwap restartability.

<cache name="persistentCache" maxEntriesLocalHeap="1000">

<persistence strategy="localTempSwap"/>

</cache>

Cache elements can also contain sub elements which take the same format of a factory class

and properties. Defined sub-elements are:

\* cacheEventListenerFactory - Enables registration of listeners for cache events, such as

put, remove, update, and expire.

\* bootstrapCacheLoaderFactory - Specifies a BootstrapCacheLoader, which is called by a

cache on initialisation to prepopulate itself.

\* cacheExtensionFactory - Specifies a CacheExtension, a generic mechanism to tie a class

which holds a reference to a cache to the cache lifecycle.

\* cacheExceptionHandlerFactory - Specifies a CacheExceptionHandler, which is called when

cache exceptions occur.

\* cacheLoaderFactory - Specifies a CacheLoader, which can be used both asynchronously and

synchronously to load objects into a cache. More than one cacheLoaderFactory element

can be added, in which case the loaders form a chain which are executed in order. If a

loader returns null, the next in chain is called.

\* copyStrategy - Specifies a fully qualified class which implements

net.sf.ehcache.store.compound.CopyStrategy. This strategy will be used for copyOnRead

and copyOnWrite in place of the default which is serialization.

Example of cache level resource tuning:

<cache name="memBound" maxBytesLocalHeap="100m" maxBytesLocalOffHeap="4g" maxBytesLocalDisk="200g" />

Cache Event Listeners

+++++++++++++++++++++

All cacheEventListenerFactory elements can take an optional property listenFor that describes

which events will be delivered in a clustered environment. The listenFor attribute has the

following allowed values:

\* all - the default is to deliver all local and remote events

\* local - deliver only events originating in the current node

\* remote - deliver only events originating in other nodes

Example of setting up a logging listener for local cache events:

<cacheEventListenerFactory class="my.company.log.CacheLogger"

listenFor="local" />

Search

++++++

A <cache> can be made searchable by adding a <searchable/> sub-element. By default the keys

and value objects of elements put into the cache will be attributes against which

queries can be expressed.

<cache>

<searchable/>

</cache>

An "attribute" of the cache elements can also be defined to be searchable. In the example below

an attribute with the name "age" will be available for use in queries. The value for the "age"

attribute will be computed by calling the method "getAge()" on the value object of each element

in the cache. See net.sf.ehcache.search.attribute.ReflectionAttributeExtractor for the format of

attribute expressions. Attribute values must also conform to the set of types documented in the

net.sf.ehcache.search.attribute.AttributeExtractor interface

<cache>

<searchable>

<searchAttribute name="age" expression="value.getAge()"/>

</searchable>

</cache>

Attributes may also be defined using a JavaBean style. With the following attribute declaration

a public method getAge() will be expected to be found on either the key or value for cache elements

<cache>

<searchable>

<searchAttribute name="age"/>

</searchable>

</cache>

In more complex situations you can create your own attribute extractor by implementing the

AttributeExtractor interface. Providing your extractor class is shown in the following example:

<cache>

<searchable>

<searchAttribute name="age" class="com.example.MyAttributeExtractor"/>

</searchable>

</cache>

Use properties to pass state to your attribute extractor if needed. Your implementation must provide

a public constructor that takes a single java.util.Properties instance

<cache>

<searchable>

<searchAttribute name="age" class="com.example.MyAttributeExtractor" properties="foo=1,bar=2"/>

</searchable>

</cache>

Attributes may also be defined with an optional type constraint on their values. The type specified

must be one of the supported types, or resolve to an enum. It is possible to use either a fully

qualified name of the class that represents the type, or its shortened version when the type is not

an enum. The type names are case sensitive, i.e. "double" is distinct from "Double".

<cache>

<searchable>

<searchAttribute name="address" type="java.lang.String" expression="value.address.toString()"/>

<searchAttribute name="income" type="Long"/>

<searchAttribute name="age" type="int"/>

<searchAttribute name="gender" expression="value.gender" type="com.example.Gender"/>

<searchable>

</cache>

If you intend to use dynamic attribute extraction (see net.sf.ehcache.Cache.registerDynamicAttributesExtractor) then

you need to enable it as follows:

<cache>

<searchable allowDynamicIndexing="true"/>

</cache>

RMI Cache Replication

+++++++++++++++++++++

Each cache that will be distributed needs to set a cache event listener which replicates

messages to the other CacheManager peers. For the built-in RMI implementation this is done

by adding a cacheEventListenerFactory element of type RMICacheReplicatorFactory to each

distributed cache's configuration as per the following example:

<cacheEventListenerFactory class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"

properties="replicateAsynchronously=true,

replicatePuts=true,

replicatePutsViaCopy=false,

replicateUpdates=true,

replicateUpdatesViaCopy=true,

replicateRemovals=true,

asynchronousReplicationIntervalMillis=<number of milliseconds>,

asynchronousReplicationMaximumBatchSize=<number of operations>"

propertySeparator="," />

The RMICacheReplicatorFactory recognises the following properties:

\* replicatePuts=true|false - whether new elements placed in a cache are

replicated to others. Defaults to true.

\* replicatePutsViaCopy=true|false - whether the new elements are

copied to other caches (true), or whether a remove message is sent. Defaults to true.

\* replicateUpdates=true|false - whether new elements which override an

element already existing with the same key are replicated. Defaults to true.

\* replicateRemovals=true - whether element removals are replicated. Defaults to true.

\* replicateAsynchronously=true | false - whether replications are

asynchronous (true) or synchronous (false). Defaults to true.

\* replicateUpdatesViaCopy=true | false - whether the new elements are

copied to other caches (true), or whether a remove message is sent. Defaults to true.

\* asynchronousReplicationIntervalMillis=<number of milliseconds> - The asynchronous

replicator runs at a set interval of milliseconds. The default is 1000. The minimum

is 10. This property is only applicable if replicateAsynchronously=true

\* asynchronousReplicationMaximumBatchSize=<number of operations> - The maximum

number of operations that will be batch within a single RMI message. The default

is 1000. This property is only applicable if replicateAsynchronously=true

JGroups Replication

+++++++++++++++++++

For the Jgroups replication this is done with:

<cacheEventListenerFactory class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory"

properties="replicateAsynchronously=true, replicatePuts=true,

replicateUpdates=true, replicateUpdatesViaCopy=false,

replicateRemovals=true,asynchronousReplicationIntervalMillis=1000"/>

This listener supports the same properties as the RMICacheReplicationFactory.

JMS Replication

+++++++++++++++

For JMS-based replication this is done with:

<cacheEventListenerFactory

class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"

properties="replicateAsynchronously=true,

replicatePuts=true,

replicateUpdates=true,

replicateUpdatesViaCopy=true,

replicateRemovals=true,

asynchronousReplicationIntervalMillis=1000"

propertySeparator=","/>

This listener supports the same properties as the RMICacheReplicationFactory.

Cluster Bootstrapping

+++++++++++++++++++++

Bootstrapping a cluster may use a different mechanism to replication. e.g you can mix

JMS replication with bootstrap via RMI - just make sure you have the cacheManagerPeerProviderFactory

and cacheManagerPeerListenerFactory configured.

There are two bootstrapping mechanisms: RMI and JGroups.

RMI Bootstrap

The RMIBootstrapCacheLoader bootstraps caches in clusters where RMICacheReplicators are

used. It is configured as per the following example:

<bootstrapCacheLoaderFactory

class="net.sf.ehcache.distribution.RMIBootstrapCacheLoaderFactory"

properties="bootstrapAsynchronously=true, maximumChunkSizeBytes=5000000"

propertySeparator="," />

The RMIBootstrapCacheLoaderFactory recognises the following optional properties:

\* bootstrapAsynchronously=true|false - whether the bootstrap happens in the background

after the cache has started. If false, bootstrapping must complete before the cache is

made available. The default value is true.

\* maximumChunkSizeBytes=<integer> - Caches can potentially be very large, larger than the

memory limits of the VM. This property allows the bootstraper to fetched elements in

chunks. The default chunk size is 5000000 (5MB).

JGroups Bootstrap

Here is an example of bootstrap configuration using JGroups boostrap:

<bootstrapCacheLoaderFactory class="net.sf.ehcache.distribution.jgroups.JGroupsBootstrapCacheLoaderFactory"

properties="bootstrapAsynchronously=true"/>

The configuration properties are the same as for RMI above. Note that JGroups bootstrap only supports

asynchronous bootstrap mode.

Cache Exception Handling

++++++++++++++++++++++++

By default, most cache operations will propagate a runtime CacheException on failure. An

interceptor, using a dynamic proxy, may be configured so that a CacheExceptionHandler can

be configured to intercept Exceptions. Errors are not intercepted.

It is configured as per the following example:

<cacheExceptionHandlerFactory class="com.example.ExampleExceptionHandlerFactory"

properties="logLevel=FINE"/>

Caches with ExceptionHandling configured are not of type Cache, but are of type Ehcache only,

and are not available using CacheManager.getCache(), but using CacheManager.getEhcache().

Cache Loader

++++++++++++

A default CacheLoader may be set which loads objects into the cache through asynchronous and

synchronous methods on Cache. This is different to the bootstrap cache loader, which is used

only in distributed caching.

It is configured as per the following example:

<cacheLoaderFactory class="com.example.ExampleCacheLoaderFactory"

properties="type=int,startCounter=10"/>

Element value comparator

++++++++++++++++++++++++

These two cache atomic methods:

removeElement(Element e)

replace(Element old, Element element)

rely on comparison of cached elements value. The default implementation relies on Object.equals()

but that can be changed in case you want to use a different way to compute equality of two elements.

This is configured as per the following example:

<elementValueComparator class="com.company.xyz.MyElementComparator"/>

The MyElementComparator class must implement the is net.sf.ehcache.store.ElementValueComparator

interface. The default implementation is net.sf.ehcache.store.DefaultElementValueComparator.

SizeOf Policy

+++++++++++++

Control how deep the SizeOf engine can go when sizing on-heap elements.

This is configured as per the following example:

<sizeOfPolicy maxDepth="100" maxDepthExceededBehavior="abort"/>

maxDepth controls how many linked objects can be visited before the SizeOf engine takes any action.

maxDepthExceededBehavior specifies what happens when the max depth is exceeded while sizing an object graph.

"continue" makes the SizeOf engine log a warning and continue the sizing. This is the default.

"abort" makes the SizeOf engine abort the sizing, log a warning and mark the cache as not correctly tracking

memory usage. This makes Ehcache.hasAbortedSizeOf() return true when this happens.

The SizeOf policy can be configured at the cache manager level (directly under <ehcache>) and at

the cache level (under <cache> or <defaultCache>). The cache policy always overrides the cache manager

one if both are set. This element has no effect on distributed caches.

Transactions

++++++++++++

To enable an ehcache as transactions, set the transactionalMode

transactionalMode="xa" - high performance JTA/XA implementation

transactionalMode="xa\_strict" - canonically correct JTA/XA implementation

transactionMode="local" - high performance local transactions involving caches only

transactionalMode="off" - the default, no transactions

If set, all cache operations will need to be done through transactions.

To prevent users keeping references on stored elements and modifying them outside of any transaction's control,

transactions also require the cache to be configured copyOnRead and copyOnWrite.

CacheWriter

++++++++++++

A CacheWriter can be set to write to an underlying resource. Only one CacheWriter can be

configured per cache.

The following is an example of how to configure CacheWriter for write-through:

<cacheWriter writeMode="write-through" notifyListenersOnException="true">

<cacheWriterFactory class="net.sf.ehcache.writer.TestCacheWriterFactory"

properties="type=int,startCounter=10"/>

</cacheWriter>

The following is an example of how to configure CacheWriter for write-behind:

<cacheWriter writeMode="write-behind" minWriteDelay="1" maxWriteDelay="5"

rateLimitPerSecond="5" writeCoalescing="true" writeBatching="true" writeBatchSize="1"

retryAttempts="2" retryAttemptDelaySeconds="1">

<cacheWriterFactory class="net.sf.ehcache.writer.TestCacheWriterFactory"

properties="type=int,startCounter=10"/>

</cacheWriter>

The cacheWriter element has the following attributes:

\* writeMode: the write mode, write-through or write-behind

These attributes only apply to write-through mode:

\* notifyListenersOnException: Sets whether to notify listeners when an exception occurs on a writer operation.

These attributes only apply to write-behind mode:

\* minWriteDelay: Set the minimum number of seconds to wait before writing behind. If set to a value greater than 0,

it permits operations to build up in the queue. This is different from the maximum write delay in that by waiting

a minimum amount of time, work is always being built up. If the minimum write delay is set to zero and the

CacheWriter performs its work very quickly, the overhead of processing the write behind queue items becomes very

noticeable in a cluster since all the operations might be done for individual items instead of for a collection

of them.

\* maxWriteDelay: Set the maximum number of seconds to wait before writing behind. If set to a value greater than 0,

it permits operations to build up in the queue to enable effective coalescing and batching optimisations.

\* writeBatching: Sets whether to batch write operations. If set to true, writeAll and deleteAll will be called on

the CacheWriter rather than write and delete being called for each key. Resources such as databases can perform

more efficiently if updates are batched, thus reducing load.

\* writeBatchSize: Sets the number of operations to include in each batch when writeBatching is enabled. If there are

less entries in the write-behind queue than the batch size, the queue length size is used.

\* rateLimitPerSecond: Sets the maximum number of write operations to allow per second when writeBatching is enabled.

\* writeCoalescing: Sets whether to use write coalescing. If set to true and multiple operations on the same key are

present in the write-behind queue, only the latest write is done, as the others are redundant.

\* retryAttempts: Sets the number of times the operation is retried in the CacheWriter, this happens after the

original operation.

\* retryAttemptDelaySeconds: Sets the number of seconds to wait before retrying an failed operation.

Pinning

+++++++

Use this element when data should remain in the cache regardless of resource constraints.

Unexpired entries can never be flushed to a lower tier or be evicted.

This element has a required attribute (store) to specify which data tiers the cache should be pinned to:

\* localMemory: Cache data is pinned to the local heap (or off-heap for BigMemory Go and BigMemory Max).

\* inCache: Cache data is pinned in the cache, which can be in any tier cache data is stored.

Example:

<pinning store="inCache"/>

Cache Extension

+++++++++++++++

CacheExtensions are a general purpose mechanism to allow generic extensions to a Cache.

CacheExtensions are tied into the Cache lifecycle.

CacheExtensions are created using the CacheExtensionFactory which has a

<code>createCacheCacheExtension()</code> method which takes as a parameter a

Cache and properties. It can thus call back into any public method on Cache, including, of

course, the load methods.

Extensions are added as per the following example:

<cacheExtensionFactory class="com.example.FileWatchingCacheRefresherExtensionFactory"

properties="refreshIntervalMillis=18000, loaderTimeout=3000,

flushPeriod=whatever, someOtherProperty=someValue ..."/>

Cache Decorator Factory

+++++++++++++++++++++++

Cache decorators can be configured directly in ehcache.xml. The decorators will be created and added to the CacheManager.

It accepts the name of a concrete class that extends net.sf.ehcache.constructs.CacheDecoratorFactory

The properties will be parsed according to the delimiter (default is comma ',') and passed to the concrete factory's

<code>createDecoratedEhcache(Ehcache cache, Properties properties)</code> method along with the reference to the owning cache.

It is configured as per the following example:

<cacheDecoratorFactory

class="com.company.DecoratedCacheFactory"

properties="property1=true ..." />

Distributed Caching with Terracotta

+++++++++++++++++++++++++++++++++++

Distributed Caches connect to a Terracotta Server Array. They are configured with the <terracotta> sub-element.

The <terracotta> sub-element has the following attributes:

\* clustered=true|false - indicates whether this cache should be clustered (distributed) with Terracotta. By

default, if the <terracotta> element is included, clustered=true.

\* copyOnRead=true|false - indicates whether cache values are deserialized on every read or if the

materialized cache value can be re-used between get() calls. This setting is useful if a cache

is being shared by callers with disparate classloaders or to prevent local drift if keys/values

are mutated locally without being put back in the cache.

The default is false.

\* consistency=strong|eventual - Indicates whether this cache should have strong consistency or eventual

consistency. The default is eventual. See the documentation for the meaning of these terms.

\* synchronousWrites=true|false

Synchronous writes (synchronousWrites="true") maximize data safety by blocking the client thread until

the write has been written to the Terracotta Server Array.

This option is only available with consistency=strong. The default is false.

\* concurrency - the number of segments that will be used by the map underneath the Terracotta Store.

Its optional and has default value of 0, which means will use default values based on the internal

Map being used underneath the store.

This value cannot be changed programmatically once a cache is initialized.

The <terracotta> sub-element also has a <nonstop> sub-element to allow configuration of cache behaviour if a distributed

cache operation cannot be completed within a set time or in the event of a clusterOffline message. If this element does not appear, nonstop behavior is off.

<nonstop> has the following attributes:

\* enabled="true" - defaults to true.

\* timeoutMillis - An SLA setting, so that if a cache operation takes longer than the allowed ms, it will timeout.

\* searchTimeoutMillis - If a cache search operation in the nonstop mode takes longer than the allowed ms, it will timeout.

\* immediateTimeout="true|false" - What to do on receipt of a ClusterOffline event indicating that communications

with the Terracotta Server Array were interrupted.

<nonstop> has one sub-element, <timeoutBehavior> which has the following attribute:

\* type="noop|exception|localReads|localReadsAndExceptionOnWrite" - What to do when a timeout has occurred. Exception is the default.

Simplest example to indicate clustering:

<terracotta/>

To indicate the cache should not be clustered (or remove the <terracotta> element altogether):

<terracotta clustered="false"/>

To indicate the cache should be clustered using "eventual" consistency mode for better performance :

<terracotta clustered="true" consistency="eventual"/>

To indicate the cache should be clustered using synchronous-write locking level:

<terracotta clustered="true" synchronousWrites="true"/>

-->

<!--

Default Cache configuration. These settings will be applied to caches

created programmatically using CacheManager.add(String cacheName).

This element is optional, and using CacheManager.add(String cacheName) when

its not present will throw CacheException

The defaultCache has an implicit name "default" which is a reserved cache name.

-->

<defaultCache

maxEntriesLocalHeap="10000"

eternal="false"

timeToIdleSeconds="120"

timeToLiveSeconds="120"

diskSpoolBufferSizeMB="30"

maxEntriesLocalDisk="10000000"

diskExpiryThreadIntervalSeconds="120"

memoryStoreEvictionPolicy="LRU">

<persistence strategy="localTempSwap"/>

</defaultCache>

<!--

Sample caches. Following are some example caches. Remove these before use.

-->

<!--

Sample cache named sampleCache1

This cache contains a maximum in memory of 10000 elements, and will expire

an element if it is idle for more than 5 minutes and lives for more than

10 minutes.

If there are more than 10000 elements it will overflow to the

disk cache, which in this configuration will go to wherever java.io.tmp is

defined on your system. On a standard Linux system this will be /tmp"

-->

<cache name="sampleCache1"

maxEntriesLocalHeap="10000"

maxEntriesLocalDisk="1000"

eternal="false"

diskSpoolBufferSizeMB="20"

timeToIdleSeconds="300"

timeToLiveSeconds="600"

memoryStoreEvictionPolicy="LFU"

transactionalMode="off">

<persistence strategy="localTempSwap"/>

</cache>

<!--

Sample cache named sampleCache2

This cache has a maximum of 1000 elements in memory. There is no overflow to disk, so 1000

is also the maximum cache size. Note that when a cache is eternal, timeToLive and

timeToIdle are not used and do not need to be specified.

-->

<cache name="sampleCache2"

maxEntriesLocalHeap="1000"

eternal="true"

memoryStoreEvictionPolicy="FIFO"

/>

<!--

Sample cache named sampleCache3. This cache overflows to disk. The disk store is

persistent between cache and VM restarts. The disk expiry thread interval is set to 10

minutes, overriding the default of 2 minutes.

-->

<cache name="sampleCache3"

maxEntriesLocalHeap="500"

eternal="false"

overflowToDisk="true"

diskPersistent="true"

timeToIdleSeconds="300"

timeToLiveSeconds="600"

diskExpiryThreadIntervalSeconds="1"

memoryStoreEvictionPolicy="LFU">

</cache>

<!--

Sample distributed cache named sampleReplicatedCache1.

This cache replicates using defaults.

It also bootstraps from the cluster, using default properties.

-->

<cache name="sampleReplicatedCache1"

maxEntriesLocalHeap="10"

eternal="false"

timeToIdleSeconds="100"

timeToLiveSeconds="100">

<cacheEventListenerFactory

class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>

<bootstrapCacheLoaderFactory

class="net.sf.ehcache.distribution.RMIBootstrapCacheLoaderFactory"/>

</cache>

<!--

Sample distributed cache named sampleReplicatedCache2.

This cache replicates using specific properties.

It only replicates updates and does so synchronously via copy

-->

<cache name="sampleRepicatedCache2"

maxEntriesLocalHeap="10"

eternal="false"

timeToIdleSeconds="100"

timeToLiveSeconds="100">

<cacheEventListenerFactory

class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"

properties="replicateAsynchronously=false, replicatePuts=false,

replicatePutsViaCopy=false, replicateUpdates=true,

replicateUpdatesViaCopy=true, replicateRemovals=false"/>

</cache>

<!--

Sample distributed cache named sampleReplicatedCache3.

This cache replicates using defaults except that the asynchronous replication

interval is set to 200ms.

This one includes / and # which were illegal in ehcache 1.5.

-->

<cache name="sampleReplicatedCache3"

maxEntriesLocalHeap="10"

eternal="false"

timeToIdleSeconds="100"

timeToLiveSeconds="100">

<cacheEventListenerFactory

class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"

properties="asynchronousReplicationIntervalMillis=200"/>

<persistence strategy="localTempSwap"/>

</cache>

<!--

Sample Terracotta clustered cache named sampleTerracottaCache.

This cache uses Terracotta to cluster the contents of the cache.

-->

<!--

<cache name="sampleTerracottaCache"

maxBytesLocalHeap="10m"

eternal="false"

timeToIdleSeconds="3600"

timeToLiveSeconds="1800">

<terracotta/>

</cache>

-->

<!--

Sample xa enabled cache named xaCache

<cache name="xaCache"

maxEntriesLocalHeap="500"

eternal="false"

timeToIdleSeconds="300"

timeToLiveSeconds="600"

diskExpiryThreadIntervalSeconds="1"

transactionalMode="xa\_strict">

</cache>

-->

<!--

Sample copy on both read and write cache named copyCache

using the default (explicitly configured here as an example) ReadWriteSerializationCopyStrategy

class could be any implementation of net.sf.ehcache.store.compound.CopyStrategy

<cache name="copyCache"

maxEntriesLocalHeap="500"

eternal="false"

timeToIdleSeconds="300"

timeToLiveSeconds="600"

diskExpiryThreadIntervalSeconds="1"

copyOnRead="true"

copyOnWrite="true">

<copyStrategy class="net.sf.ehcache.store.compound.ReadWriteSerializationCopyStrategy" />

</cache>

-->

<!--

Sample, for Enterprise Ehcache only, demonstrating a tiered cache with in-memory, off-heap and disk stores. In this example the in-memory (on-heap) store is limited to 10,000 items ... which for example for 1k items would use 10MB of memory, the off-heap store is limited to 4GB and the disk store is unlimited in size.

<cache name="tieredCache"

maxEntriesLocalHeap="10000"

eternal="false"

timeToLiveSeconds="600"

overflowToOffHeap="true"

maxBytesLocalOffHeap="4g"

diskExpiryThreadIntervalSeconds="1">

<persistence strategy="localTempSwap"/>

</cache>

-->

<!--

Sample, for Enterprise Ehcache only, demonstrating a restartable cache with in-memory and off-heap stores.

<cache name="restartableCache"

maxEntriesLocalHeap="10000"

eternal="true"

overflowToOffHeap="true"

maxBytesLocalOffHeap="4g"

<persistence strategy="localRestartable"/>

</cache>

-->

</ehcache>

[orafmw@ip-192-168-2-92 ehcache-2.10.2]$

The on-heap store refers to objects that will be present in the Java heap (and also subject to GC). On the other hand, the off-heap store refers to (serialized) objects that are managed by EHCache, but stored outside the heap (and also not subject to GC). As the off-heap store continues to be managed in memory, it is slightly slower than the on-heap store, but still faster than the disk store.

The internal details involved in management and usage of the off-heap store aren't very evident in the link posted in the question, so it would be wise to check out the details of [Terrcotta BigMemory](http://www.terracotta.org/bigmemory), which is used to manage the off-disk store. BigMemory (the off-heap store) is to be used to avoid the overhead of GC on a heap that is several Megabytes or Gigabytes large. BigMemory uses the memory address space of the JVM process, via [direct ByteBuffers](http://download.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html) that are not subject to GC unlike other native Java objects.

It requires no special configuration to enable, and

its overall size is taken from the Java heap size. Since it exists in the heap, it is limited by Java GC constraints.

**EHcache.xml Attributes**

If you are working with Ehcache configuration, then it is very important to understand the meaning and usage of each and every element and attribute used in the ehcache.xml file. Here I am explaining the use of some of the useful attributes that are most commonly used in the ehcache configurations.

**defaultCache**

This element is a mandatory default cache configuration. These settings will be applied to caches created programmtically using CacheManager.add(String cacheName). The defaultCache has an implicit name “default” which is a reserved cache name.

**maxElementsInMemory**

The attribute tells the Ehcache that how many caches would be stored in the memory. If the total caches reached the limit, then the caches will be pushed to the disk if the configurations set as overflowToDisk=true. If this attribute set as false, old caches will be evicted and new caches will be replace the old one.

**eternal**

If this attribute is set as true, then other attributes timeToIdleSecondsand timeToLiveSeconds need not be configured. If you configure these attributes, then the configure values will not be considered. The default value of 0 will be taken for both the attributes. Note that, the configuration eternal=true implies that caching configuration should not consider the timeToIdleSeconds and timeToLiveSeconds. It means that the caches will not be expired for ever in the store. It has to be cleared manually by the server administrator by restarting the server. Be careful on using this attribute.

**overflowToDisk**

This attribute sets whether cache elements can overflow to disk when the memory store has reached the maxElementsInMemory limit. If the attribute set to false, cache elements will not overflow to disk when store has reached the maxElementsInMemory limit.

**timeToIdleSeconds**

This attribute sets the idle time before it expires. For example, if you set this attribute as 10, the caches that are not accessed for 10 seconds will be automatically expired. The default value for this attribute is 0. If the value is 0, this attribute will not be considered for the expiration. timeToIdleSeconds is valid only when eternal attribute value is false.

**timeToLiveSeconds**

This attribute sets the total expiry time for the cache. It is the time between creating time and expiration time. The default value for this attribute is 0. If the value is 0, this attribute will not be considered for the expiration. timeToLiveSeconds is valid only when eternal attribute value is false.

**maxElementsOnDisk**

This attribute sets the maximum number of objects that will be maintained in the DiskStore. The default value is zero, meaning unlimited. This attribute has been deprecated from the version 2.5.

**maxEntriesLocalDisk vs maxElementsOnDisk**

maxElementsOnDisk is the historical name, deprecated since 2.5 This attribute is replaced by new attribute maxElementsLocalDisk. But the new attribute controls the exact same internal variable.

**Where the cache files are stored?**

The default disk store path for the cache file is the default temp directory of the operating system. For example, if you are running it on linux, the default path would be /tmp. Please cont’ confuse with the temp directory of the application server. Here is the pre-defined variables to specifying the default directories.

* user.home – User’s home directory
* user.dir – User’s current working directory
* java.io.tmpdir – Default temp file path
* ehcache.disk.store.dir – A system property

If you want to create a sub-directory, it can be specified as e.g. java.io.tmpdir/one