# Ehcache Configuration Guide

Version 2.9

October 2014

EHCACHE

**Document ID: EHC-CG-29-20141015**

# Table of Contents

# 1 Configuring Cache

## About Ehcache Configuration

Ehcache supports declarative configuration via an XML configuration file, as well as programmatic configuration via class-constructor APIs. Choosing one approach over the other can be a matter of preference or a requirement, such as when an application requires a certain run-time context to determine appropriate configuration settings.

If your project permits the separation of configuration from run time use, there are advantages to the declarative approach:

- Cache configuration can be changed more easily at deployment time.
- Configuration can be centrally organized for greater visibility.
- Configuration lifecycle can be separated from application-code lifecycle.
- Configuration errors are checked at startup rather than causing an unexpected runtime error.
- If the configuration file is not provided, a default configuration is always loaded at runtime.

This guide focuses on XML declarative configuration. Programmatic configuration is explored in certain examples and is documented in the Javadoc.

## XML Configuration

By default, Ehcache looks for an ASCII or UTF8 encoded XML configuration file called ehcache.xml at the top level of the Java classpath. You may specify alternate paths and filenames for the XML configuration file by using the various CacheManager constructors as described in the CacheManager Javadoc.

To avoid resource conflicts, one XML configuration is required for each CacheManager that is created. For example, directory paths and listener ports require unique values. Ehcache will attempt to resolve conflicts, and, if one is found, it will emit a warning reminding the user to use separate configurations for multiple CacheManagers.

A sample ehcache.xml file is included in the Ehcache distribution. It contains full commentary on how to configure each element. This file can also be downloaded from http://ehcache.org/ehcache.xml.

**Note:** Prior to ehcache-1.6, Ehcache only supported ASCII ehcache.xml configuration files. Starting with ehcache-1.6, UTF8 is supported, so that configuration can use Unicode. Because UTF8 is backwardly compatible with ASCII, no conversion is necessary.

**Note:** Some elements documented in the ehcache.xml sample file pertain only to the Terracotta BigMemory products and are not valid for the open-source version of Ehcache.

### ehcache.xsd

Ehcache configuration files must comply with the Ehcache XML schema, ehcache.xsd, which can be downloaded from http://ehcache.org/ehcache.xsd.

The Ehcache distribution also contains a copy of ehcache.xsd.

**Note:** Note that some elements documented by the Ehcache XML schema pertain only to the Terracotta BigMemory products and are not valid for the open-source version of Ehcache.

### ehcache-failsafe.xml

If the CacheManager default constructor or factory method is called, Ehcache looks for a file called ehcache.xml in the top level of the classpath. Failing that, it looks for ehcache-failsafe.xml in the classpath. The ehcache-failsafe.xml file is packaged in the Ehcache JAR and should always be found.

ehcache-failsafe.xml provides a simple default configuration to enable users to get started before they create their own ehcache.xml.

When ehcache-failsafe.xml is used, Ehcache will emit a warning, reminding the user to set up a proper configuration. The meaning of the elements and attributes are explained in the section on ehcache.xml.

```
<ehcache>
  <diskStore path="java.io.tmpdir"/>
  <defaultCache
     maxEntriesLocalHeap="10000"
     eternal="false"
     timeToIdleSeconds="120"
     timeToLiveSeconds="120"
     maxEntriesLocalDisk="10000000"
     diskExpiryThreadIntervalSeconds="120"
     memoryStoreEvictionPolicy="LRU">
     <persistence strategy="localTempSwap"/>
  </defaultCache>
</ehcache>
```

## About Default Cache

The *defaultCache* configuration is applied to any cache that is not explicitly configured. The defaultCache appears in the ehcache-failsafe.xml file by default, and can also be added to any Ehcache configuration file.

While the defaultCache configuration is not required, an error is generated if caches are created by name (programmatically) with no defaultCache loaded.

## Dynamically Changing Cache Configuration

While most of the Ehcache configuration is not changeable after startup, since Ehcache 2.0, certain cache configuration parameters can be modified dynamically at runtime. These include the following:

- Expiration settings
  - **timeToLive** – The maximum number of seconds an element can exist in the cache regardless of access. The element expires at this limit and will no longer be returned from the cache. The default value is 0, which means no TTL eviction takes place (infinite lifetime).
  - **timeToIdle** – The maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be returned from the cache. The default value is 0, which means no TTI eviction takes place (infinite lifetime).

  Note that the `eternal` attribute, when set to "true", overrides `timeToLive` and `timeToIdle` so that no expiration can take place.
- Local sizing attributes
  - `maxEntriesLocalHeap`
  - `maxBytesLocalHeap`
  - `maxEntriesLocalDisk`
  - `maxBytesLocalDisk`.
- memory-store eviction policy.
- CacheEventListeners can be added and removed dynamically

This example shows how to dynamically modify the cache configuration of a running cache:

```
Cache cache = manager.getCache("sampleCache");
CacheConfiguration config = cache.getCacheConfiguration();
config.setTimeToIdleSeconds(60);
config.setTimeToLiveSeconds(120);
config.setmaxEntriesLocalHeap(10000);
config.setmaxEntriesLocalDisk(1000000);
```

Dynamic cache configurations can also be disabled to prevent future changes:

```
Cache cache = manager.getCache("sampleCache");
cache.disableDynamicFeatures();
```

In ehcache.xml, you can disable dynamic configuration by setting the `<ehcache>` element's `dynamicConfig` attribute to "false".

## Passing Copies Instead of References

By default, a get() operation on a store returns a reference to the requested data, and any changes to that data are immediately reflected in the memory store. In cases where an application requires a *copy* of data rather than a reference to it, you can configure the store to return a copy. This allows you to change a copy of the data without affecting the original data in the memory store.

This is configured using the `copyOnRead` and `copyOnWrite` attributes of the <cache> and <defaultCache> elements in your configuration, or programmatically as follows:

```
CacheConfiguration config = new CacheConfiguration("copyCache", 1000)
                                  .copyOnRead(true).copyOnWrite(true);
Cache copyCache = new Cache(config);
```

The default configuration is “false” for both options.

To copy elements on put()-like and/or get()-like operations, a copy strategy is used. The default implementation uses serialization to copy elements. You can provide your own implementation of net.sf.ehcache.store.compound.CopyStrategy using the <copyStrategy> element:

```
<cache name="copyCache"
    maxEntriesLocalHeap="10"
    eternal="false"
    timeToIdleSeconds="5"
    timeToLiveSeconds="10"
    copyOnRead="true"
    copyOnWrite="true">
  <copyStrategy class="com.company.ehcache.MyCopyStrategy"/>
</cache>
```

A single instance of your CopyStrategy is used per cache. Therefore, in your implementation of CopyStrategy.copy(T), T must be thread-safe.

A copy strategy can be added programmatically in the following way:

```
CacheConfiguration cacheConfiguration = new CacheConfiguration("copyCache", 10);
CopyStrategyConfiguration copyStrategyConfiguration = new CopyStrategyConfiguration();
copyStrategyConfiguration.setClass("com.company.ehcache.MyCopyStrategy");
cacheConfiguration.addCopyStrategy(copyStrategyConfiguration);
```

# 2 Configuring Storage Tiers

## About Storage Tiers

Ehcache has three storage tiers, summarized here:

- **Memory store** – Heap memory that holds a copy of the hottest subset of data from the off-heap store. Subject to Java GC.
- **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.
- **Disk store** – Backs up in-memory data and provides overflow capacity to the other tiers. Can store serialized data only.

This document defines the standalone storage tiers and their suitable element types and then details the configuration for each storage tier.

Before running in production, it is strongly recommended that you test the tiers with the actual amount of data you expect to use in production. For information about sizing the tiers, refer to "Sizing Storage Tiers" on page 17.

## Configuring Memory Store

The memory store is always enabled and exists in heap memory. For the best performance, allot as much heap memory as possible without triggering garbage collection (GC) pauses, and use the off-heap store to hold the data that cannot fit in heap (without causing GC pauses).

The memory store has the following characteristics:

- Accepts all data, whether serializable or not
- Fastest storage option
- Thread safe for use by multiple concurrent threads

The memory store is the top tier and is automatically used by Ehcache to store the data hotset because it is the fastest store. 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.

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

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. 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. Legal values are LRU (default), LFU and FIFO:

- **Least Recently Used (LRU)**—LRU is the default setting. 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)** —For each get call on the element the number of hits is updated. 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)** — Elements are evicted in the same order as they come in. 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).

For all the eviction policies there are also putQuiet() and getQuiet() methods which do not update the last used timestamp.

When there is a get() or a getQuiet() on an element, it is checked for expiry. If expired, it is removed and null is returned. Note that at any point in time there will usually be some expired elements in the cache. Memory sizing of an application must always take into account the maximum size of each cache.

**Tip:** calculateInMemorySize() is a convenient method that can provide an estimate of the size (in bytes) of the memory store. It returns the serialized size of the cache, providing a rough estimate. Do not use this method in production as it is has a negative effect on performance.

An alternative is to have an expiry thread. This is a trade-off between lower memory use and short locking periods and CPU utilization. The design is in favor of the latter. For those concerned with memory use, simply reduce the tier size. For more information, refer to "Sizing Storage Tiers" on page 17.

## Configuring Disk Store

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

This section describes local disk usage. You can find additional information about configuring the disk store in "Configuring Restartability and Persistence" on page 33.

### Serialization

Only data that is Serializable can be placed in the disk store. 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.

Serialization speed is affected by the size of the objects being serialized and their type. It has been found 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 the Disk Store

Disk stores are configured on a per CacheManager basis. If one or more caches requires a disk store but none is configured, a default directory is used and a warning message is logged to encourage explicit configuration of the disk store path.

Configuring a disk store is optional. If all caches use only memory, then there is no need to configure a disk store. This simplifies configuration, and uses fewer threads. This also makes it unnecessary to configure multiple disk store paths when multiple CacheManagers are being used.

Two disk store options are available:

- Temporary store (`localTempSwap`)
- Persistent store (`localRestartable`)

#### localTempSwap

The `localTempSwap` persistence strategy allows the memory store to overflow to disk when it becomes full. This option makes the disk a temporary store because overflow data does not survive restarts or failures. When the node is restarted, any existing data on disk is cleared because it is not designed to be reloaded.

If the disk store path is not specified, a default path is used, and the default will be auto-resolved in the case of a conflict with another CacheManager.

The localTempSwap disk store creates a data file for each cache on startup called "*<cache_name>*.data".

#### localRestartable

This option implements a restartable store for all in-memory data. After any restart, the data set is automatically reloaded from disk to the in-memory stores.

The path to the directory where any required disk files will be created is configured with the `<diskStore>` sub-element of the Ehcache configuration. In order to use the restartable store, a unique and explicitly specified path is required.

### The diskStore Configuration Element

Files are created in the directory specified by the `<diskStore>` configuration element. The `<diskStore>` element has one attribute called `path`.

```
<diskStore path="/path/to/store/data"/>
```

Legal values for the path attribute are legal file system paths. For example, for Unix:

```
/home/application/cache
```

The following system properties are also legal, in which case they 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—for example, `java -Dehcache.disk.store.dir=/u01/myapp/diskdir`.

Subdirectories can be specified below the system property, for example:

```
user.dir/one
```

To programmatically set a disk store path:

```
DiskStoreConfiguration diskStoreConfiguration = new DiskStoreConfiguration();
diskStoreConfiguration.setPath("/my/path/dir");
// Already created a configuration object ...
configuration.addDiskStore(diskStoreConfiguration);
CacheManager mgr = new CacheManager(configuration);
```

**Note:** A CacheManager's disk store path cannot be changed once it is set in configuration. If the disk store path is changed, the CacheManager must be recycled for the new path to take effect.

### Disk Store Expiry and Eviction

Expired elements are eventually evicted to free up disk space. The element is also removed from the in-memory index of elements.

One thread per cache is used to remove expired elements. The optional attribute `diskExpiryThreadIntervalSeconds` sets the interval between runs of the expiry thread.

**Important:** Setting `diskExpiryThreadIntervalSeconds` to a low value can cause excessive disk-store locking and high CPU utilization. The default value is 120 seconds.

If a cache's disk store has a limited size, Elements will be evicted from the disk store when it exceeds this limit. The LFU algorithm is used for these evictions. It is not configurable or changeable.

**Note:** With the `localTempSwap` strategy, you can use `maxEntriesLocalDisk` or `maxBytesLocalDisk` at either the Cache or CacheManager level to control the size of the disk tier.

### Turning off Disk Stores

To turn off disk store path creation, comment out the `diskStore` element in ehcache.xml.

The default Ehcache configuration, ehcache-failsafe.xml, uses a disk store. To avoid use of a disk store, specify a custom ehcache.xml with the `diskStore` element commented out.

# 3 Sizing Storage Tiers

## The Sizing Attributes

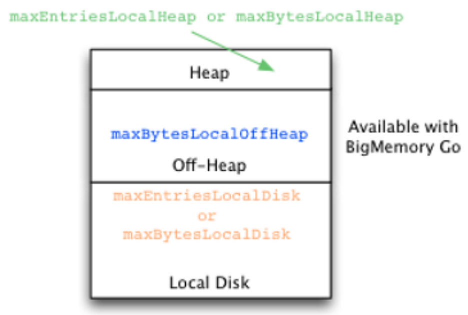
Tuning Ehcache often involves sizing the data storage tiers appropriately. You can size the different data tiers in a number of ways using simple sizing attributes. These sizing attributes affect memory and disk resources.

The following table summarizes the sizing attributes you can use.

| Tier | Attribute | Description |
|---|---|---|
| Memory Store (Heap) | `maxEntriesLocalHeapmax BytesLocalHeap` | The maximum number of entries or bytes a data set can use in local heap memory, or when set at the CacheManager level (maxBytesLocalHeap only), as a pool available to all data sets under that CacheManager. This setting is required for every cache or at the CacheManager level.<br><br>Pooling is available at the CacheManager level using `maxBytesLocalHeap` only. |
| Off-heap Store | `maxBytesLocalOffHeap` | The maximum number of bytes a data set can use in off-heap memory, or when set at the CacheManager level, as a pool available to all data sets under that CacheManager.<br><br>Pooling is available at the CacheManager level. |
| Disk Store | `maxEntriesLocalDiskmax BytesLocalDisk` | The maximum number of entries or bytes a data set can use on the local disk, or when set at the CacheManager level (maxBytesLocalDisk only), as a pool available to all data sets under that CacheManager. Note that these settings apply to temporary disk usage (`localTempSwap`); these settings do not apply to disk persistence.<br><br>Pooling is available at the CacheManager level using `maxBytesLocalDisk` only. |

Attributes that set a number of entries or elements take an integer. Attributes that set a memory size (bytes) use the Java -Xmx syntax (for example: “500k”, “200m”, “2g”) or percentage (for example: “20%”). Percentages, however, can be used only in the case where a CacheManager-level pool has been configured.

The following diagram illustrates the tiers and their effective sizing attributes.


maxEntriesLocalHeap or maxBytesLocalHeap
Heap
maxBytesLocalOffHeap
Off-Heap
Available with BigMemory Go
maxEntriesLocalDisk or maxBytesLocalDisk
Local Disk

## Pooling Resources Versus Sizing Individual Caches

You can constrain the size of any cache on a specific tier in that cache's configuration. You can also constrain the size of all of a CacheManager's caches in a specific tier by configuring an overall size at the CacheManager level.

If there is no CacheManager-level pool specified for a tier, an individual cache claims the amount of that tier specified in its configuration. If there is a CacheManager-level pool specified for a tier, an individual cache claims that amount *from the pool*. In this case, caches with no size configuration for that tier receive an equal share of the remainder of the pool (after caches with explicit sizing configuration have claimed their portion).

For example, if CacheManager with eight caches pools one gigabyte of heap, and two caches each explicitly specify 200MB of heap while the remaining caches do not specify a size, the remaining caches will share 600MB of heap equally. Note that caches must use bytes-based attributes to claim a portion of a pool; entries-based attributes such as `maxEntriesLocal` cannot be used with a pool.

On startup, the sizes specified by caches are checked to ensure that any CacheManager-level pools are not over-allocated. If over-allocation occurs for any pool, an InvalidConfigurationException is thrown. Note that percentages should not add up to more than 100% of a single pool.

If the sizes specified by caches for any tier take exactly the entire CacheManager-level pool specified for that tier, a warning is logged. In this case, caches that do not specify a size for that tier cannot use the tier as nothing is left over.

### Memory Store (Heap)

A size must be provided for the heap, either in the CacheManager (`maxBytesLocalHeap` only) or in each individual cache (`maxBytesLocalHeap` or `maxEntriesLocalHeap`). Not doing so causes an InvalidConfigurationException.

If a pool is configured, it can be combined with a heap setting in an individual cache. This allows the cache to claim a specified portion of the heap setting configured in the pool. However, in this case the cache setting must use `maxBytesLocalHeap` (same as the CacheManager).

In any case, every cache *must* have a heap setting, either configured explicitly or taken from the pool configured in the CacheManager.

### Local Disk Store

The local disk can be used as a data tier, either for temporary storage or for disk persistence, but not both at once.

To use the disk as a temporary tier during BigMemory operation, set the `persistenceStrategy` to "localTempSwap", and use the `maxBytesLocalDisk` setting to configure the size of this tier. For more information about using the disk as a temporary tier, see "Configuring Disk Store" on page 13.

For information about using the disk store for data persistence, see "Cache Persistence Implementation" on page 34.

## Sizing Examples

The following examples illustrate both pooled and individual cache-sizing configurations.

**Note:** Some of the following examples include allocations for off-heap storage. Off-heap data storage (i.e., the off-heap tier) is only available with the Terracotta BigMemory products.

### Pooled Resources

The following configuration sets pools for all of this CacheManager's caches:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="100M"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="50G">
...
<cache name="Cache1" ... </cache>
<cache name="Cache2" ... </cache>
<cache name="Cache3" ... </cache>
</ehcache>
```

CacheManager CM1 automatically allocates these pools equally among its three caches. Each cache gets one third of the allocated heap, off-heap, and local disk. Note that at the CacheManager level, resources can be allocated in bytes only.

### Explicitly Sizing Caches

You can explicitly allocate resources to specific caches:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="100M"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="60G">
...
<cache name="Cache1" ...
        maxBytesLocalHeap="50M"
        ...
  </cache>
<cache name="Cache2" ...
       maxBytesLocalOffHeap="5G"
       ...
  </cache>
<cache name="Cache3" ... </cache>
</ehcache>
```

In the example above, Cache1 reserves 50Mb of the 100Mb local-heap pool; the other caches divide the remaining portion of the pool equally. Cache2 takes half of the local off-heap pool; the other caches divide the remaining portion of the pool equally. Cache3 receives 25Mb of local heap, 2.5Gb of off-heap, and 20Gb of the local disk.

Caches that reserve a portion of a pool are not required to use that portion. Cache1, for example, has a fixed portion of the local heap but may have any amount of data in heap up to the configured value of 50Mb.

Note that caches must use the same sizing attributes used to create the pool. Cache1, for example, cannot use `maxEntriesLocalHeap` to reserve a portion of the pool.

### Mixed Sizing Configurations

If a CacheManager does not pool a particular resource, that resource can still be allocated in cache configuration, as shown in the following example.

```
<ehcache xmlns...
        Name="CM2"
        maxBytesLocalHeap="100M">
...
<cache name="Cache4" ...
        maxBytesLocalHeap="50M"
        maxEntriesLocalDisk="100000"
        ...
  </cache>
<cache name="Cache5" ...
       maxBytesLocalOffHeap="10G"
       ...
  </cache>
<cache name="Cache6" ... </cache>
</ehcache>
```

CacheManager CM2 creates one pool (local heap). Its caches all use the local heap and are constrained by the pool setting, as expected. However, cache configuration can allocate other resources as desired. In this example, Cache4 allocates disk space for its data, and Cache5 allocates off-heap space for its data. Cache6 gets 25Mb of local heap only.

### Using Percents

The following configuration sets pools for each tier:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="1G"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="50G">
...
<!-- Cache1 gets 400Mb of heap, 2.5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache1" ...
maxBytesLocalHeap="40%">
</cache>
<!-- Cache2 gets 300Mb of heap, 5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache2" ...
maxBytesLocalOffHeap="50%">
</cache>
<!-- Cache2 gets 300Mb of heap, 2.5Gb of off-heap, and 40Gb of disk. -->
<cache name="Cache3" ...
maxBytesLocalDisk="80%">
</cache>
</ehcache>
```

**Note:** You can use a percentage of the total JVM heap for the CacheManager maxBytesLocalHeap. The CacheManager percentage, then, is a portion of the total JVM heap, and in turn, the Cache percentage is the portion of the CacheManager pool for that tier.

### Sizing Without a Pool

The CacheManager in this example does not pool any resources.

```
<ehcache xmlns...
        Name="CM3"
       ... >
...
<cache name="Cache7" ...
        maxBytesLocalHeap="50M"
        maxEntriesLocalDisk="100000"
        ...
  </cache>
<cache name="Cache8" ...
       maxEntriesLocalHeap="1000"
       maxBytesLocalOffHeap="10G"
       ...
  </cache>
<cache name="Cache9" ...
       maxBytesLocalHeap="50M"
...
</cache>
</ehcache>
```

Caches can be configured to use resources as necessary. Note that every cache in this example must declare a value for local heap. This is because no pool exists for the local heap; implicit (CacheManager configuration) or explicit (cache configuration) local-heap allocation is required.

## Pinning and Size Limitations

Pinned caches can override the limits set by cache-configuration sizing attributes, potentially causing OutOfMemory errors. This is because pinning prevents flushing of cache entries to lower tiers. For more information on pinning, see "Pinning Data" on page 29.

## Built-In Sizing Computation and Enforcement

Internal Ehcache mechanisms track data-element sizes and enforce the limits set by CacheManager sizing pools.

### Sizing of Elements

Elements put in a memory-limited cache will have their memory sizes measured. The entire Element instance added to the cache is measured, including key and value, as well as the memory footprint of adding that instance to internal data structures. Key and value are measured as object graphs – each reference is followed and the object reference also measured. This goes on recursively.

Shared references will be measured by each class that references it. This will result in an overstatement. Shared references should therefore be ignored.

### Ignoring for Size Calculations

For the purposes of measurement, references can be ignored using the `@IgnoreSizeOf` annotation. The annotation may be declared at the class level, on a field, or on a package. You can also specify a file containing the fully qualified names of classes, fields, and packages to be ignored.

This annotation is not inherited, and must be added to any subclasses that should also be excluded from sizing.

The following example shows how to ignore the `Dog` class.

```
@IgnoreSizeOf
public class Dog {
  private Gender gender;
  private String name;
}
```

The following example shows how to ignore the `sharedInstance` field.

```
public class MyCacheEntry {
  @IgnoreSizeOf
  private final SharedClass sharedInstance;
    ...
}
```

Packages may be also ignored if you add the @IgnoreSizeOf annotation to appropriate package-info.java of the desired package. Here is a sample package-info.java for and in the com.pany.ignore package:

```
@IgnoreSizeOf
package com.pany.ignore;
import net.sf.ehcache.pool.sizeof.filter.IgnoreSizeOf;
```

Alternatively, you may declare ignored classes and fields in a file and specify a `net.sf.ehcache.sizeof.filter` system property to point to that file.

```
# That field references a common graph between all cached entries
com.pany.domain.cache.MyCacheEntry.sharedInstance
# This will ignore all instances of that type
com.pany.domain.SharedState
# This ignores a package
com.pany.example
```

Note that these measurements and configurations apply only to on-heap storage. Once Elements are moved to disk, they are serialized as byte arrays. The serialized size is then used as the basis for measurement.

### Configuration for Limiting the Traversed Object Graph

As noted above, sizing caches involves traversing object graphs, a process that can be limited with annotations. This process can also be controlled at both the CacheManager and cache levels.

### Size-Of Limitation at the CacheManager Level

Control how deep the size-of engine can go when sizing on-heap elements by adding the following element at the CacheManager level:

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

This element has the following attributes

- `maxDepth` – Controls how many linked objects can be visited before the size-of engine takes any action. This attribute is required.
- `maxDepthExceededBehavior` – Specifies what happens when the max depth is exceeded while sizing an object graph:
  - "continue" – (DEFAULT) Forces the size-of engine to log a warning and continue the sizing operation. If this attribute is not specified, "continue" is the behavior used.
  - "abort" – Forces the SizeOf engine to abort the sizing, log a warning, and mark the cache as not correctly tracking memory usage. With this setting, `Ehcache.hasAbortedSizeOf()` returns true.

The SizeOf policy can be configured at the CacheManager level (directly under `<ehcache>`) and at the cache level (under `<cache>` or `<defaultCache>`). The cache policy always overrides the CacheManager if both are set.

### Size-Of Limitation at the Cache level

Use the `<sizeOfPolicy>` as a sub-element in any `<cache>` block to control how deep the size-of engine can go when sizing on-heap elements belonging to the target cache. This cache-level setting overrides the CacheManager size-of setting.

### Debugging of Size-Of Related Errors

If warnings or errors appear that seem related to size-of measurement (usually caused by the size-of engine walking the graph), generate more log information on sizing activities:

- Set the `net.sf.ehcache.sizeof.verboseDebugLogging` system property to true.
- Enable debug logs on `net.sf.ehcache.pool.sizeof` in your chosen implementation of SLF4J.

## Eviction When Using CacheManager-Level Storage

When a CacheManager-level storage pool is exhausted, a cache is selected on which to perform eviction to recover pool space. The eviction from the selected cache is performed using the cache's configured eviction algorithm (LRU, LFU, etc...). The cache from which eviction is performed is selected using the “minimal eviction cost” algorithm described below:

```
eviction-cost = mean-entry-size * drop-in-hit-rate
```

Eviction cost is defined as the increase in bytes requested from the underlying SOR (System of Record, e.g., database) per unit time used by evicting the requested number of bytes from the cache.

If we model the hit distribution as a simple power-law then:

```
P(hit n'th element) ~ 1/n^{alpha}
```

In the continuous limit, this means the total hit rate is proportional to the integral of this distribution function over the elements in the cache. The change in hit rate due to an eviction is then the integral of this distribution function between the initial size and the final size. Assuming that the eviction size is small compared to the overall cache size, we can model this as:

```
drop ~ access * 1/x^{alpha} * Delta(x)
```

where “access” is the overall access rate (hits + misses), and x is a unit-less measure of the “fill level” of the cache. Approximating the fill level as the ratio of hit rate to access rate, and substituting in to the eviction-cost expression, we get:

```
eviction-cost = mean-entry-size * access * 1/(hits/access)^{alpha}
                   * (eviction / (byteSize / (hits/access)))
```

Simplifying:

```
eviction-cost = (byteSize / countSize) * access * 1/(h/A)^{alpha}
                   * (eviction * hits)/(access * byteSize)
eviction-cost = (eviction * hits) / (countSize * (hits/access)^{alpha})
```

Removing the common factor of "eviction", which is the same in all caches, lead us to evicting from the cache with the minimum value of:

```
 eviction-cost = (hits / countSize) / (hits/access)^{alpha}
```

When a cache has a zero hit-rate (it is in a pure loading phase), we deviate from this algorithm and allow the cache to occupy 1/nth of the pool space, where "n" is the number of caches using the pool. Once the cache starts to be accessed, we re-adjust to match the actual usage pattern of that cache.

# 4 Managing Data Life

## Configuration Options that Affect Data Life

This topic covers managing the life of the data in each of the data-storage tiers, including the pinning features of Automatic Resource Control (ARC).

You use the options to manage data life within the data-storage tiers:

- **Flush** – To move an entry to a lower tier. Flushing is used to free up resources while still keeping data in Ehcache .
- **Fault** – To copy an entry from a lower tier to a higher tier. Faulting occurs when data is required at a higher tier but is not resident there. The entry is not deleted from the lower tiers after being faulted.
- **Eviction** – To remove an entry from Ehcache. The entry is deleted; it can only be reloaded from an outside source. Entries are evicted to free up resources.
- **Expiration** – A status based on Time-To-Live and Time-To-Idle settings. To maintain performance, expired entries may not be immediately flushed or evicted.
- **Pinning** – To keep data in memory indefinitely.

## Setting Expiration

Data entries expire based on parameters with configurable values. When eviction occurs, expired elements are the first to be removed. Having an effective expiration configuration is critical to optimizing the use of resources such as heap and maintaining overall performance.

To add expiration, specify values for the following `<cache>` attributes, and tune these values based on results of performance tests:

- `timeToIdleSeconds` – The maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be returned from Ehcache . The default value is 0, which means no TTI eviction takes place (infinite lifetime).
- `timeToLiveSeconds` – The maximum number of seconds an element can exist in the cache regardless of use. The element expires at this limit and will no longer be returned from Ehcache . The default value is 0, which means no TTL eviction takes place (infinite lifetime).
- `maxEntriesLocalDisk` – The maximum sum total number of elements (cache entries) allowed on the disk tier for a cache. If this target is exceeded, eviction occurs to bring the count within the allowed target. The default value is 0, which means no eviction takes place (infinite size is allowed). *A setting of 0 means that no eviction of the cache's entries takes place, and consequently can cause the node to run out of disk space.*

- `eternal` – If the cache's `eternal` flag is set, it overrides any finite TTI/TTL values that have been set. Individual cache elements may also be set to eternal. Eternal elements and caches do not expire, however they may be evicted.

For information about how configuration can impact eviction, see "How Configuration Affects Element Flushing and Eviction" on page 30.

## Pinning Data

Without pinning, expired cache entries can be flushed and eventually evicted, and even most non-expired elements can also be flushed and evicted as well, if resource limitations are reached. Pinning gives per-cache control over whether data can be evicted from Ehcache .

Data that should remain in memory can be pinned. You cannot pin individual entries, only an entire cache. As described in the following topics, there are two types of pinning, depending upon whether the pinning configuration should take precedence over resource constraints or the other way around.

### Configuring Pinning

Entire caches can be pinned using the `pinning` element in the Ehcache configuration. This element has a required attribute (`store`) to specify how the pinning will be accomplished.

The `store` attribute can have either of the following values:

- inCache – Data is pinned in the cache, in any tier in which cache data is stored. The tier is chosen based on performance-enhancing efficiency algorithms. Unexpired entries can never be evicted.
- localMemory – Data is pinned to the memory store. Entries are evicted only in the event that the store's configured size is exceeded.

For example, the following cache is configured to pin its entries:

```
<cache name="Cache1" ... >
     <pinning store="inCache" />
</cache>
```

The following cache is configured to pin its entries to heap only:

```
<cache name="Cache2" ... >
     <pinning store="localMemory" />
</cache>
```

### Pinning and Cache Sizing

The interaction of the pinning configuration with the cache sizing configuration depends upon which pinning option is used.

For `inCache` pinning, the pinning setting takes priority over the configured cache size. Elements resident in a cache with this pinning option cannot be evicted if they have not expired. This type of pinned cache is not eligible for eviction at all, and `maxEntriesInCache` should not be configured for this cache.

**Important:** Potentially, pinned caches could grow to an unlimited size. Caches should never be pinned unless they are designed to hold a limited amount of data (such as reference data) or their usage and expiration characteristics are understood well enough to conclude that they cannot cause errors.

For `localMemory` pinning, the configured cache size takes priority over the pinning setting. `localMemory` pinning should be used for optimization, to keep data in heap memory, unless or until the tier becomes too full. If the number of entries surpasses the configured size, entries will be evicted. For example, in the following cache the `maxEntriesOnHeap` and `maxBytesLocalOffHeap` settings override the pinning configuration. (Off-heap storage is only available in the Terracotta BigMemory products.)

```
<cache name="myCache"
     maxEntriesOnHeap="10000"
     maxBytesLocalOffHeap="8g"
     ... >
    <pinning store="localMemory" />
</cache>
```

#### Scope of Pinning

Pinning achieved programmatically will not be persisted — after a restart the pinned entries are no longer pinned.

#### Explicitly Removing Data from a Pinned Cache

To unpin all of a cache's pinned entries, clear the cache. Specific entries can be removed from a cache using Cache.remove(). To empty the cache, Cache.removeAll(). If the cache itself is removed (Cache.dispose() or CacheManager.removeCache()), then any data still remaining in the cache is also removed locally. However, that remaining data is *not* removed from disk (if localRestartable).

## How Configuration Affects Element Flushing and Eviction

The following example shows a cache with certain expiration settings:

```
<cache name="myCache"
      eternal="false" timeToIdleSeconds="3600"
      timeToLiveSeconds="0" memoryStoreEvictionPolicy="LFU">
</cache>
```

Note the following about the myCache configuration:

- If a client accesses an entry in myCache that has been idle for more than an hour (`timeToIdleSeconds`), that element is evicted.
- If an entry expires but is not accessed, and no resource constraints force eviction, then the expired entry remains in place until a periodic evictor removes it.

- Entries in myCache can live forever if accessed at least once per 60 minutes (`timeToLiveSeconds`). However, unexpired entries may still be flushed based on other limitations. For details, see "Sizing Storage Tiers" on page 17.

## Data Freshness and Expiration

Databases and other systems of record (SORs) that were not built to accommodate caching outside of the database do not normally come with any default mechanism for notifying external processes when data has been updated or modified.

When using Ehcache as a caching system, the following strategies can help to keep the data in the cache in sync:

- **Data Expiration** Use the eviction algorithms included with Ehcache, along with the `timeToIdleSeconds` and `timetoLiveSeconds` settings, to enforce a maximum time for elements to live in the cache (forcing a re-load from the database or SOR).
- **Message Bus**: Use an application to make all updates to the database. When updates are made, post a message onto a message queue with a key to the item that was updated. All application instances can subscribe to the message bus and receive messages about data that is updated, and can synchronize their local copy of the data accordingly (for example by invalidating the cache entry for updated data)
- **Triggers**: Using a database trigger can accomplish a similar task as the message bus approach. Use the database trigger to execute code that can publish a message to a message bus. The advantage to this approach is that updates to the database do not have to be made only through a special application. The downside is that not all database triggers support full execution environments and it is often inadvisable to execute heavy-weight processing such as publishing messages on a queue during a database trigger.

The Data Expiration method is the simplest and most straightforward. It gives you the most control over the data synchronization, and doesn't require cooperation from any external systems. You simply set a data expiration policy and let Ehcache expire data from the cache, thus allowing fresh reads to re-populate and re-synchronize the cache.

If you choose the Data Expiration method, you can read more about the cache configuration settings in “Cache Eviction Algorithms” in the *Developer Guide* for Ehcache and review the timeToIdle and timeToLive configuration settings in “Setting Expiration.” The most important consideration when using the expiration method is balancing data freshness with database load. The shorter you make the expiration settings - meaning the more “fresh” you try to make the data - the more load you will place on the database.

Try out some numbers and see what kind of load your application generates. Even modestly short values such as five or ten minutes will produce significant load reductions.

# 5 Configuring Restartability and Persistence

## About Restartability and Persistence

Ehcache offers persistence using the local disk as a cache storage tier. While Ehcache offers various disk usage choices, as of version 2.6, the recommended option for persistence is the Fast Restart store, which is available in BigMemory Go and BigMemory Max. Open-source Ehcache offers a limited version of persistence, as noted in this document.

The Fast Restart feature provides enterprise-ready crash resilience with an option to store a fully consistent copy of the cache on the local disk at all times. The persistent storage of the cache on disk means that after any kind of shutdown — planned or unplanned — the next time that the application starts up, all of the previously cached data is still available and very quickly accessible.

The advantages of the Fast Restart store include:

- A persistent store of the cache on disk survives crashes, providing the fastest restart. Because cached data does not need to be reloaded from the data source after a crash, but is instead loaded from the local disk, applications can resume at full speed after restart. Recovery of even terabytes of data after a failure will be very fast, minimizing downtime.
- A persistent store on disk always contains a real-time copy of the cache, providing true fault tolerance. Even with BigMemory, where terabytes of data can be held in memory, the synchronous backup of data to disk provides the equivalent of a hot mirror right at the application and server nodes.
- A consistent copy of the cache on local disk provides many possibilities for business requirements, such as working with different datasets according to time-based needs or moving datasets around to different locations. It can range from a simple key-value persistence mechanism with fast read performance, to an operational store with in-memory speeds during operation for both reads and writes.

## Cache Persistence Implementation

Ehcache has a RestartStore which provides fast restartability and options for cache persistence. The RestartStore implements an on-disk mirror of the in-memory cache. After any restart, data that was last in the cache will automatically load from disk into the RestartStore, and from there the data will be available to the cache.

Data persistence is configured by adding the `<persistence>` sub-element to a cache configuration. The `<persistence>` sub-element includes two attributes: `strategy` and `synchronousWrites`.

### Strategy Options

The options for the `strategy` attribute are:

- `"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. *This option is available for BigMemory Go only.* For more information about this strategy, see the *BigMemory Configuration Guide* on the Terracotta Documentation website.
- `"localTempSwap"` — Enables temporary local disk usage. This option provides an extra tier for storage during cache operation, but this disk storage is not persisted. After a restart, the disk tier is cleared of any cache data.
- `"none"` — Does not offload data to disk. With this option, all of the working data is kept in memory only. This is the default mode.

### Synchronous Writes Options

If the `strategy` attribute is set to "localRestartable", then the `synchronousWrites` attribute can be configured. The options for `synchronousWrites` are:

- **synchronousWrites="false"** — This option specifies that an eventually consistent record of the data is kept on disk at all times. Writes to disk happen when efficient, and cache operations proceed without waiting for acknowledgment of writing to disk. After a restart, the data is recovered as it was when last synced. This option is faster than `synchronousWrites="true"`, but after a crash, the last 2-3 seconds of written data may be lost.

  If not specified, the default for `synchronousWrites` is "false".
- **synchronousWrites="true"** — This option specifies that a fully consistent record of the data is kept on disk at all times. As changes are made to the data set, they are synchronously recorded on disk. The write to disk happens before a return to the caller. After a restart, the data is recovered exactly as it was before shutdown. This option is slower than `synchronousWrites="false"`, but after a crash, it provides full data consistency.

  For transaction caching with `synchronousWrites`, soft locks are used to protect access. If there is a crash in the middle of a transaction, then upon recovery the soft locks are cleared on next access.

### DiskStore Path

he path to the directory where any required disk files will be created is configured with the `<diskStore>` sub-element of the Ehcache configuration.For "localTempSwap", if the DiskStore path is not specified, a default path is used for the disk storage tier, and the default path will be auto-resolved in the case of a conflict with another CacheManager.

## Configuration Examples

This section presents possible disk usage configurations for open-source Ehcache 2.6 and higher.

### Temporary Disk Storage

The "localTempSwap" persistence strategy allows the cache to use the local disk during cache operation. The disk storage is temporary and is cleared after a restart.

```
<ehcache>
  <diskStore path="/auto/default/path"/>
  <cache>
    <persistence strategy=”localTempSwap”/>
  </cache>
</ehcache>
```

**Note:** With the "localTempSwap" strategy, you can use `maxEntriesLocalDisk` or `maxBytesLocalDisk` at either the Cache or CacheManager level to control the size of the disk tier.

### In-memory Only Cache

When the persistence strategy is "none", all cache stays in memory (with no overflow to disk nor persistence on disk).

```
<cache>
  <persistence strategy=”none”/>
</cache>
```

### Programmatic Configuration Example

The following is an example of how to programmatically configure cache persistence on disk:

```
Configuration cacheManagerConfig = new Configuration()
    .diskStore(new DiskStoreConfiguration()
    .path("/path/to/store/data"));
CacheConfiguration cacheConfig = new CacheConfiguration()
    .name("my-cache")
    .maxBytesLocalHeap(16, MemoryUnit.MEGABYTES)
    .persistence(new PersistenceConfiguration()
        .strategy(Strategy.LOCALTEMPSWAP));
cacheManagerConfig.addCache(cacheConfig);
CacheManager cacheManager = new CacheManager(cacheManagerConfig);
Ehcache myCache = cacheManager.getEhcache("my-cache");
```

## Compatibility with Previous Versions

### Comparison of Disk Usage Options

The following table summarizes the configuration options for disk usage in Ehcache 2.6 and higher as compared with previous versions.

| Disk Usage | Ehcache 2.6 (and higher) | Ehcache 2.5 and Earlier |
|---|---|---|
| Fast Restartability | `persistence strategy="localRestartable” synchronousWrites="true"` | Not available |

| Disk Usage | Ehcache 2.6 (and higher) | Ehcache 2.5 and Earlier |
|---|---|---|
| with Strong Consistency | | |
| Fast Restartability with Eventual Consistency | `persistence strategy="localRestartable” synchronousWrites="false"` (Enterprise only) | Not available |
| Persistence for Clustered Caches | `persistence strategy="distributed"` | Remove or edit out any disk persistence configuration elements |
| Non-Fault-Tolerant Persistence | Use one of the fault-tolerant options above* | `overflowToDisk="true" diskpersistent="true"**` |
| Temporary Storage Tier | `persistence strategy="localTempSwap"` | `overflowToDisk="true" diskPersistent="false"` |
| In-memory Only (no disk usage) | `persistence strategy="none"` | |

*It is recommended to use one of the fault-tolerant options, however non-fault-tolerant persistence is still available. If `<persistence>` has not been specified, you can still use `overflowToDisk="true" diskPersistent="true"`.

**In Ehcache 2.5 and earlier, cache persistence on disk for standalone Ehcache is configured with the `overflowToDisk` and `diskPersistent` attributes. If both are set to "true", cached data is saved to disk asynchronously and can be recovered after a clean shutdown or planned flush. To prevent corrupt or inconsistent data from being returned, checking measures are performed upon a restart, and if any discrepancy is found, the cache that was stored on disk is emptied and must be reloaded from the data source.

### Upgrading from Ehcache 2.5 and Earlier

After upgrading from a version of Ehcache previous to 2.6, it is strongly recommended to add the `<persistence>` sub-element to your cache configuration, and to delete, disable, or edit out disk persistence configuration elements from previous versions. The previous elements include:

- `overflowToDisk`
- `diskPersistence`
- `DiskStoreBootstrapCacheLoaderFactory`

**Note:** If any of the elements above are specified in the same configuration with either the `<persistence>` sub-element or the `<terracotta>` sub-element, it will cause an Invalid Configuration Exception.

After upgrading, however, it is not mandatory to add the `<persistence>` sub-element. In Ehcache 2.6 or higher, disk persistence configuration elements from previous Ehcache versions will continue to be available with the same functionality, as long as the `<persistence>` sub-element has not been specified.

For cache persistence on disk, you should continue to use the `overflowToDisk` and `diskPersistent` attributes. For more information, refer to the “Persistence” section in the Ehcache 2.5 documentation.

# 6 Configuring the Update Checker

## Configuring the Update Checker

The update checker is used to see whether you have the latest version of Ehcache. It is also used to get non-identifying feedback on the operating system architectures using Ehcache. To disable the check, do one of the following.

Set the following system property:

```
-Dnet.sf.ehcache.skipUpdateCheck=true
```

Set the `updateCheck` attribute in the outer echace> element in the ehcache configuration file. This attribute is false, by default.

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="ehcache.xsd"
  updateCheck="false"
  monitoring="autodetect"
  dynamicConfig="true">
```

# A System Properties

## Special System Properties

### net.sf.ehcache.disabled

Setting this system property to `true` (using `java -Dnet.sf.ehcache.disabled=true` in the Java command line) disables caching in ehcache. If disabled, no elements can be added to a cache (puts are silently discarded).

### net.sf.ehcache.use.classic.lru

When LRU is selected as the eviction policy, set this system property to `true` (using `java -Dnet.sf.ehcache.use.classic.lru=true` in the Java command line) to use the older LruMemoryStore implementation. This is provided for ease of migration.