Ehcache API Developer Guide

Version 10.2

April 2018

TERRACOTTA

This document applies to Terracotta DB and Terracotta Ehcache Version 10.2 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: TC-EHC-DG-102-20180604

Table of Contents

1 Caching Basics

Primary Classes

At the core of the concept of the Ehcache API are three classes:

- CacheManager
- Cache
- UserManagedCache

A CacheManager provides all necessary functionality to manage Caches and associated Services.

A Cache contains mappings of key to value, so-called entries. There are methods to create, access, update and delete these key-value pairs.

A UserManagedCache allows to actively define a specific cache handling in cases where the full functionality of CacheManager may not be necessary, for instance when the lifecycle of the cache is shorter than the application lifecycle.

For more information on these three classes refer to the following sections ...

- "Comparison of CacheManager to UserManagedCache" on page 8
- "Creating and Configuring a CacheManager Using Java" on page 11
- "User Managed Caches" on page 35

... and the Java API docs of the Ehcache API on http://www.ehcache.org/documentation/.

Comparison of CacheManager to UserManagedCache

The first step is to create an instance that manages a cache, and the second step is to create the cache itself.

There are two variations of managing a cache:

By means of a CacheManager

or ...

By means of a UserManagedCache

The decision when to use either the standard CacheManager or the 'lightweight' UserManagedCache depends on the particular use case, since each approach has pros and cons:

CacheManager

Pros:

Offers numerous standard services out of the box - a good starting point for setting up the basic framework. Cons: Brings along a certain level of richness and complexity that in some cases might offer more than needed.
UserManagedCache Pros: Offers a lightweight approach for examples such as Method local caches, thread local caches, and a cache lifecycle shorter than the application lifecycle. Cons: Does not offer out-of-the-box services, which must be configured on-instance basis.

A CacheManager can be created using either Ehcache directly or the *JSR 107 JCACHE - Java Temporary Caching API*.

Programmatically configuring the instance of a CacheManager and its cache can be done either in Java or via XML.

2 Creating and Configuring a CacheManager Using Java

Going Through the Lifecycle of a Cache

Java configuration is most easily achieved through the use of builders that offer a fluent API.

The canonical way of dealing with a `Cache` is through a `CacheManager`. Creating, using and closing a cache with `CacheManager` is illustrated in this example:

```
CacheManager cacheManager
    = CacheManagerBuilder.newCacheManagerBuilder() // <1>
    .withCache("preConfigured",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
            String.class, ResourcePoolsBuilder.heap(10))) // <2>
    .build(); // <3>
cacheManager.init(); // <4>
Cache<Long, String> preConfigured = cacheManager.getCache("preConfigured",
    Long.class, String.class); // <5>
Cache<Long, String> myCache =
    cacheManager.createCache("myCache", // <6>
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
        String.class, ResourcePoolsBuilder.heap(10)));
myCache.put(1L, "da one!"); // <7>
String value = myCache.get(1L); // <8>
cacheManager.removeCache("preConfigured"); // <9>
cacheManager.close(); // <10>
```

The following list items refer to the commented numbers in the code.

<1> Using the Builders

The static method `org.ehcache.config.builders.CacheManagerBuilder.newCacheManagerBuilder` returns a new `org.ehcache.config.builders.CacheManagerBuilder` instance.

<2> Declare a cache configuration

Use the builder to define a `Cache` with alias "preConfigured". This cache will be created when `cacheManager.build()` is invoked on the actual `CacheManager` instance.

The first String argument is the cache alias, which is used to retrieve the cache from the `CacheManager`.

The second argument, `org.ehcache.config.CacheConfiguration`, is used to configure the `Cache`.

In this case, the static `newCacheConfigurationBuilder()` method is used on `org.ehcache.config.builders.CacheConfigurationBuilder` to create a default configuration.

<3> Instantiate a CacheManager

Invoking `build()` returns a fully instantiated, but uninitialized, `CacheManager` ready to use.

<4> Initialize the CacheManager

Before using the `CacheManager` it needs to be initialized, which can be done in 1 of 2 ways:

- Calling `CacheManager.init()` on the `CacheManager` instance, or
- Calling the `CacheManagerBuilder.build(boolean init)` method with the boolean parameter set to true.

<5> Retrieving the preConfigured Cache and Type-Safety

A cache is retrieved by passing its alias, key type and value type to the `CacheManager`. For instance, to obtain the cache declared in step 2 you need its `alias=preConfigured`, `keyType=Long.class` and `valueType=String.class`.

Asking for both key and value types to be passed in ensures type-safety. Should these differ from the ones expected, the `CacheManager` throws a `ClassCastException` early in the application's lifecycle. This also guards the `Cache` from being polluted by random types.

<6> Create a new Cache

The `CacheManager` can also be used to create new `Cache` as needed.

Just as in step 2, it requires passing an alias as well as a `CacheConfiguration`.

The instantiated and fully initialized `Cache` added will be returned and/or accessed through the `CacheManager.getCache` API.

<7> Store and ...

The newly added `Cache` can now be used to store entries, which are comprised of key value pairs. The put method's first parameter is the key and the second parameter is the value. Remember the key and value types must be the same types as those defined in the `CacheConfiguration`. Additionally the key must be unique and is only associated with one value.

<8> Retrieve data

A value is retrieved from a cache by calling the `cache.get(key)` method. It only takes one parameter which is the key, and returns the value associated with that key. If there is no value associated with that key then null is returned.

<9> Remove and close a given Cache

With `CacheManager.removeCache(String)` any given `Cache` can be removed.

The `CacheManager` will not only remove its reference to the `Cache`, but will also close it. The `Cache` releases all locally held transient resources (such as memory). References to this `Cache` become unusable.

<10> Close all Cache instances

In order to release all transient resources (memory, threads, ...) that a `CacheManager` provides to its managed `Cache` instances, `CacheManager.close()` needs to be invoked. This closes all `Cache` instances known at the time.

Configuring Storage Tiers using Java

Ehcache offers a tiering model that allows storing increased amounts of less frequently used data on slower tiers (which are generally more abundant).

More frequently used data (the "hottest data") would preferably be stored on faster (commonly less abundant) storage, whereas less frequently used data (less "hot" data) can be moved to slower (commonly more abundant) storage tiers.

Three Tiers

A classical example would be using 3 tiers with a persistent disk storage.

```
PersistentCacheManager persistentCacheManager =
    CacheManagerBuilder.newCacheManagerBuilder()
    .with(CacheManagerBuilder.persistence(
        new File(getStoragePath(), "myData")))                    // <1>
    .withCache("threeTieredCache",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(
            Long.class, String.class,
            ResourcePoolsBuilder.newResourcePoolsBuilder()
                .heap(10, EntryUnit.ENTRIES)                      // <2>
                .offheap(1, MemoryUnit.MB)                        // <3>
                .disk(20, MemoryUnit.MB, true)                    // <4>
            )
    ).build(true);
Cache<Long, String> threeTieredCache =
    persistentCacheManager.getCache("threeTieredCache",
    Long.class, String.class);
threeTieredCache.put(1L, "stillAvailableAfterRestart");           // <5>
persistentCacheManager.close();
```

1. If you wish to use disk storage (in the same way as for persistent `Cache` instances), you have to provide a location where data should be stored on disk to the `CacheManagerBuilder.persistence()` static method.
2. Define a resource pool for the heap. This will be your faster but smaller pool.
3. Define a resource pool for the off-heap. This is still quite fast and a bit bigger.
4. Define a persistent resource pool for the disk. It is persistent because the last parameter is `true`.
5. All values stored in the cache will be available after a JVM restart (assuming the `CacheManager` has been closed cleanly by calling `close()`).

Creating a Cache Manager with Clustering Support

To enable clustering with Terracotta, firstly you will have to start the Terracotta server configured with clustered storage. In addition, for creating the cache manager with clustering support, you will need to provide the clustering service configuration:

```
CacheManagerBuilder<PersistentCacheManager>
    clusteredCacheManagerBuilder =
    CacheManagerBuilder.newCacheManagerBuilder()              // <1>
        .with(ClusteringServiceConfigurationBuilder.cluster(URI.create(
            "terracotta://localhost:9410/my-application"))    // <2>
            .autoCreate());                                   // <3>
PersistentCacheManager cacheManager =
    clusteredCacheManagerBuilder.build(true);                 // <4>
cacheManager.close();                                         // <5>
```

<1>
Returns the `org.ehcache.config.builders.CacheManagerBuilder` instance;

<2>
Use the `ClusteringServiceConfigurationBuilder`'s static method `cluster(URI)` for connecting the cache manager to the clustering storage at the URI specified that returns the clustering service configuration builder instance. The sample URI provided in the example points to the clustered storage with clustered storage identifier `my-application` on the Terracotta server (assuming the server is running on localhost and port 9410); the query-param `auto-create` creates the clustered storage in the server if it doesn't already exist.

<3>
Returns a fully initialized cache manager that can be used to create clustered caches.

<4>
Close the cache manager.

Data Freshness

In Ehcache, data freshness is controlled through `Expiry`. The following example illustrates how to configure a *time-to-live* expiry.

```
CacheConfiguration<Long, String> cacheConfiguration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(
    Long.class, String.class, ResourcePoolsBuilder.heap(100))  // <1>
    .withExpiry(Expirations.timeToLiveExpiration(              // <2>
        Duration.of(20, TimeUnit.SECONDS)))
    .build();
```

1. Expiry is configured at the cache level, so start by defining a cache configuration,
2. then add to it an `Expiry`, here using the predefined *time-to-live* one, configured with the required `Duration`.

3 Configuring a CacheManager Using XML

Configuring Storage Tiers using XML

As an alternative to programmatic configuration, a `CacheManager` can also be configured using XML, as illustrated in this example:

```
<config>
  <cache alias="foo">                                          <!--1-->
    <key-type>java.lang.String</key-type>                      <!--2-->
    <resources>
      <heap unit="entries">2000</heap>                         <!--3-->
      <offheap unit="MB">500</offheap>                         <!--4-->
    </resources>
  </cache>
  <cache-template name="myDefaults">                           <!--5-->
    <key-type>java.lang.Long</key-type>
    <value-type>java.lang.String</value-type>
    <heap unit="entries">200</heap>
  </cache-template>
  <cache alias="bar" uses-template="myDefaults">               <!--6-->
    <key-type>java.lang.Number</key-type>
  </cache>
  <cache alias="simpleCache" uses-template="myDefaults" /> <!--7-->
</config>
```

1. Declares a `Cache` aliased to `foo`.
2. The keys of `foo` are declared as type `String`.

   Since the value type is not specified, the values will be of type `java.lang.Object`.
3. `foo` is declared to hold up to 2,000 entries on heap ...
4. ... as well as up to 500 MB of off-heap memory before the `Cache` starts evicting
5. `<cache-template>` elements create an abstract configuration that can be extended by further `<cache>` configurations
6. `bar` is an example for such a `Cache`.

   `bar` uses the `<cache-template>` named `myDefaults` and overrides its `key-type` to a wider type.
7. `simpleCache` is another such `Cache`.

   `simpleCache` uses `myDefaults` configuration for its sole `CacheConfiguration`.

The schema and format of the XML is explained in detail in the following section *The XML Schema Definition*.

The type `XmlConfiguration` allows for parsing an XML configuration:

```
final URL myUrl = this.getClass().getResource("/my-config.xml");    // <1>
Configuration xmlConfig = new XmlConfiguration(myUrl);              // <2>
CacheManager myCacheManager =
    CacheManagerBuilder.newCacheManager(xmlConfig);                 // <3>
```

As the steps are ...

1. Obtain a URL to the location of the XML file.

2. Instantiate an `XmlConfiguration` passing the URL of the XML file.
3. Create the `CacheManager` instance using the `Configuration` from the `XmlConfiguration`.

The XML Schema Definition

The following schema elements are available when using an XML file for configuring a `CacheManager` at creation time:

`<config>` - root element
This is the root element of our XML configuration.

One `<config>` element in an XML file provides the definition for a `CacheManager`.

> **Note:** You can create multiple `CacheManager` instances using the same XML configuration file.
>
> Advanced:
>
> In contrast to the JSR-107 `javax.cache.spi.CachingProvider`, Ehcache does not maintain a registry of `CacheManager` instances.

`<service>`
`<service>` elements are extension points for specifying services managed by the `CacheManager`.

Each such `Service` defined in this way is managed with the same lifecycle as the `CacheManager`.

The `Service.start` is called during `CacheManager.init` processing.

The `Service.stop` method is called during `CacheManager.close` processing.

These `Service` instances can then be used by `Cache` instances managed by the `CacheManager`.

> **Note:** JSR-107 uses this extension point of the XML configuration (and Ehcache's modular architecture). For more information see section *The JSR-107 Provider*.

`<default-serializers>`
A `<default-serializers>` element represents `Serializers` configured at `CacheManager` level. It is a collection of `<serializer>` elements that require a `type` and a fully qualified class name of the `Serializer`.

`<default-copiers>`
A `<default-copiers>` element represents `Copiers` configured at `CacheManager` level. It is a collection of `<copier>` elements that requires a `type` and a fully qualified class name of the `Copier`.

<persistence>
A <persistence> element that represents Persistence and that needs to be used when creating a PersistentCacheManager.

It requires the directory location where data will be stored on disk.

<cache>
A <cache> element represent a Cache instance that will be created and managed by the CacheManager.

Each <cache> requires the alias attribute, used at runtime to retrieve the corresponding Cache<K, V> instance using the org.ehcache.CacheManager.getCache(String, Class<K>, Class<V>) method.

The optional uses-template attribute references a <cache-template> element's name attribute.

See the section *XML Programmatic Parsing* for more details.

Following nested elements are optionally available:

- <key-type>

  the fully qualified class name (FQCN) of the keys (<K>) held in the Cache<K, V>; defaults to java.lang.Object

- <value-type>

  FQCN of the values (<V>) held in the Cache; defaults to java.lang.Object

- <expiry>

  control the expiry type and its parameters

- <eviction-advisor>

  FQCN of a org.ehcache.config.EvictionAdvisor<K, V> implementation, defaults to null, i.e. none

- <integration>

  configure a CacheLoaderWriter for a cache-through pattern

- <resources>

  configure the tiers and their capacity. When using on-heap only, you can replace this element by the <heap> one.

<cache-template>
<cache-template> elements represent a uniquely named template for <cache> elements to inherit from.

This unique name is specified by using the mandatory name attribute.

A <cache> element that references a <cache-template> by its name using the uses-template attribute, will inherit all properties of the <cache-template>. A <cache> can override these properties as it needs.

A `<cache-template>` element may contain all the same child elements as a `<cache>` element.

Property replacement in XML configuration files

Java system properties can be referenced inside XML configuration files. The property value will replace the property reference during the configuration parsing.

This is done by using the `${prop.name}` syntax. It is supported in all attributes and elements values that accept the `${}` characters as legal characters. This currently rules out all numbers, mostly used in sizing things, and identifiers, such as cache and template names.

Note: If the system property does not exist, this will make the configuration parsing fail.

A classical use case for this feature is for providing a disk file location inside the `directory` attribute of the `persistence` tag:

```
<persistence directory="${user.home}/cache-data"/>  <!-- 1 -->
```

1	Here `user.home` will be replaced by the value of the system property, for example `/home/user`.

XML Programmatic Parsing

The following section goes through the steps and possibilities of automatically configuring a `CacheManager` using XML.

Note: If you are obtaining your `CacheManager` through API calls based on JSR-107, what follows is done automatically when invoking `javax.cache.spi.CachingProvider.getCacheManager(java.net.URI, java.lang.ClassLoader)`.

```
final URL myUrl =
    getClass().getResource("/configs/docs/getting-started.xml"); // <1>
XmlConfiguration xmlConfig = new XmlConfiguration(myUrl); // <2>
CacheManager myCacheManager =
    CacheManagerBuilder.newCacheManager(xmlConfig); // <3>
myCacheManager.init();  // <4>
```

1. Obtain a `URL` to your XML file's location
2. Instantiate an `XmlConfiguration` passing the URL of the XML file to it.
3. Create your `CacheManager` instance using the `Configuration` from the `XmlConfiguration`.
4. Initialize the `CacheManager` before it is used.

We can also use `<cache-template>` declared in the XML file to seed instances of `CacheConfigurationBuilder`. In order to use a `<cache-template>` element from an XML file, the XML file contains the following XML fragment:

```
  <cache-template name="example">
    <key-type>java.lang.Long</key-type>
    <value-type>java.lang.String</value-type>
    <heap unit="entries">200</heap>
  </cache-template>
```

Creating a `CacheConfigurationBuilder` of that example `<cache-template>` element would be done as follows:

```
XmlConfiguration xmlConfiguration = new XmlConfiguration(getClass()
    .getResource("/configs/docs/template-sample.xml"));
CacheConfigurationBuilder<Long, String> configurationBuilder =
xmlConfiguration.newCacheConfigurationBuilderFromTemplate("example",
    Long.class, String.class); // <1>
configurationBuilder = configurationBuilder.
    withResourcePools(ResourcePoolsBuilder.heap(1000)); // <2>
```

1. Creates a builder, inheriting the capacity constraint of 200 entries.
2. The inherent properties can be overridden by simply providing a different value prior to building the `CacheConfiguration`.

4 The JCache (JSR-107) Cache Provider

Overview of JCache

The Java Temporary Caching API (JSR-107), also referred to as JCache, is a specification (not a software implementation) that defines the `javax.cache` API. The specification was developed under the Java Community Process, and its purpose is to provide standardized caching concepts and mechanisms for Java applications.

The API is simple to use, it is designed as a caching standard and is vendor-neutral. It eliminates the stark contrast that has in the past existed between APIs of different vendors, which caused developers to stick with the proprietary API they were already using, rather than investigating a new API, as the bar to investigating other products was too high.

So it is easy for you as an application developer to develop an application using the JCache API from one vendor, then if you so choose, try out another vendor's JCache support without having to change a single line of your application code. All you have to do is use the JCache caching library from your chosen vendor. This means you can avoid having to rewrite a lot of your caching related code in an application just to try out a new caching solution.

Using Ehcache as a JCache Provider

To use JCache API calls in an application, you require both of the following jar files:

- The JCache jar, which defines the JCache APIs.
- The Ehcache jar, which is the caching provider jar that implements the JCache APIs. It translates the JCache API calls to their Ehcache API equivalent.

You can use the JCache API to develop a complete application, without the need to use any Ehcache API calls.

Setting up Ehcache as the Caching Provider for JCache

To use Ehcache as the caching provider for your application, add the file javax.cache:cache-api:1.*y.y* .jar (where *y.y* is a version-dependent string) to your application's classpath. This is of course assuming Ehcache is already on that same classpath.

No other setup steps are required.

The JCache jar file is available as a download from the JSR-107 section of the web pages of the Java Community Process.

Note: If you were already using JCache with another caching provider, ensure that you remove the other provider's jar file before starting your application.

Getting Started with Ehcache and JCache (JSR-107)

In addition to the `Cache` interface, the JCache specification defines the interfaces `CachingProvider` and `CacheManager`. Applications need to use a `CacheManager` to create/retrieve a `Cache`. Similarly a `CachingProvider` is required to get/access a `CacheManager`.

Here is a code sample that demonstrates the usage of the basic JCache configuration APIs:

```
CachingProvider provider = Caching.getCachingProvider();  // <1>
CacheManager cacheManager = provider.getCacheManager();   // <2>
MutableConfiguration<Long, String> configuration =
    new MutableConfiguration<Long, String>()  // <3>
        .setTypes(Long.class, String.class)   // <4>
        .setStoreByValue(false)   // <5>
        .setExpiryPolicyFactory(
            CreatedExpiryPolicy.factoryOf(Duration.ONE_MINUTE));  // <6>
Cache<Long, String> cache =
    cacheManager.createCache("jCache", configuration); // <7>
cache.put(1L, "one"); // <8>
String value = cache.get(1L); // <9>
assertThat(value, is("one"));
```

1	Retrieves the default `CachingProvider` implementation from the application's classpath. This method will work if and only if there is exactly one JCache implementation jar in the classpath. If there are multiple providers in your classpath then use the fully qualified name `org.ehcache.jsr107.EhcacheCachingProvider` to retrieve the Ehcache caching provider. You can do this by using the `Caching.getCachingProvider(String)` static method instead.
2	Retrieve the default `CacheManager` instance using the provider.
3	Create a cache configuration using `MutableConfiguration`…
4	with key type and value type as Long and String respectively…
5	configured to store the cache entries by reference (not by value)…
6	and with an expiry time of one minute defined for entries from the moment they are created.
7	Using the cache manager, create a cache named jCache with the configuration created in step <3>.
8	Put some data into the cache.

9	Retrieve the data from the same cache.

Integrating JCache and Ehcache Configurations

As mentioned already, JCache offers a minimal set of configuration that is ideal for an in-memory cache. But Ehcache native APIs support topologies that are much more complex and provide more features. At times, application developers might want to configure caches that are much complex (in terms of topology or features) than the ones that the JCache `MutableConfiguration` permits, and still be able to use JCache's caching APIs. Ehcache provides several ways to achieve this, as described in the following sections.

Accessing the underlying Ehcache configuration from a JCache configuration

When you create a `Cache` on a `CacheManager` using a `MutableConfiguration` - in other words, using only JCache types - you can still get to the underlying Ehcache`CacheRuntimeConfiguration`:

```
MutableConfiguration<Long, String> configuration =
    new MutableConfiguration<Long, String>();
configuration.setTypes(Long.class, String.class);
Cache<Long, String> cache = cacheManager.createCache("someCache",
    configuration); // <1>
CompleteConfiguration<Long, String> completeConfiguration =
    cache.getConfiguration(CompleteConfiguration.class); // <2>
Eh107Configuration<Long, String> eh107Configuration =
    cache.getConfiguration(Eh107Configuration.class); // <3>
CacheRuntimeConfiguration<Long, String> runtimeConfiguration =
    eh107Configuration.unwrap(CacheRuntimeConfiguration.class); // <4>
```

1	Create a JCache cache using the `MutableConfiguration` interface from the JCache specification.
2	Get to the JCache `CompleteConfiguration`.
3	Get to the configuration bridge connecting Ehcache and JCache.
4	Unwrap to the Ehcache`CacheRuntimeConfiguration` type.

CacheManager level configuration

If you need to configure features at the `CacheManager` level, like the persistence directory, you will have to use provider specific APIs.

The way you do this is as follows:

```
CachingProvider cachingProvider = Caching.getCachingProvider();
EhcacheCachingProvider ehcacheProvider =
   (EhcacheCachingProvider) cachingProvider;    // 1
DefaultConfiguration configuration =
```

```
  new DefaultConfiguration(ehcacheProvider.getDefaultClassLoader(),
  new DefaultPersistenceConfiguration(getPersistenceDirectory())); // 2
CacheManager cacheManager = ehcacheProvider.getCacheManager(
   ehcacheProvider.getDefaultURI(), configuration); // 3
```

1	Cast the `CachingProvider` into the Ehcache specific implementation `org.ehcache.jsr107.EhcacheCachingProvider`,
2	Create a configuration using the specific Ehcache`DefaultConfiguration` and pass it some `CacheManager` level configurations,
3	Create the `CacheManager` using the method that takes an Ehcache configuration as a parameter.

Cache level configuration

You can also create a JCache `Cache` using an Ehcache`CacheConfiguration`. When using this mechanism, no JCache `CompleteConfiguration` is used and so you cannot get to one.

```
CacheConfiguration<Long, String> cacheConfiguration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
        String.class,
    ResourcePoolsBuilder.heap(10)).build(); // <1>
Cache<Long, String> cache = cacheManager.createCache("myCache",
    Eh107Configuration.fromEhcacheCacheConfiguration(cacheConfiguration)); // <2>
Eh107Configuration<Long, String> configuration =
    cache.getConfiguration(Eh107Configuration.class);
configuration.unwrap(CacheConfiguration.class); // <3>
configuration.unwrap(CacheRuntimeConfiguration.class); // <4>
try {
  cache.getConfiguration(CompleteConfiguration.class); // <5>
  throw new AssertionError("IllegalArgumentException expected");
} catch (IllegalArgumentException iaex) {
  // Expected
}
```

1	Create an Ehcache`CacheConfiguration`. You can use a builder as shown here, or alternatively use an XML configuration (as described in the following section).
2	Get a JCache configuration by wrapping the Ehcache configuration.
3	Get back to the Ehcache`CacheConfiguration`.
4	... or even to the runtime configuration.
5	No JCache `CompleteConfiguration` is available in this context.

Building the JCache configuration using an Ehcache XML configuration

Another way to have the full Ehcache configuration options on your JCache caches while having no code dependency on Ehcache as the cache provider is to use XML-based configuration. See "Configuring a CacheManager Using XML" on page 17 for more details on configuring caches in XML.

The following is an example of an XML configuration:

```
<config
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns='http://www.ehcache.org/v3'
    xsi:schemaLocation="
    http://www.ehcache.org/v3 http://www.ehcache.org/schema/ehcache-core-3.0.xsd">
  <cache alias="ready-cache">
    <key-type>java.lang.Long</key-type>
    <value-type>com.pany.domain.Product</value-type>
    <loader-writer>
      <class>com.pany.ehcache.integration.ProductCacheLoaderWriter</class>
    </loader-writer>
    <heap unit="entries">100</heap>
  </cache>
</config>
```

Here is an example of how to access the XML configuration using JCache:

```
CachingProvider cachingProvider = Caching.getCachingProvider();
CacheManager manager = cachingProvider.getCacheManager( // <1>
    getClass().getResource("/org/ehcache/docs/ehcache-jsr107-config.xml")
        .toURI(), // <2>
    getClass().getClassLoader()); // <3>
Cache<Long, Product> readyCache = manager.getCache("ready-cache",
    Long.class, Product.class); // <4>
```

1	Invoke `javax.cache.spi.CachingProvider.getCacheManager(java.net.URI, java.lang.ClassLoader)`
2	... and pass in a URI that resolves to an Ehcache XML configuration file.
3	The second argument is the `ClassLoader` to use to load user types if needed; i.e. `Class` instances that are stored in the `Cache` managed by our `CacheManager`.
4	Get the configured `Cache` out of the `CacheManager`.

Note: You can alternatively use the `CachingProvider.getCacheManager()` method that takes no arguments. The `URI` and `ClassLoader` used to configure the `CacheManager` will then use the vendor specific values returned by `CachingProvider.getDefaultURI` and `.getDefaultClassLoader` respectively. Be aware that these are not entirely specified for Ehcache currently and may change in future releases!

Enabling/Disabling MBeans for JCache using an Ehcache XML configuration

When using an Ehcache XML configuration, you may want to enable management and / or statistics MBeans for JCache caches. This gives you control over the following:

- `javax.cache.configuration.CompleteConfiguration.isStatisticsEnabled`
- `javax.cache.configuration.CompleteConfiguration.isManagementEnabled`

You can do this at two different levels:

```
<config
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns='http://www.ehcache.org/v3'
    xmlns:jsr107='http://www.ehcache.org/v3/jsr107'
    xsi:schemaLocation="
        http://www.ehcache.org/v3
        http://www.ehcache.org/schema/ehcache-core-3.0.xsd
        http://www.ehcache.org/v3/jsr107
        http://www.ehcache.org/schema/ehcache-107-ext-3.0.xsd">
  <service>
    <jsr107:defaults enable-management="true" enable-statistics="true"/> <!--1-->
  </service>
  <cache alias="stringCache"> <!--2-->
    <key-type>java.lang.String</key-type>
    <value-type>java.lang.String</value-type>
    <heap unit="entries">2000</heap>
  </cache>
  <cache alias="overrideCache">
    <key-type>java.lang.String</key-type>
    <value-type>java.lang.String</value-type>
    <heap unit="entries">2000</heap>
    <jsr107:mbeans enable-management="false" enable-statistics="false"/> <!--3-->
  </cache>
  <cache alias="overrideOneCache">
    <key-type>java.lang.String</key-type>
    <value-type>java.lang.String</value-type>
    <heap unit="entries">2000</heap>
    <jsr107:mbeans enable-statistics="false"/> <!--4-->
  </cache>
</config>
```

1	Using the JCache service extension, you can enable MBeans by default.
2	The cache `stringCache` will have both MBeans enabled, according to the service configuration.
3	The cache `overrideCache` will have both MBeans disabled, overriding the service configuration.
4	The cache `overrideOneCache` will have the statistics MBean disabled, whereas the management MBean will be enabled according to the service configuration.

Supplementing JCache cache configurations using Ehcache XML extensions

You can also create `cache-templates`. See the *Cache Templates* topic of the section "The XML Schema Definition" on page 19 for more details. Ehcache as a caching provider for JCache comes with an extension to the regular XML configuration so you can:

1. Configure a default template from which all programmatically created `Cache` instances inherit, and
2. Configure a given named `Cache` to inherit from a specific template.

This feature is particularly useful to configure `Cache` beyond the scope of the JCache specification, for example, giving `Cache` a capacity constraint. To do this, add a `jsr107` service in your XML configuration file:

```
<config
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns='http://www.ehcache.org/v3'
    xmlns:jsr107='http://www.ehcache.org/v3/jsr107'
    xsi:schemaLocation="
        http://www.ehcache.org/v3
            http://www.ehcache.org/schema/ehcache-core-3.0.xsd
        http://www.ehcache.org/v3/jsr107
            http://www.ehcache.org/schema/ehcache-107-ext-3.0.xsd"> <!--1-->
  <service> <!--2-->
    <jsr107:defaults default-template="tinyCache"> <!--3-->
      <jsr107:cache name="foos" template="clientCache"/> <!--4-->
      <jsr107:cache name="byRefCache" template="byRefTemplate"/>
      <jsr107:cache name="byValCache" template="byValueTemplate"/>
      <jsr107:cache name="weirdCache1" template="mixedTemplate1"/>
      <jsr107:cache name="weirdCache2" template="mixedTemplate2"/>
    </jsr107:defaults>
  </service>
  <cache-template name="clientCache">
    <key-type>java.lang.String</key-type>
    <value-type>com.pany.domain.Client</value-type>
    <expiry>
      <ttl unit="minutes">2</ttl>
    </expiry>
    <heap unit="entries">2000</heap>
  </cache-template>
  <cache-template name="tinyCache">
    <heap unit="entries">20</heap>
  </cache-template>
  <cache-template name="byRefTemplate">
    <key-type copier=
        "org.ehcache.impl.copy.IdentityCopier">java.lang.Long</key-type>
    <value-type copier=
        "org.ehcache.impl.copy.IdentityCopier">com.pany.domain.Client</value-type>
    <heap unit="entries">10</heap>
  </cache-template>
  <cache-template name="byValueTemplate">
    <key-type copier=
        "org.ehcache.impl.copy.SerializingCopier">java.lang.Long</key-type>
    <value-type copier=
        "org.ehcache.impl.copy.SerializingCopier">com.pany.domain.Client</value-type>
    <heap unit="entries">10</heap>
  </cache-template>
  <cache-template name="mixedTemplate1">
    <key-type copier=
        "org.ehcache.impl.copy.IdentityCopier">java.lang.Long</key-type>
```

```
    <value-type copier=
        "org.ehcache.impl.copy.SerializingCopier">com.pany.domain.Client</value-type>
    <heap unit="entries">10</heap>
  </cache-template>
  <cache-template name="mixedTemplate2">
    <key-type copier=
        "org.ehcache.impl.copy.SerializingCopier">java.lang.Long</key-type>
    <value-type copier=
        "org.ehcache.impl.copy.IdentityCopier">com.pany.domain.Client</value-type>
    <heap unit="entries">10</heap>
  </cache-template>
</config>
```

1	First, declare a namespace for the JCache extension, e.g. `jsr107`.
2	Within a service element at the top of your configuration, add a `jsr107:defaults` element.
3	The element takes an optional attribute `default-template`, which references the `cache-template` to use for all `javax.cache.Cache` elements created by the application at runtime using `javax.cache.CacheManager.createCache`. In this example, the default `cache-template` used will be `tinyCache`, meaning that in addition to their particular configuration, any programmatically created `Cache` instances will have their capacity constrained to 20 entries.
4	Nested within the `jsr107:defaults` element, add specific `cache-templates` to use for the given named `Cache`. So, for example, when creating the `Cache` named `foos` at runtime, Ehcache will enhance its configuration, giving it a capacity of 2000 entries, as well as ensuring that both key and value types are `String`.

Note: The XSD schema definitions that describe the syntax used for the Ehcache XML configuration are referenced at "XSD namespaces and locations" on page 102.

Using the above configuration, you can not only supplement but also override the configuration of JCache-created caches without modifying the application code.

```
MutableConfiguration<Long, Client> mutableConfiguration =
    new MutableConfiguration<Long, Client>();
mutableConfiguration.setTypes(Long.class, Client.class); // <1>
Cache<Long, Client> anyCache =
    manager.createCache("anyCache", mutableConfiguration); // <2>
CacheRuntimeConfiguration<Long, Client> ehcacheConfig =
    (CacheRuntimeConfiguration<Long, Client>)anyCache.getConfiguration(
    Eh107Configuration.class).unwrap(CacheRuntimeConfiguration.class); // <3>
ehcacheConfig.getResourcePools().
    getPoolForResource(ResourceType.Core.HEAP).getSize(); // <4>
Cache<Long, Client> anotherCache =
    manager.createCache("byRefCache", mutableConfiguration);
assertFalse(anotherCache.
    getConfiguration(Configuration.class).isStoreByValue()); // <5>
```

```
MutableConfiguration<String, Client> otherConfiguration =
    new MutableConfiguration<String, Client>();
otherConfiguration.setTypes(String.class, Client.class);
otherConfiguration.setExpiryPolicyFactory(CreatedExpiryPolicy.
    factoryOf(Duration.ONE_MINUTE)); // <6>
Cache<String, Client> foosCache =
    manager.createCache("foos", otherConfiguration);// <7>
CacheRuntimeConfiguration<Long, Client> foosEhcacheConfig =
    (CacheRuntimeConfiguration<Long, Client>)foosCache.getConfiguration(
    Eh107Configuration.class).unwrap(CacheRuntimeConfiguration.class);
Client client1 = new Client("client1", 1);
foosEhcacheConfig.getExpiry().getExpiryForCreation(42L, client1).
    getLength(); // <8>
CompleteConfiguration<String, String> foosConfig =
    foosCache.getConfiguration(CompleteConfiguration.class);
try {
  final Factory<ExpiryPolicy> expiryPolicyFactory =
      foosConfig.getExpiryPolicyFactory();
  ExpiryPolicy expiryPolicy = expiryPolicyFactory.create(); // <9>
  throw new AssertionError("Expected UnsupportedOperationException");
} catch (UnsupportedOperationException e) {
  // Expected
}
```

1	Assume existing JCache configuration code, which is store-by-value by default
2	... that creates JCache `Cache`.
3	If you were to get to the Ehcache`RuntimeConfiguration`
4	... you could verify that the template configured capacity is applied to the cache and returns 20 here.
5	The cache template will override the JCache cache's store-by-value configuration to store-by-reference, since the `byRefTemplatetemplate` that is used to create the cache is configured explicitly using `IdentityCopier`.
6	Templates will also override the JCache configuration, in this case using a configuration with Time to Live (TTL) 1 minute.
7	Create a cache where the template sets the TTL to 2 minutes.
8	And we can indeed verify that the configuration provided in the template has been applied; the duration will be 2 minutes and not 1 minute.

9	One drawback of this is that when getting at the `CompleteConfiguration`, you no longer have access to the factories from JCache.

Note: As mentioned in step 5, in order to override the store-by-value configuration of a JCache cache using templates, you can explicitly configure the template using `IdentityCopier`. But the usage of `IdentityCopier` is not mandatory to get a store-by-reference cache. You can use any custom copier implementation that does not perform any "copying" but returns the exact same reference that gets passed into the copy methods. `IdentityCopier` is just an example that we have provided for your convenience.

Differences in Default Behavior between Ehcache and Ehcache through JCache

Ehcache used natively and Ehcache used through JCache do not always agree on default behavior. While native Ehcache can behave the way JCache specifies, depending on the used configuration mechanism, you may see differences in defaults.

by-reference or by-value

Ehcache and Ehcache through JCache disagree on the default mode for heap-only caching.

Ehcache configuration with JCache `MutableConfiguration`

Unless you invoke `MutableConfiguration.setStoreByValue(boolean)`, the default value is "true". This means that you will be limited to `Serializable` keys and values when using Ehcache.

Under the cover, this will trigger the use of serializing copiers and pick the appropriate serializer from the default ones. See the section "Serializers and Copiers" on page 79 for related information.

Ehcache configuration with native XML or code

- Heap only: When using heap only caches, the default is by-reference unless you configure a `Copier`.
- Other tiering configuration: When using any other tiers, since serialization comes into play the default is "by-value".

See the sections "Copiers" on page 85 and "Serializers" on page 80 for related information.

Cache-through and compare-and-swap operations

Ehcache and Ehcache through JCache disagree on the role of the cache loader for compare-and-swap operations.

Ehcache through JCache behaviour

When using compare-and-swap operations, such as `putIfAbsent(K, V)`, the cache loader will not be used if the cache has no mapping present. If the `putIfAbsent(K, V)` succeeds then the cache writer will be used to propagate the update to the system of record. This could result in the cache behaving like INSERT but effectively causing a blind update on the underlying system of record.

Native Ehcache behaviour

The `CacheLoaderWriter` will always be used to load missing mappings with and to write updates. This enables the `putIfAbsent(K, V)` in cache-through to behave as an INSERT on the system of record.

If you need *Ehcache through JCache* behaviour, the following shows the relevant XML configuration:

```
<service>
  <jsr107:defaults jsr-107-compliant-atomics="true"/>
</service>
```

5 User Managed Caches

Overview of User Managed Caches

What are user managed caches and what do they offer?

A user managed cache gives you a simple way to configure a cache directly, without the complexity of setting up or using a `CacheManager`. The choice whether to use a `UserManagedCache` rather than a `CacheManager` usually depends on whether you need all of the built-in functionality of a `CacheManager`. In cases where your cache requirements are relatively straightforward, and you do not require the full range of features of a `CacheManager`, consider using a `UserManagedCache` instead.

Typical scenarios for using a `UserManagedCache` are: method local caches, thread local caches or any other place where the lifecycle of the cache is shorter than the application lifecycle.

API Extensions

User Managed Cache

If you use a `UserManagedCache`, you need to configure all required services by hand.

The `UserManagedCache` class extends the `Cache` class by offering additional methods:

- `init()` - initializes the cache
- `close()` - releases the cache resources
- `getStatus()` - returns a status

The `init` and `close` methods deal with the lifecycle of the cache and need to be called explicitly, whereas these methods are hidden when the cache is inside a `CacheManager`.

The interface definition is shown in this code:

```
package org.ehcache;
import java.io.Closeable;
/**
 * Represents a {@link Cache} that is not managed by a
 *   {@link org.ehcache.CacheManager CacheManager}.
 * <P>
 *   These caches must be {@link #close() closed} in order to release
 *   all their resources.
 * </P>
 *
 * @param <K> the key type for the cache
 * @param <V> the value type for the cache
 */
public interface UserManagedCache<K, V> extends Cache<K, V>, Closeable {
  /**
   * Transitions this {@code UserManagedCache} to
   *     {@link org.ehcache.Status#AVAILABLE AVAILABLE}.
   * <P>
   * If an error occurs before the {@code UserManagedCache} is {@code AVAILABLE},
   * it will revert to {@link org.ehcache.Status#UNINITIALIZED UNINITIALIZED}
```

```
   * and attempt to properly release all resources.
   * </P>
   *
   * @throws IllegalStateException if the {@code UserManagedCache} is not
   *     {@code UNINITIALIZED}
   * @throws StateTransitionException if the {@code UserManagedCache} could not
   *     be made {@code AVAILABLE}
   */
  void init() throws StateTransitionException;
  /**
   * Transitions this {@code UserManagedCache} to
   *     {@link Status#UNINITIALIZED UNINITIALIZED}.
   * <P>
   *   This will release all resources held by this cache.
   * </P>
   * <P>
   *   Failure to release a resource will not prevent other resources from being
   *   released.
   * </P>
   *
   * @throws StateTransitionException if the {@code UserManagedCache} could not
   *     reach {@code UNINITIALIZED} cleanly
   * @throws IllegalStateException if the {@code UserManagedCache} is not
   *     {@code AVAILABLE}
   */
  @Override
  void close() throws StateTransitionException;
  /**
   * Returns the current {@link org.ehcache.Status Status} of this
   *     {@code UserManagedCache}.
   *
   * @return the current {@code Status}
   */
  Status getStatus();
}
```

User Managed Persistent Cache

A user managed persistent cache holds cached data in a persistent store such as disk, so that the stored data can outlive the JVM in which your caching application runs.

If you want to create a user managed persistent cache, there is an additional interface `PersistentUserManagedCache` that extends `UserManagedCache` and adds the `destroy` method.

The `destroy` method deletes all data structures, including data stored persistently on disk, for a `PersistentUserManagedCache`.

The `destroy` method deals with the lifecycle of the cache and needs to be called explicitly.

The interface definition is shown in this code:

```
package org.ehcache;
/**
 * A {@link UserManagedCache} that holds data that can outlive the JVM.
 *
 * @param <K> the key type for the cache
 * @param <V> the value type for the cache
 */
public interface PersistentUserManagedCache<K, V>
   extends UserManagedCache<K, V> {
```

```
  /**
   * Destroys all persistent data structures for this
   *    {@code PersistentUserManagedCache}.
   *
   * @throws java.lang.IllegalStateException if state
   *    {@link org.ehcache.Status#MAINTENANCE MAINTENANCE} couldn't be reached
   * @throws CachePersistenceException if the persistent data cannot be destroyed
   */
  void destroy() throws CachePersistenceException;
}
```

Code examples for User Managed Caches

Example of a basic cache lifecycle

Here is a simple example showing a basic lifecycle of a user managed cache:

```
UserManagedCache<Long, String> userManagedCache =
    UserManagedCacheBuilder.newUserManagedCacheBuilder(Long.class, String.class)
        .build(false); // <1>
userManagedCache.init(); // <2>
userManagedCache.put(1L, "The one!"); // <3>
userManagedCache.close(); // <4>
```

1	Create a `UserManagedCache` instance. You can either pass `true` to have the builder `init()` it for you, or you can pass `false` and it is up to you to `init()` it prior to using it.
2	Since `false` was passed in <1>, you have to `init()` the `UserManagedCache` prior to using it.
3	You can use the cache exactly as a managed cache.
4	In the same vein, a `UserManagedCache` requires you to close it explicitly using `UserManagedCache.close()`. If you are also using managed caches simultaneously, the `CacheManager.close()` operation would not impact the user managed cache(s).

From this basic example, explore the API of UserManagedCacheBuilder in code or through Javadoc to discover all the directly available features. The following features apply in the exact same way to user managed caches:

- Serializers and copiers. See the section "Serializers and Copiers" on page 79 for related information.
- Eviction advisor. See the section "Eviction Advisors" on page 77 for related information.

Simply use the methods from `UserManagedCacheBuilder` which are equivalent to the ones from `CacheConfigurationBuilder`.

Below we will describe a more advanced setup where you need to maintain a service instance in order to have a working user managed cache.

Example with disk persistence and lifecycle

If you want to use disk persistent cache, you will need to create and lifecycle the persistence service.

```
LocalPersistenceService persistenceService = new DefaultLocalPersistenceService(
    new DefaultPersistenceConfiguration(
        new File(getStoragePath(), "myUserData")));  // <1>
PersistentUserManagedCache<Long, String> cache =
    UserManagedCacheBuilder.newUserManagedCacheBuilder(Long.class, String.class)
    .with(new UserManagedPersistenceContext<Long, String>("cache-name",
          persistenceService)) // <2>
    .withResourcePools(ResourcePoolsBuilder.newResourcePoolsBuilder()
        .heap(10L, EntryUnit.ENTRIES)
        .disk(10L, MemoryUnit.MB, true)) // <3>
    .build(true);
// Work with the cache
cache.put(42L, "The Answer!");
assertThat(cache.get(42L), is("The Answer!"));
cache.close(); // <4>
cache.destroy(); // <5>
persistenceService.stop(); // <6>
```

1	Create the persistence service to be used by the cache for storing data on disk.
2	Pass the persistence service to the builder as well as a name for the cache. Note that this will make the builder produce a more specific type: `PersistentUserManagedCache`.
3	As usual, indicate here if the data should outlive the cache.
4	Closing the cache will not delete the data it saved on disk, since the cache is marked as persistent.
5	To delete the data on disk after closing the cache, you need to invoke the `destroy` method explicitly.
6	You need to stop the persistence service once you have finished using the cache.

Example with cache event listeners

Cache event listeners require executor services in order to work. You will have to provide either a `CacheEventDispatcher` implementation or make use of the default one by providing two executor services: one for ordered events and one for unordered ones.

Note: The ordered events executor must be single threaded to guarantee ordering.

For more information on cache event listeners, see the section "Cache Event Listeners" on page 73.

```
UserManagedCache<Long, String> cache =
    UserManagedCacheBuilder.newUserManagedCacheBuilder(Long.class, String.class)
    .withEventExecutors(Executors.newSingleThreadExecutor(),
        Executors.newFixedThreadPool(5)) // <1>
    .withEventListeners(CacheEventListenerConfigurationBuilder
        .newEventListenerConfiguration(ListenerObject.class, EventType.CREATED,
            EventType.UPDATED)
        .asynchronous()
        .unordered()) // <2>
    .withResourcePools(ResourcePoolsBuilder.newResourcePoolsBuilder()
        .heap(3, EntryUnit.ENTRIES))
    .build(true);
cache.put(1L, "Put it");
cache.put(1L, "Update it");
cache.close();
```

1	Provide the `ExecutorService` for ordered and unordered event delivery.
2	Provide a listener configuration using `CacheEventListenerConfigurationBuilder`.

6 Cache Usage Patterns

There are several common access patterns when using a cache. Ehcache supports the following patterns:

Cache-aside

With the cache-aside pattern, application code uses the cache directly.

This means that application code which accesses the system-of-record (SoR) should consult the cache first, and if the cache contains the data, then return the data directly from the cache, bypassing the SoR. Otherwise, the application code must fetch the data from the system-of-record, store the data in the cache, and then return it. When data is written, the cache must be updated along with the system-of-record.

Pseudocode for reading values

```
v = cache.get(k)
if(v == null) {
    v = sor.get(k)
    cache.put(k, v)
}
```

Pseudocode for writing values

```
v = newV
sor.put(k, v)
cache.put(k, v)
```

Cache-as-SoR

The cache-as-SoR pattern implies using the cache as though it were the primary system-of-record (SoR).

The pattern delegates SoR reading and writing activities to the cache, so that application code is (at least directly) absolved of this responsibility. To implement the cache-as-SoR pattern, use a combination of the following read and write patterns:

- read-through
- write-through or write-behind

Advantages of using the cache-as-SoR pattern are:

- Less cluttered application code (improved maintainability through centralized SoR read/write operations)
- Choice of write-through or write-behind strategies on a per-cache basis
- Allows the cache to solve the *thundering-herd* problem

A disadvantage of using the cache-as-SoR pattern is:

- Less directly visible code-path

Read-through

Under the read-through pattern, the cache is configured with a *loader* component that knows how to load data from the system-of-record (SoR).

When the cache is asked for the value associated with a given key and such an entry does not exist within the cache, the cache invokes the loader to retrieve the value from the SoR, then caches the value, then returns it to the caller.

The next time the cache is asked for the value for the same key it can be returned from the cache without using the loader (unless the entry has been evicted or expired).

Write-through

Under the write-through pattern, the cache is configured with a *writer* component that knows how to write data to the system-of-record (SoR).

When the cache is asked to store a value for a key, the cache invokes the writer to store the value in the SoR, as well as updating the cache.

Write-behind

The write-behind pattern changes the timing of the write to the system-of-record. Rather than writing to the system-of-record while the thread making the update waits (as with write-through), write-behind queues the data for writing at a later time. This allows the user's thread to move along more quickly, at the cost of introducing some lag in time before the SoR is updated.

7 Data Freshness and Expiry

Data Freshness

Many databases and other systems of record (SORs) are not built to accommodate caching outside of the database. This means that they do not normally come with any default mechanism for notifying external processes when data has been updated or modified. If Ehcache is used to cache data from such a database or SOR, Ehcache will not be automatically informed if data in the database or SOR has changed.

This leads to the idea of *data freshness*: if a set of data in the cache still largely matches (in other words, most of the cached data entries are still *in sync with*) the data in the original data in the database or SOR, the data is termed *fresh*, but if many changes in the database or SOR have occurred without the cache also being updated, the data in the cache becomes increasing less fresh - it becomes *stale*.

When using Ehcache as a caching system, the following strategies can help to keep the data in the cache *fresh*, i.e. in sync with the database or SOR:

- **Data Expiry:** Use the eviction algorithms included with Ehcache, along with the time-to-idle (TTI) and time-to-live (TTL) settings, to enforce a maximum time for elements to live in the cache (forcing a re-load from the database or SOR). See the section "Expiry" on page 45 for related information.
- **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 Expiry strategy 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 expiry 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 Expiry strategy, the most important consideration is balancing data freshness with database load. The shorter you make the expiry settings - meaning the more "fresh" you try to make the data - the more load you will place on the database.

Try out some numbers for time-to-idle and time-to-live and see what kind of load your application generates. Even modestly short values such as five or ten minutes can produce significant load reductions.

Expiry

Introduction

Expiry is one of the key aspects of caching. In Ehcache this is addressed with the `Expiry` interface and its use in controlling the age of cache mappings.

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

Both Java and XML offer direct support for three types of expiry:

no expiry	If this setting is selected, cache entries do not expire, so they remain in the cache without a time limit; however they may be evicted. This setting overrides any finite TTI/TTL values that have been set. Individual cache elements may also receive this setting.
time-to-live (TTL)	The maximum number of seconds an element can exist in the cache, regardless of whether it is used or not. 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).
time-to-idle (TTI)	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).

For Java configuration, see `org.ehcache.expiry.Expirations`. For XML configuration, see the XSD schema.

Configuration

Expiry is configured at the cache level, in Java or in XML:

```
CacheConfiguration<Long, String> cacheConfiguration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
        String.class, ResourcePoolsBuilder.heap(100))  // 1
    .withExpiry(Expirations.timeToLiveExpiration(Duration.of(20,
        TimeUnit.SECONDS)))  // 2
    .build();
```

1	Expiry is configured at the cache level, so start by defining a cache configuration,
2	then add to it an `Expiry`, here using the predefined *time-to-live* one, configured with the required `Duration`.

```
<cache alias="withExpiry">
  <expiry>
    <ttl unit="seconds">20</ttl>  <!-- 1 -->
  </expiry>
  <heap>100</heap>
</cache>
```

1	At the cache level, using the predefined *time-to-live* again.

Read on to implement your own expiration scheme.

Custom expiry

Support your own expiration scheme simply means implementing the `Expiry` interface:

```
/**
 * A policy object that governs expiration for mappings
 * in a {@link org.ehcache.Cache Cache}.
 * <P>
 *   Previous values are not accessible directly but are rather available
 *   through a {@link ValueSupplier value supplier}
 *   to indicate that access can require computation (such as deserialization).
 * </P>
 * <P>
 * NOTE: Some cache configurations (eg. caches with eventual consistency) may
 * use local (ie. non-consistent) state
 * to decide whether to call {@link #getExpiryForUpdate(Object, ValueSupplier,
 * Object)}  vs. {@link #getExpiryForCreation(Object, Object)}.
 * For these cache configurations it is advised to return the same
 * value for both of these methods
 * </P>
 * <P>
 * See {@link Expirations} for helper methods to create common {@code Expiry}
 * instances.
 * </P>
 *
 * @param <K> the key type for the cache
 * @param <V> the value type for the cache
 *
 * @see Expirations
 */
public interface Expiry<K, V> {
  /**
   * Returns the lifetime of an entry when it is initially added to a
   * {@link org.ehcache.Cache Cache}.
   * <P>
   *   This method must not return {@code null}.
   * </P>
   * <P>
   *   Exceptions thrown from this method will be swallowed and result in
   *   the expiry duration being
```

```
   *   {@link Duration#ZERO ZERO}.
   * </P>
   *
   * @param key the key of the newly added entry
   * @param value the value of the newly added entry
   * @return a non-null {@link Duration}
   */
  Duration getExpiryForCreation(K key, V value);
  /**
   * Returns the expiration {@link Duration} (relative to the current time)
   * when an existing entry is accessed from a
   * {@link org.ehcache.Cache Cache}.
   * <P>
   *   Returning {@code null} indicates that the expiration time
   *   remains unchanged.
   * </P>
   * <P>
   *   Exceptions thrown from this method will be swallowed and result
   *   in the expiry duration being
   *   {@link Duration#ZERO ZERO}.
   * </P>
   *
   * @param key the key of the accessed entry
   * @param value a value supplier for the accessed entry
   * @return an expiration {@code Duration}, {@code null} means unchanged
   */
  Duration getExpiryForAccess(K key, ValueSupplier<? extends V> value);
  /**
   * Returns the expiration {@link Duration} (relative to the current time)
   * when an existing entry is updated in a
   * {@link org.ehcache.Cache Cache}.
   * <P>
   *   Returning {@code null} indicates that the expiration time
   *   remains unchanged.
   * </P>
   * <P>
   *   Exceptions thrown from this method will be swallowed and
   *   result in the expiry duration being
   *   {@link Duration#ZERO ZERO}.
   * </P>
   *
   * @param key the key of the updated entry
   * @param oldValue a value supplier for the previous value of the entry
   * @param newValue the new value of the entry
   * @return an expiration {@code Duration}, {@code null} means unchanged
   */
  Duration getExpiryForUpdate(K key, ValueSupplier<? extends V> oldValue,
      V newValue);
}
```

The main points to remember on the return value from these methods:

some `Duration`	indicates that the mapping will expire after that duration,
`Duration.ZERO`	indicates that the mapping is immediately expired,
`Duration.INFINITE`	indicates that the mapping will never expire,

`null Duration`	indicates that the previous expiration time is to be left unchanged, illegal at mapping creation time.

Note that you can access the details of the mapping, thus providing expiration times that are different per mapping.

Also when used from XML, Ehcache expects your expiry implementation to have a *no-arg* constructor.

Once you have implemented your own expiry, simply configure it.

In Java:

```
CacheConfiguration<Long, String> cacheConfiguration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
        String.class,
        ResourcePoolsBuilder.heap(100))
    .withExpiry(new CustomExpiry())   // 1
    .build();
```

1	Simply pass your custom expiry instance into the cache builder.

In XML:

```
<cache alias="withCustomExpiry">
  <expiry>
    <class>com.pany.ehcache.MyExpiry</class>   <!-- 1 -->
  </expiry>
  <heap>100</heap>
</cache>
```

1	Simply pass the fully qualified class name of your custom expiry.

For an example of how to migrate per-mapping code from Ehcache v2, see the section "Migrating Code from Ehcache v2" on page 141.

8 Transactions Support

What is supported and what are the limitations?

Ehcache supports caches that work within the context of an XA transaction controlled by a Java Transaction API (JTA) transaction manager. Within this context, Ehcache supports the two-phase commit protocol, including crash recovery.

- Bitronix Transaction Manager 2.1.4, which is an open source project hosted on GitHub, is the only tested transaction manager. Other transaction managers may work but have not yet been tested.
- Read-Committed is the only supported isolation level.
- The isolation level is guaranteed by the use of the `Copier` mechanism. When no copiers are configured for either the key or the value, default ones are automatically used instead. You cannot disable the `Copier` mechanism for a transactional cache.
- Accessing a cache outside of a JTA transaction context is forbidden.
- There is no protection against the ABA problem.
- Everything else works orthogonally.

Configuring it all in Java

The simplest case

The simplest possible configuration is to configure a cache manager as transactionally aware by using the provided Bitronix transaction manager integration.

This `INFO` level log entry informs you of the detected transaction manager:

```
INFO org.ehcache.transactions.xa.txmgr.btm.BitronixTransactionManagerLookup -
Using looked up transaction manager :
    a BitronixTransactionManager with 0 in-flight transaction(s)
```

Here is an example:

```
BitronixTransactionManager transactionManager =
    TransactionManagerServices.getTransactionManager();  // 1
CacheManager cacheManager = CacheManagerBuilder.newCacheManagerBuilder()
    .using(new LookupTransactionManagerProviderConfiguration(
        BitronixTransactionManagerLookup.class))    // 2
    .withCache("xaCache", CacheConfigurationBuilder.newCacheConfigurationBuilder(
        Long.class, String.class,                   // 3
        ResourcePoolsBuilder.heap(10))              // 4
        .add(new XAStoreConfiguration("xaCache"))   // 5
        .build()
    )
    .build(true);
final Cache<Long, String> xaCache = cacheManager.getCache("xaCache", Long.class,
    String.class);
transactionManager.begin();    // 6
{
  xaCache.put(1L, "one");      // 7
}
```

```
transactionManager.commit();   // 8
cacheManager.close();
transactionManager.shutdown();
```

1	First start the Bitronix transaction manager. By default, Ehcache will auto-detect it but will throw an exception during the cache manager initialization if BTM isn't started.
2	Configure the cache manager such as it can handle transactions by having a `TransactionManagerProvider` loaded and configured to use Bitronix.
3	Register a cache the normal way.
4	Give it the resources you wish.
5	Add a `XAStoreConfiguration` object to make the cache XA transactional. You must also give the cache a unique XAResource identifier as some transaction managers require this.
6	Begin a JTA transaction the normal way.
7	Work with the cache the normal way, all operations are supported. Note that concurrent transactions will not see those pending changes.
8	Commit the JTA transaction. Other transactions can now see the changes you made to the cache.

Configuring your transaction manager

While only the Bitronix JTA implementation has been tested so far, plugging-in another one is possible.

You will need to implement a `org.ehcache.transactions.xa.txmgr.provider.TransactionManagerLookup` and make sure you understand its expected lifecycle as well as the one of the `org.ehcache.transactions.xa.txmgr.provider.LookupTransactionManagerProvider`.

If such a lifecycle does not match your needs, you will have to go one step further and implement your own `org.ehcache.transactions.xa.txmgr.provider.TransactionManagerProvider`.

XA write-through cache

When a XA cache is configured in write-though mode, the targeted SoR will automatically participate in the JTA transaction context. Nothing special needs to be

configured for this to happen, just ensure that the configured `CacheLoaderWriter` is configured to work with XA transactions.

```
BitronixTransactionManager transactionManager =
    TransactionManagerServices.getTransactionManager();  // 1
Class<CacheLoaderWriter<?, ?>> klazz =
    (Class<CacheLoaderWriter<?, ?>>) (Class) (SampleLoaderWriter.class);
CacheManager cacheManager = CacheManagerBuilder.newCacheManagerBuilder()
    .using(new LookupTransactionManagerProviderConfiguration(
        BitronixTransactionManagerLookup.class))       // 2
    .withCache("xaCache", CacheConfigurationBuilder.newCacheConfigurationBuilder(
        Long.class, String.class,                      // 3
        ResourcePoolsBuilder.heap(10))                 // 4
            .add(new XAStoreConfiguration("xaCache"))  // 5
            .add(new DefaultCacheLoaderWriterConfiguration(klazz,
                 singletonMap(1L, "eins")))            // 6
            .build()
    )
    .build(true);
final Cache<Long, String> xaCache = cacheManager.getCache("xaCache",
    Long.class, String.class);
transactionManager.begin();                        // 7
{
  assertThat(xaCache.get(1L), equalTo("eins"));  // 8
  xaCache.put(1L, "one");                          // 9
}
transactionManager.commit();                       // 10
cacheManager.close();
transactionManager.shutdown();
```

1	First start the Bitronix transaction manager. By default, Ehcache will auto-detect it but will throw an exception during the cache manager initialization if BTM isn't started.
2	Configure the cache manager such as it can handle transactions by having a `TransactionManagerProvider` loaded and configured to use Bitronix.
3	Register a cache the normal way.
4	Give it the resources you wish.
5	Add a `XAStoreConfiguration` object to make the cache XA transactional. You must also give the cache a unique XAResource identifier as some transaction managers require this.
6	Add a `CacheLoaderWriter` configuration. This one is a mocked SoR backed by a map for illustration purpose that is filled with `1L/"eins"` key/value pair at startup.
7	Begin a JTA transaction the normal way.

8	The cache is empty at startup, so the `CacheLoaderWriter` will be called to load the value.
9	Update the value. This will make the `CacheLoaderWriter` write to the SoR.
10	Commit the JTA transaction. Other transactions can now see the changes you made to the cache and the SoR.

Transactional scope

A XA cache can only be accessed within a JTA transaction's context. Any attempt to access one outside of such context will result in `XACacheException` to be thrown.

```
BitronixTransactionManager transactionManager =
    TransactionManagerServices.getTransactionManager();  // 1
CacheManager cacheManager = CacheManagerBuilder.newCacheManagerBuilder()
    .using(new LookupTransactionManagerProviderConfiguration(
        BitronixTransactionManagerLookup.class))    // 2
    .withCache("xaCache", CacheConfigurationBuilder.newCacheConfigurationBuilder(
        Long.class, String.class,                   // 3
        ResourcePoolsBuilder.heap(10))              // 4
        .add(new XAStoreConfiguration("xaCache"))   // 5
        .build()
    )
    .build(true);
final Cache<Long, String> xaCache = cacheManager.getCache("xaCache",
    Long.class, String.class);
try {
  xaCache.get(1L);   // 6
  fail("expected XACacheException");
} catch (XACacheException e) {
  // expected
}
cacheManager.close();
transactionManager.shutdown();
```

1	First start the Bitronix transaction manager. By default, Ehcache will auto-detect it but will throw an exception during the cache manager initialization if BTM isn't started.
2	Configure the cache manager such as it can handle transactions by having a `TransactionManagerProvider` loaded and configured to use Bitronix.
3	Register a cache the normal way.
4	Give it the resources you wish.
5	Add a `XAStoreConfiguration` object to make the cache XA transactional. You must also give the cache a unique XAResource identifier as some transaction managers require this.

6	The cache is being accessed with no prior call to `transactionManager.begin()` which makes it throw `XACacheException`.

Note: there is one exception to that rule: the `Cache.clear()` method will always wipe the cache's contents non-transactionally.

XA cache with three tiers and persistence

When a cache is configured as persistent, the in-doubt transactions are preserved and can be recovered across restarts.

This `INFO` log informs you about that in-doubt transactions journaling is persistent too:

```
INFO o.e.t.x.j.DefaultJournalProvider - Using persistent XAStore journal
```

Here is an example:

```
BitronixTransactionManager transactionManager =
    TransactionManagerServices.getTransactionManager();  // 1
PersistentCacheManager persistentCacheManager =
    CacheManagerBuilder.newCacheManagerBuilder()
    .using(new LookupTransactionManagerProviderConfiguration(
        BitronixTransactionManagerLookup.class))         // 2
    .with(new CacheManagerPersistenceConfiguration(new File(getStoragePath(),
        "testXACacheWithThreeTiers")))                   // 3
    .withCache("xaCache", CacheConfigurationBuilder.newCacheConfigurationBuilder(
        Long.class, String.class,                        // 4
        ResourcePoolsBuilder.newResourcePoolsBuilder()   // 5
                    .heap(10, EntryUnit.ENTRIES)
                    .offheap(10, MemoryUnit.MB)
                    .disk(20, MemoryUnit.MB, true)
            )
            .add(new XAStoreConfiguration("xaCache"))    // 6
            .build()
    )
    .build(true);
final Cache<Long, String> xaCache = persistentCacheManager.getCache("xaCache",
    Long.class, String.class);
transactionManager.begin();    // 7
{
  xaCache.put(1L, "one");      // 8
}
transactionManager.commit();   // 9
persistentCacheManager.close();
transactionManager.shutdown();
```

1	First start the Bitronix transaction manager. By default, Ehcache will auto-detect it but will throw an exception during the cache manager initialization if BTM isn't started.
2	Configure the cache manager such as it can handle transactions by having a `TransactionManagerProvider` loaded and configured to use Bitronix.
3	Configure persistence support to enable the use of the disk tier.

4	Register a cache the normal way.
5	Give it the resources you want.
6	Add a `XAStoreConfiguration` object to make the cache XA transactional. You must also give the cache a unique XAResource identifier as some transaction managers require this.
7	Begin a JTA transaction the normal way.
8	Update the value.
9	Commit the JTA transaction. Other transactions can now see the changes you made to the cache and the SoR.

Configuring it with XML

You can create a XML file to configure a `CacheManager`, lookup a specific transaction manager and configure XA caches:

```
<service>
  <tx:jta-tm transaction-manager-lookup-class=
  "org.ehcache.transactions.xa.txmgr.btm.BitronixTransactionManagerLookup"/>
                                                    <!-- 1 -->
</service>
<cache alias="xaCache">                             <!-- 2 -->
  <key-type>java.lang.String</key-type>
  <value-type>java.lang.String</value-type>
  <heap unit="entries">20</heap>
  <tx:xa-store unique-XAResource-id="xaCache" />    <!-- 3 -->
</cache>
```

1	Declare a `TransactionManagerLookup` that will lookup your transaction manager.
2	Configure a `xaCache` cache the normal way.
3	Configure `xaCache` as an XA cache, giving it `xaCache` as its unique XAResource ID.

In order to parse an XML configuration, you can use the `XmlConfiguration` type:

```
BitronixTransactionManager transactionManager =
      TransactionManagerServices.getTransactionManager();   // 1
URL myUrl = this.getClass().getResource("/docs/configs/xa-getting-started.xml");
                                                            // 2
Configuration xmlConfig = new XmlConfiguration(myUrl);      // 3
CacheManager myCacheManager = CacheManagerBuilder.newCacheManager(xmlConfig);
```

```
                                                                  // 4
myCacheManager.init();
myCacheManager.close();
transactionManager.shutdown();
```

1	The Bitronix transaction manager must be started before the cache manager is initialized.
2	Create a URL to your XML file's location.
3	Instantiate a XmlConfiguration passing it the XML file's URL.
4	Using the staticorg.ehcache.config.builders.CacheManagerBuilder.newCacheManager (org.ehcache.config.Configuration) lets you create your CacheManager instance using the Configuration from the XmlConfiguration.

And here is what the `BitronixTransactionManagerLookup` implementation looks like:

```
public class BitronixTransactionManagerLookup
    implements TransactionManagerLookup {                              // 1
  private static final Logger LOGGER = LoggerFactory.getLogger(
      BitronixTransactionManagerLookup.class);
  @Override
  public TransactionManagerWrapper lookupTransactionManagerWrapper() {   // 2
    if (!TransactionManagerServices.isTransactionManagerRunning()) {     // 3
      throw new IllegalStateException("BTM must be started beforehand");
    }
    TransactionManagerWrapper tmWrapper = new TransactionManagerWrapper(
        TransactionManagerServices.getTransactionManager(),
        new BitronixXAResourceRegistry());                             // 4
    LOGGER.info("Using looked up transaction manager : {}", tmWrapper);
    return tmWrapper;
  }
}
```

1	The `TransactionManagerLookup` interface must be implemented and the offer a no-arg constructor.
2	The `lookupTransactionManagerWrapper()` method must return a `TransactionManagerWrapper` instance.
3	Here is the check that makes sure BTM is started.
4	The `TransactionManagerWrapper` class is constructed with both the `javax.transaction.TransactionManager` instance as well as a `XAResourceRegistryinstance`. The latter is used to register the javax.transaction.xa.XAResource instances of the cache with the transaction manager using an implementation-specific mechanism.

If your JTA implementation doesn't require that, you can use the `NullXAResourceRegistryinstead`.

9 Tiering Options

Ehcache supports the concept of tiered caching. This section covers the different available configuration options. It also explains rules and best practices to benefit the most from tiered caching.

For a general overview of storage tiers, see the section *Storage Tiers* in the *About Terracotta Ehcache* guide.

Moving out of heap

The moment you have a tier other than the heap tier in a cache, a few things happen:

- Adding a mapping to the cache means that the key and value have to be serialized.
- Reading a mapping from the cache means that the key and value may have to be deserialized.

With these two points above, you need to realize that the binary representation of the data and how it is transformed to and from serialized data will play a significant role in caching performance. Make sure you know about the options available for serializers (see the section "Serializers" on page 80). Also this means that some configurations, while making sense on paper, may not offer the best performance depending on the real use case of the application.

Single tier setups

All tiering options can be used in isolation. For example, you can have caches with data only in *offheap* or only *clustered*.

The following possibilities are valid configurations:

- heap
- offheap
- disk
- clustered

For this, simply define the single resource in the cache configuration:

```
CacheConfigurationBuilder.newCacheConfigurationBuilder(
  Long.class, String.class,                        // 1
  ResourcePoolsBuilder.newResourcePoolsBuilder()
    .offheap(2, MemoryUnit.GB)).build();           // 2
```

1	Start with defining the key and value type in the configuration builder.
2	Then specify the resource (the tier) you want to use. Here we use off-heap only.

Heap Tier

The starting point of every cache and also the faster since no serialization is necessary. You can optionally use copiers (see the section "Serializers and Copiers" on page 79) to pass keys and values by-value, the default being by-reference.

A heap tier can be sized by entries or by size.

```
ResourcePoolsBuilder.newResourcePoolsBuilder()
    .heap(10, EntryUnit.ENTRIES);               // 1
// or
ResourcePoolsBuilder.newResourcePoolsBuilder()
    .heap(10);                                  // 2
// or
ResourcePoolsBuilder.newResourcePoolsBuilder()
    .heap(10, MemoryUnit.MB);                   // 3
```

1	Only 10 entries are allowed on heap. Eviction will occur when full.
2	A shortcut to specify 10 entries.
3	Only 10 MB are allowed. Eviction will occur when full.

Byte-sized heap

For every tier except the heap tier, calculating the size of the cache is fairly easy. You more or less sum the size of all byte buffers containing the serialized entries.

When heap is limited by size instead of entries, it is a bit more complicated.

Note: Byte sizing has a runtime performance impact that depends on the size and graph complexity of the data cached.

```
CacheConfiguration<Long, String> usesConfiguredInCacheConfig =
  CacheConfigurationBuilder.newCacheConfigurationBuilder(
    Long.class, String.class,
    ResourcePoolsBuilder.newResourcePoolsBuilder()
    .heap(10, MemoryUnit.KB)                                    // 1
    .offheap(10, MemoryUnit.MB))                                // 2
  .withSizeOfMaxObjectGraph(1000)
  .withSizeOfMaxObjectSize(1000, MemoryUnit.B)                  // 3
  .build();
CacheConfiguration<Long, String> usesDefaultSizeOfEngineConfig =
  CacheConfigurationBuilder.newCacheConfigurationBuilder(
    Long.class, String.class,
    ResourcePoolsBuilder.newResourcePoolsBuilder()
    .heap(10, MemoryUnit.KB))
  .build();
CacheManager cacheManager =
  CacheManagerBuilder.newCacheManagerBuilder()
  .withDefaultSizeOfMaxObjectSize(500, MemoryUnit.B)
  .withDefaultSizeOfMaxObjectGraph(2000)                        // 4
  .withCache("usesConfiguredInCache", usesConfiguredInCacheConfig)
  .withCache("usesDefaultSizeOfEngine",
     usesDefaultSizeOfEngineConfig)
  .build(true);
```

1	This will limit the amount of memory used by the heap tier for storing key-value pairs. There is a cost associated with sizing objects.
2	The settings are only used by the heap tier. So off-heap won't use it at all.
3	The sizing can also be further restrained by 2 additional configuration settings: The first one specifies the maximum number of objects to traverse while walking the object graph (default: `1000`), the second defines the maximum size of a single object (default: `Long.MAX_VALUE`, so almost infinite). If the sizing goes above any of these two limits, the entry won't be stored in cache.
4	A default configuration can be provided at `CacheManager` level to be used by the caches unless defined explicitly.

Off-heap Tier

If you wish to use off-heap, you'll have to define a resource pool, giving the memory size you want to allocate.

```
ResourcePoolsBuilder.newResourcePoolsBuilder()
    .offheap(10, MemoryUnit.MB);                    // 1
```

1	Only 10 MB are allowed off-heap. Eviction will occur when full.

The example above allocates a very small amount of off-heap. You will normally use a much bigger space.

Remember that data stored off-heap will have to be serialized and deserialized - and is thus slower than heap.

You should thus favor off-heap for large amounts of data where on-heap would have too severe an impact on garbage collection.

Do not forget to define in the Java options the `-XX:MaxDirectMemorySize` option, according to the off-heap size you intend to use.

Disk Tier

For the Disk tier, the data is stored on disk. The faster and more dedicated the disk is, the faster accessing the data will be.

```
PersistentCacheManager persistentCacheManager =
    CacheManagerBuilder.newCacheManagerBuilder()       // <1>
    .with(CacheManagerBuilder.persistence(new File(
        getStoragePath(), "myData")))                  // <2>
    .withCache("persistent-cache",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(
        Long.class, String.class,
```

```
        ResourcePoolsBuilder.newResourcePoolsBuilder()
        .disk(10, MemoryUnit.MB, true))                // <3>
      )
      .build(true);
    persistentCacheManager.close();
```

1	Obtain a `PersistentCacheManager`, which is a normal `CacheManager` but with the ability to destroy caches. See the section "Destroying Persistent Tiers" on page 66 for further information.
2	Provide a location where data should be stored.
3	Define a resource pool for the disk that will be used by the cache. The third parameter is a boolean value which is used to set whether the disk pool is persistent. When set to true, the pool is persistent. When the version with 2 parameters `disk(long, MemoryUnit)` is used, the pool is not persistent.

The example above allocates a very small amount of disk storage. You will normally use a much bigger storage.

Persistence means the cache will survive a JVM restart. Everything that was in the cache will still be there after restarting the JVM and creating a `CacheManager` disk persistence at the same location.

Note: A disk tier can't be shared between cache managers. A persistence directory is dedicated to one cache manager at the time.

Remember that data stored on disk will have to be serialized / deserialized and written to / read from disk - and is thus slower than heap and offheap. So disk storage is interesting if:

- You have a large amount of data that can't fit off-heap
- Your disk is much faster than the storage it is caching
- You are interested in persistence

Note: The open source disk tier offers no data integrity guarantee in the case of a crash. There is an enterprise version that provides this and more, see below.

Segments

Disk storage is separated into segments which provide concurrency access but also hold open file pointers. The default is 16. In some cases, you might want to reduce the concurrency and save resources by reducing the number of segments.

```
String storagePath = getStoragePath();
PersistentCacheManager persistentCacheManager =
  CacheManagerBuilder.newCacheManagerBuilder()
  .with(CacheManagerBuilder.persistence(new File(storagePath, "myData")))
  .withCache("less-segments",
    CacheConfigurationBuilder.newCacheConfigurationBuilder(
      Long.class, String.class,
```

```
      ResourcePoolsBuilder.newResourcePoolsBuilder()
        .disk(10, MemoryUnit.MB))
    .add(new OffHeapDiskStoreConfiguration(2))    // 1
  )
  .build(true);
persistentCacheManager.close();
```

1	Define an OffHeapDiskStoreConfiguration instance specifying the required number of segments.

Clustered

A clustered tier means the client connects to the Terracotta Server Array where the cached data is stored. It is also as way to have a shared cache between JVMs.

See the section "Clustered Caches" on page 117 for details of using the cluster tier.

Multiple tier setup

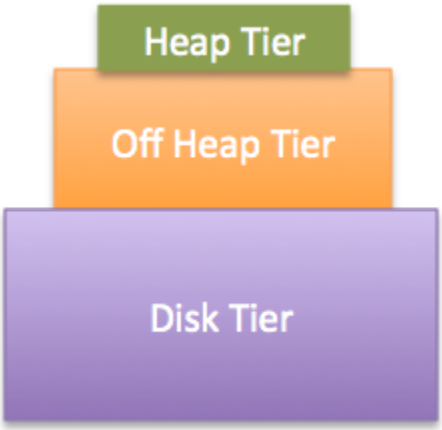
If you want to use more than one tier, you have to observe some constraints:

1. There must always be a heap tier in a multi-tier setup.
2. You cannot combine disk tiers and clustered tiers.
3. Tiers should be sized in a pyramidal fashion, i.e. tiers higher up the pyramid are configured to use less memory than tiers lower down.

For 1, this is a limitation of the current implementation.

For 2, this restriction is necessary, because having two tiers with content that can outlive the life of a single JVM can lead to consistency questions on restart.

For 3, the idea is that tiers are related to each other. The fastest tier (the heap tier) is on top, while the slower tiers are below. In general, heap is more constrained than the total memory of the machine, and offheap memory is more constrained than disk or the memory available on the cluster. This leads to the typical pyramid shape for a multi-tiered setup.



Ehcache requires the size of the heap tier to be smaller than the size of the offheap tier, and the size of the offheap tier to be smaller than the size of the disk tier. While Ehcache cannot verify at configuration time that a count-based sizing for the heap tier will be smaller than a byte-based sizing for another tier, you should make sure that is the case during testing.

Taking the above into account, the following possibilities are valid configurations:

- heap + offheap
- heap + offheap + disk
- heap + offheap + clustered
- heap + disk
- heap + clustered

Here is an example using heap, offheap and clustered.

```
PersistentCacheManager persistentCacheManager =
  CacheManagerBuilder.newCacheManagerBuilder()
  .with(cluster(CLUSTER_URI).autoCreate())                        // 1
  .withCache("threeTierCache",
    CacheConfigurationBuilder.newCacheConfigurationBuilder(
      Long.class, String.class,
      ResourcePoolsBuilder.newResourcePoolsBuilder()
        .heap(10, EntryUnit.ENTRIES)                              // 2
        .offheap(1, MemoryUnit.MB)                                // 3
        .with(ClusteredResourcePoolBuilder.clusteredDedicated(
            "primary-server-resource", 2, MemoryUnit.MB))         // 4
    )
  ).build(true);
```

1	Cluster-specific information telling how to connect to the Terracotta cluster
2	Define the Heap tier, which is the smallest but fastest caching tier.
3	Define the Offheap tier. Next in line as caching tier.
4	Define the Clustered tier. The authoritative tier for this cache

Resource Pools

Tiers are configured using resource pools. Most of the time using a `ResourcePoolsBuilder`. Let's revisit an example used earlier:

```
PersistentCacheManager persistentCacheManager =
  CacheManagerBuilder.newCacheManagerBuilder()
  .with(CacheManagerBuilder.persistence(
    new File(getStoragePath(), "myData")))
  .withCache("threeTieredCache",
    CacheConfigurationBuilder.newCacheConfigurationBuilder(
      Long.class, String.class,
      ResourcePoolsBuilder.newResourcePoolsBuilder()
        .heap(10, EntryUnit.ENTRIES)
```

```
        .offheap(1, MemoryUnit.MB)
        .disk(20, MemoryUnit.MB, true)
    )
  ).build(true);
```

This is a cache using 3 tiers (heap, offheap, disk). They are created and chained using the `ResourcePoolsBuilder`. The declaration order doesn't matter (e.g. offheap can be declared before heap) because each tier has a *height*. The higher the height of a tier is, the *closer* the tier will be to the client.

It is really important to understand that a resource pool is only specifying a configuration. It is not an actual pool that can be shared between caches. Consider for instance this code:

```
ResourcePools pool = ResourcePoolsBuilder
    .newResourcePoolsBuilder().heap(10).build();
CacheManager cacheManager =
  CacheManagerBuilder.newCacheManagerBuilder()
  .withCache("test-cache1", CacheConfigurationBuilder
    .newCacheConfigurationBuilder(Integer.class, String.class, pool))
  .withCache("test-cache2", CacheConfigurationBuilder
    .newCacheConfigurationBuilder(Integer.class, String.class, pool))
  .build(true);
```

You will end up with two caches that can contain 10 entries each. Not a shared pool of 10 entries. Pools are never shared between caches. The exception being clustered caches, that can be shared or dedicated.

Updating Resource Pools

Limited size adjustment can be performed on a live cache.

Note: `updateResourcePools()` only allows you to change the heap tier sizing, not the pool type. Thus you can't change the sizing of off-heap or disk tiers.

```
ResourcePools pools = ResourcePoolsBuilder
  .newResourcePoolsBuilder().heap(20L, EntryUnit.ENTRIES).build();  // 1
cache.getRuntimeConfiguration().updateResourcePools(pools);         // 2
assertThat(cache.getRuntimeConfiguration().getResourcePools()
  .getPoolForResource(ResourceType.Core.HEAP).getSize(), is(20L));
```

1	You will need to create a new `ResourcePools` object with resources of the required size, using `ResourcePoolsBuilder`. This object can then be passed to the said method so as to trigger the update.
2	To update the capacity of `ResourcePools`, the `updateResourcePools(ResourcePools)` method in `RuntimeConfiguration` can be of help. The `ResourcePools` object created earlier can then be passed to this method so as to trigger the update.

Destroying Persistent Tiers

The disk tier and cluster tier are the two persistent tiers. This means that when the JVM is stopped, all the created caches and their data still exist on disk or on the cluster.

Once in a while, you might want to fully remove them. Their definition as `PersistentCacheManager` gives access to the following methods:

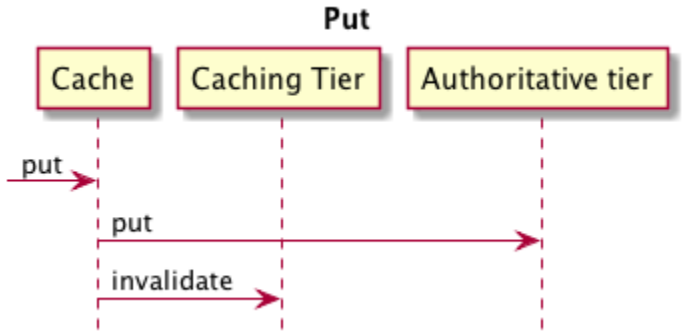
destroy()
This method destroys everything related to the cache manager (including caches, of course). The cache manager must be closed or uninitialized to call this method. Also, for a cluster tier, no other cache manager should currently be connected to the same cache manager server entity.

destroyCache(String cacheName)
Ths method destroys a given cache. The cache shouldn't be in use by another cache manager.

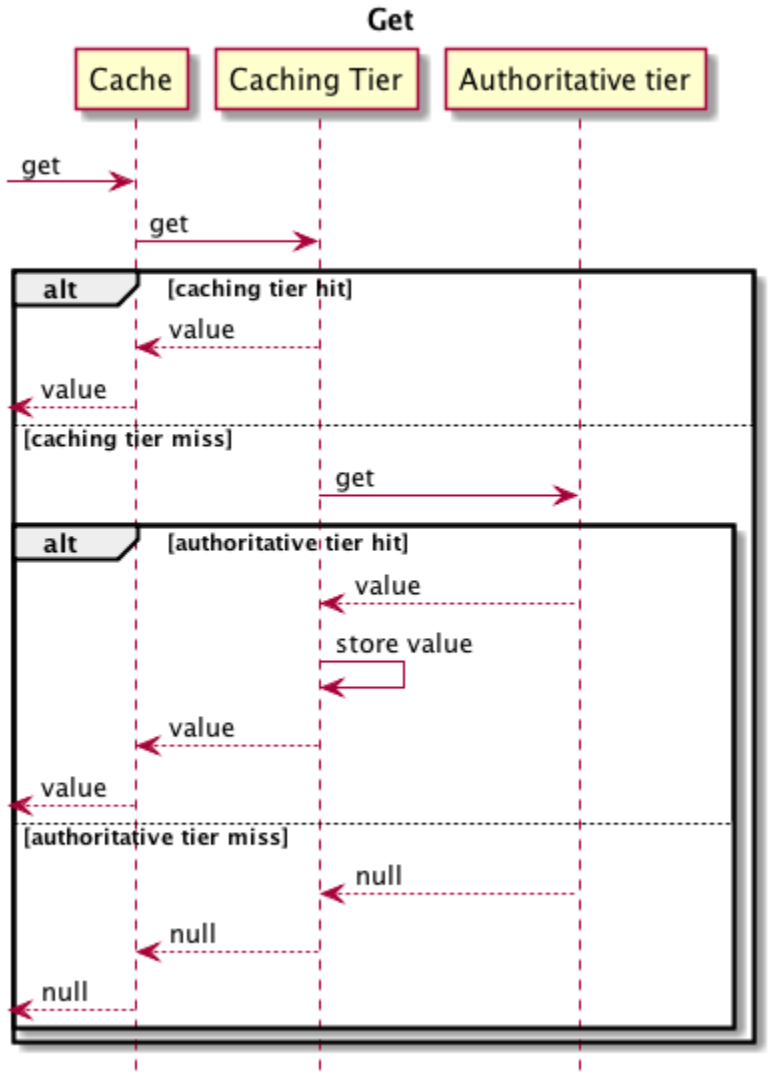
Sequence Flow for Cache Operations with Multiple Tiers

In order to understand what happens for different cache operations when using multiple tiers, here are examples of *Put* and *Get* operations. The sequence diagrams are oversimplified but still show the main points.

Multiple tiers using Put



Multiple tiers using Get



You should then notice the following:

- When putting a value into the cache, it goes straight to the authoritative tier, which is the lowest tier.
- A following `get` will push the value upwards in the caching tiers.
- Of course, as soon as a value is put in the authoritative tier, all higher-level caching tiers are invalidated.
- A full cache miss (the value isn't on any tier) will always go all the way down to the authoritative tier.

Note: The slower your authoritative tier, the slower your `put` operations will be. For a normal cache usage, it usually doesn't matter since`get` operations are much

more frequent than `put` operations. The opposite would mean you probably shouldn't be using a cache in the first place.

10 Cache Loaders and Writers

Introduction to Cache Loaders and Writers

Note: Ehcache clustering is not yet compatible with *cache-through*.

This section documents the specifics behind the cache-through implementation in Ehcache. Refer to the section "Cache Usage Patterns" on page 41 if you are not familiar with terms like *cache-through*, *read-through*, *write-through* or *system of record*.

Ehcache merges the concepts of read-through and write-through behind a single interface, the `CacheLoaderWriter`.

As indicated by its API, this interface provides methods with logical grouping:

read-through
The `load(K)` and `loadAll(Iterable<? super K>)` methods cover the *read-through* part of cache-through.

write-through
The `write(K, V)`,`writeAll(Iterable<? extends Map.Entry<? extends K, ? extends V>>)`,`delete(K)` and `deleteAll(Iterable<? super K>)` methods cover the *write-through* part of cache-through.

The reasoning behind having a unified interface is that if you want a *read-through* only cache, you need to decide what to do about mutative method calls. What happens if someone calls `put(K, V)` on the cache? This risks making it inconsistent with the underlying system of record.

In this context, the unified interface forces you to make a choice: either no-op `write*` / `delete*` methods or throwing when mutation happens.

For a *write-through* only cache, it remains possible by simply having no-op `load*` methods.

Write-behind

An additional feature provided by Ehcache is *write-behind*, where writes are made asynchronously to the backing system of record. The way this works in Ehcache is by simply telling the system to register a wrapper around your provided `CacheLoaderWriter` implementation.

From there, you will have extra configuration options around batching and coalescing of writes.

Ehcache does not support retry of failed writes at the write-behind wrapper level. You, as the application developer and system of record owner, know better when a retry should happen and how. So if you need that functionality, make it part of your `CacheLoaderWriter` implementation.

Write-behind introduces the following concepts:

queue size
Indicates how many pending write operations there can be before applying back pressure on cache operations.

concurrency level
Indicates how many parallel processing threads and queues there will be for write behind. Effectively the maximum number of in-flight writes is "concurrency level * queue size".

batching and batch size
Mutative operations will be grouped in *batch size* sets before reaching the `CacheLoaderWriter`. When batching, the queue size is effectively the number of pending batches there can be. This means that the maximum number of in-flight writes becomes "concurrency level * queue size * batch size".

coalescing
When batching, coalescing means that you only send the latest mutation on a per key basis to the `CacheLoaderWriter`.

maximum write delay
When batching, you can indicate the maximum write delay for an incomplete batch. After this time has elapsed, the batch is processed even if incomplete.

Implementing Cache-Through

```
CacheManager cacheManager =
    CacheManagerBuilder.newCacheManagerBuilder().build(true);
Cache<Long, String> writeThroughCache =
    cacheManager.createCache("writeThroughCache",
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
         String.class, ResourcePoolsBuilder.heap(10))
        .withLoaderWriter(new SampleLoaderWriter<Long,
             String>(singletonMap(41L, "zero")))  // <1>
        .build());
assertThat(writeThroughCache.get(41L), is("zero")); // <2>
writeThroughCache.put(42L, "one"); // <3>
assertThat(writeThroughCache.get(42L), equalTo("one"));
cacheManager.close();
```

1	We register a sample `CacheLoaderWriter` that knows about the mapping ("41L" maps to "zero") .
2	Since the cache has no content yet, this will delegate to the `CacheLoaderWriter`. The returned mapping will populate the cache and be returned to the caller.
3	While creating this cache mapping, the `CacheLoaderWriter` will be invoked to write the mapping into the system of record.

Adding Write-Behind

```
CacheManager cacheManager =
    CacheManagerBuilder.newCacheManagerBuilder().build(true);
Cache<Long, String> writeBehindCache =
    cacheManager.createCache("writeBehindCache",
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
        String.class, ResourcePoolsBuilder.heap(10))
        .withLoaderWriter(new SampleLoaderWriter<Long,
            String>(singletonMap(41L, "zero")))  // <1>
        .add(WriteBehindConfigurationBuilder     // <2>
            .newBatchedWriteBehindConfiguration(1, TimeUnit.SECONDS, 3) // <3>
            .queueSize(3)        // <4>
            .concurrencyLevel(1) // <5>
            .enableCoalescing()) // <6>
        .build());
assertThat(writeBehindCache.get(41L), is("zero"));
writeBehindCache.put(42L, "one");
writeBehindCache.put(43L, "two");
writeBehindCache.put(42L, "This goes for the record");
assertThat(writeBehindCache.get(42L), equalTo("This goes for the record"));
cacheManager.close();
```

1	For write-behind you need a configured `CacheLoaderWriter`.
2	Additionally, register a `WriteBehindConfiguration` on the cache by using the `WriteBehindConfigurationBuilder`.
3	Here we configure write behind or batching with a batch size of 3 and a maximum write delay of 1 second.
4	We also set the maximum size of the write-behind queue.
5	Define the concurrency level of write-behind queue(s). This indicates how many writer threads work in parallel to update the underlying system of record asynchronously.
6	Enable the write coalescing behavior, which ensures that only one update per key per batch reaches the underlying system of record.

11 Cache Event Listeners

Introduction

Cache listeners allow implementers to register callback methods that will be executed when a cache event occurs.

Listeners are registered at the cache level - and therefore only receive events for caches that they have been registered with.

```
CacheEventListenerConfigurationBuilder cacheEventListenerConfiguration =
    CacheEventListenerConfigurationBuilder
    .newEventListenerConfiguration(new ListenerObject(), EventType.CREATED,
        EventType.UPDATED)                                  // 1
    .unordered().asynchronous();                            // 2
final CacheManager manager = CacheManagerBuilder.newCacheManagerBuilder()
    .withCache("foo",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(String.class,
             String.class, ResourcePoolsBuilder.heap(10))
            .add(cacheEventListenerConfiguration)           // 3
    ).build(true);
final Cache<String, String> cache = manager.getCache("foo", String.class, String.class);
cache.put("Hello", "World");                                // 4
cache.put("Hello", "Everyone");                             // 5
cache.remove("Hello");                                      // 6
```

1	Create a `CacheEventListenerConfiguration` using the builder indicating the listener and the events to receive (in this case create and update events)
2	Optionally indicate the delivery mode - defaults are *asynchronous* and *unordered* (for performance reasons)
3	Pass the configuration of the listener into the cache configuration
4	You will be notified on creation
5	And on update
6	But not on removal, because it wasn't included at step 1

Created, updated, and removed events are triggered by user execution of mutative methods as outlined in the table below. Eviction and expiration events can be triggered by both internal processes and by user execution of methods targeting both related and unrelated keys within the cache.

Table 1. Cache entry event firing behaviors for mutative methods

input	operation	output	event {key, old-value, new-value}
{}	put(K, V)	{K, V}	created {K, null, V}
{K, V1}	put(K, V2)	{K, V2}	updated {K, V1, V2}
{}	put(K, V) [immediately expired]	{}	none
{K, V1}	put(K, V2) [immediately expired]	{}	none
{}	putIfAbsent(K, V)	{K, V}	created {K, null, V}
{}	putIfAbsent(K, V) [immediately expired]	{}	none
{K, V1}	replace(K, V2)	{K, V2}	updated {K, V1, V2}
{K, V1}	replace(K, V2) [immediately expired]	{}	none
{K, V1}	replace(K, V1, V2)	{K, V2}	updated {K, V1, V2}
{K, V1}	replace(K, V1, V2) [immediately expired]	{}	no events
{K, V}	remove(K)	{}	removed {K, V, null}

Note: Ehcache provides an abstract class `CacheEventAdapter` for convenient implementation of event listeners when you are interested only on specific events.

Registering Event Listeners during runtime

Cache event listeners may also be added and removed while the cache is being used.

```
ListenerObject listener = new ListenerObject();                    // 1
cache.getRuntimeConfiguration().registerCacheEventListener(listener,
    EventOrdering.ORDERED,
    EventFiring.ASYNCHRONOUS, EnumSet.of(EventType.CREATED,
        EventType.REMOVED));                                                  // 2
cache.put(1L, "one");
cache.put(2L, "two");
cache.remove(1L);
cache.remove(2L);
cache.getRuntimeConfiguration().deregisterCacheEventListener(listener);    // 3
cache.put(1L, "one again");
cache.remove(1L);
```

1	Create a `CacheEventListener` implementation instance.
2	Register it on the `RuntimeConfiguration`, indicating the delivery mode and events of interest. The following `put()` and `remove()` cache calls will make the listener receive events.
3	Unregister the previously registered `CacheEventListener` instance. The following `put()` and `remove()` cache calls will have no effect on the listener anymore.

Event Processing Queues

Advanced users may want to tune the level of concurrency which may be used for delivery of events.

```
CacheConfiguration<Long, String> cacheConfiguration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
        String.class, ResourcePoolsBuilder.heap(5L))
    .withDispatcherConcurrency(10)                          // 1
    .withEventListenersThreadPool("listeners-pool")
    .build();
```

1	Indicate the level of concurrency desired

This will enable parallel processing of events at the cost of more threads being required by the system.

12 Eviction Advisors

Note: This is an advanced topic/feature that will not be of interest to most users.

You can affect which elements are selected for eviction from the cache by providing a class that implements the `org.ehcache.config.EvictionAdvisor` interface.

Note: Eviction advisors are not used for clustered storage tiers. For example, in a cache with a heap tier and clustered storage tier, the heap tier will use the eviction advisor but the clustered storage tier will evict independently, irrespective of the eviction advisor. The description below applies to using an eviction advisor for the cache tiers other than a clustered storage tier.

`EvictionAdvisor` implementations are invoked when Ehcache is attempting to evict entries from the cache (in order to make room for new entries) in order to determine whether the given entry should not be considered a good candidate for eviction. If the eviction is advised against, Ehcache will try to honor the preference of preserving that entry in the cache, though there is no full guarantee of such.

```
CacheConfiguration<Long, String> cacheConfiguration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
    String.class,
    ResourcePoolsBuilder.heap(2L))   // 1
    .withEvictionAdvisor(new OddKeysEvictionAdvisor<Long, String>()) // 2
    .build();
CacheManager cacheManager = CacheManagerBuilder.newCacheManagerBuilder()
    .withCache("cache", cacheConfiguration)
    .build(true);
Cache<Long, String> cache = cacheManager.getCache("cache", Long.class,
    String.class);
// Work with the cache
cache.put(42L, "The Answer!");
cache.put(41L, "The wrong Answer!");
cache.put(39L, "The other wrong Answer!");
cacheManager.close();
```

1	Configure a constrained heap, as the eviction advisor is only relevant when mappings get evicted from the cache.
2	If you want to give the eviction algorithm a hint to advise against the eviction of some mappings, you have to configure an instance of `EvictionAdvisor`.

In this particular example, the `OddKeysEvictionAdvisor` class will advise against eviction of any key that is an odd number. The cache is constrained to only be allowed to contain two entries, however the code has put three entries into the cache - which will trigger capacity eviction. By the time the cache manager gets closed, only mappings with

odd keys should be left in the cache as their prime candidacy for eviction would have been advised against.

Note:

1. The eviction advisor may only be invoked when a mapping is written to the cache. This means that proper eviction advisor implementations are expected to be constant for a key-value pair.
2. Please keep in mind that configuring an eviction advisor can slow down eviction: the more often you advise against eviction, the harder the cache has to work to evict an element when room is required. After a certain time, if a cache determines that the configured eviction advisor rejected too many eviction candidates, the cache can decide to completely bypass the eviction advisor and evict anything it sees fit.

13 Serializers and Copiers

Overview of Serializers and Copiers

While Ehcache is a Java cache, it cannot always store its mappings as Java objects.

The on-heap store is capable of storing cached objects either by reference (where the given key and value references are stored) or by value (where a copy of the given key and value are made and those copies are then stored). All other stores are only capable of storing a byte representation of the key/value pair. See the section "Heap Tier" on page 60 for more details.

`Serializer` and `Copier` are the abstractions to enable these different storage options.

Serializers

All stores but the on-heap one need some form of serialization/deserialization of objects to be able to store and retrieve mappings. This is because they cannot internally store plain java objects but only binary representations of them.

`Serializer` is the Ehcache abstraction solving this: every cache that has at least one store that cannot store by reference is going to use a pair of `Serializer` instances, one for the key and another one for the value.

A `Serializer` is scoped at the cache level and all stores of a cache will be using and sharing the same pair of serializers.

How is a serializer configured?

There are two places where serializers can be configured:

- at the cache level where one can use
  - `CacheConfigurationBuilder.withKeySerializer(Class<? extends Serializer<K>> keySerializerClass),`
  - `CacheConfigurationBuilder.withKeySerializer(Serializer<K> keySerializer),`
  - `CacheConfigurationBuilder.withValueSerializer(Class<? extends Serializer<V>> valueSerializerClass),`
  - and `CacheConfigurationBuilder.withValueSerializer(Serializer<V> valueSerializer),`

    which allow by instance or by class configuration.
- at the cache manager level where one can use
  - `CacheManagerBuilder.withSerializer(Class<C> clazz, Class<? extends Serializer<C>> serializer)`

If a serializer is configured directly at the cache level, it will be used, ignoring any cache manager level configuration.

If a serializer is configured at the cache manager level, upon initialization, a cache with no specifically configured serializer will search through its cache manager's registered list of serializers and try to find one that directly matches the cache's key or value type. If such search fails, all the registered serializers will be tried in the added order to find one that handles compatible types.

For instance, let's say you have a `Person` interface and two subclasses: `Employee` and `Customer`. If you configure your cache manager as follows:

```
CacheManagerBuilder.newCacheManagerBuilder().withSerializer(Employee.class,
EmployeeSerializer.class).withSerializer(Person.class, PersonSerializer.class)
```

then configuring a `Cache<Long, Employee>` would make it use the `EmployeeSerializer` while a `Cache<Long, Customer>` would make it use the `PersonSerializer`.

A `Serializer` configured at the cache level by class will not be shared to other caches when instantiated.

Note: Given the above, it is recommended to limit `Serializer` registration to concrete classes and not aim for generality.

Bundled implementations

By default, cache managers are pre-configured with specially optimized `Serializer` that can handle the following types, in the following order:

- `java.io.Serializable`
- `java.lang.Long`
- `java.lang.Integer`
- `java.lang.Float`
- `java.lang.Double`
- `java.lang.Character`
- `java.lang.String`
- `byte[]`

All bundled `Serializer` implementations support both persistent and transient caches.

Note: A consequence of providing serializers registered by default is that you will not be able to register a generic `Serializer` for `Number` or any other super type and expect it to be picked instead of the default ones for the types listed above.

However, registering a different `Serializer` for one of the given type means it will be used instead of the default.

Lifecycle: instances vs. class

When a `Serializer` is configured by providing an *instance*, it is up to the provider of that instance to manage its lifecycle. It will need to dispose of any resource the serializer might hold, persisting or reloading the serializer's state.

When a `Serializer` is configured by providing a *class* either at the cache or cache manager level, since Ehcache is responsible for creating the instance, it also is responsible for disposing of it. If the `Serializer` implements `java.io.Closeable` then `close()` will be called when the cache is closed and the `Serializer` no longer needed.

Writing your own serializer

`Serializer` defines a very strict contract. So if you're planning to write your own implementation you have to keep in mind that the class of the serialized object MUST be retained after deserialization, that is:

```
object.getClass().equals(
    mySerializer.read(mySerializer.serialize(object)).getClass() )
```

This is especially important when you are planning to write a serializer for an abstract type, e.g. a serializer of type `com.pany.MyInterface` should

- deserialize a `com.pany.MyClassImplementingMyInterface` when the serialized object is of class `com.pany.MyClassImplementingMyInterface`
- return a `com.pany.AnotherClassImplementingMyInterface` object when the serialized object is of class `com.pany.AnotherClassImplementingMyInterface`

Implement the following interface, from package `org.ehcache.spi.serialization`:

```
/**
 * Defines the contract used to transform type instances to and
 * from a serial form.
 * <P>
 *   Implementations must be thread-safe.
 * </P>
 * <P>
 * When used within the default serialization provider, there are additional
 * requirements.
 * The implementations must define either or both of the two constructors:
 * <dl>
 *   <dt><code><i>Serializer</i>(ClassLoader loader)</code>
 *   <dd>This constructor is used to initialize the serializer for transient caches.
 *   <dt><code><i>Serializer</i>(ClassLoader loader,
 *         org.ehcache.core.spi.service.FileBasedPersistenceContext context)</code>
 *   <dd>This constructor is used to initialize the serializer for persistent caches.
 * </dl>
 * The {@code ClassLoader} value may be {@code null}.  If not {@code null}, the
 * class loader
 * instance provided should be used during deserialization to load classes needed
 * by the deserialized objects.
 * </P>
 * <p>
 *   The serialized object's class must be preserved; deserialization of the serial
 *   form of an object must
 *   return an object of the same class. The following contract must always be true:
 *   <p>
 *   <code>object.getClass().equals( mySerializer.read(mySerializer.serialize(object))
```

```
 *       .getClass() )</code>
 *   </p>
 * </p>
 *
 * @param <T> the type of the instances to serialize
 *
 * @see SerializationProvider
 */
public interface Serializer<T> {
  /**
   * Transforms the given instance into its serial form.
   *
   * @param object the instance to serialize
   *
   * @return the binary representation of the serial form
   *
   * @throws SerializerException if serialization fails
   */
  ByteBuffer serialize(T object) throws SerializerException;
  /**
   * Reconstructs an instance from the given serial form.
   *
   * @param binary the binary representation of the serial form
   *
   * @return the de-serialized instance
   *
   * @throws SerializerException if reading the byte buffer fails
   * @throws ClassNotFoundException if the type to de-serialize to cannot be found
   */
  T read(ByteBuffer binary) throws ClassNotFoundException, SerializerException;
  /**
   * Checks if the given instance and serial form {@link Object#equals(Object)
   * represent} the same instance.
   *
   * @param object the instance to check
   * @param binary the serial form to check
   *
   * @return {@code true} if both parameters represent equal instances,
   *  {@code false} otherwise
   *
   * @throws SerializerException if reading the byte buffer fails
   * @throws ClassNotFoundException if the type to de-serialize to cannot be found
   */
  boolean equals(T object, ByteBuffer binary) throws ClassNotFoundException,
      SerializerException;
}
```

As the Javadoc states, there are some constructor rules, see the section "Persistent vs. transient caches" on page 84 for that.

You can optionally implement `java.io.Closeable`. If you do, Ehcache will call `close()` when a cache using such a serializer gets disposed of, but **only if**Ehcache instantiated the serializer itself.

ClassLoaders

When Ehcache instantiates a serializer itself, it will pass it a `ClassLoader` via the constructor. Such class loader must be used to access the classes of the serialized types as they might not be available in the current class loader

Persistent vs. transient caches

All custom serializers must have a constructor with the following signature:

```
public MySerializer(ClassLoader classLoader) {
}
```

Attempting to configure a serializer that lacks such a constructor on a cache using either of `CacheConfigurationBuilder.withKeySerializer(Class<? extends Serializer<K>> keySerializerClass)` or `CacheConfigurationBuilder.withValueSerializer(Class<? extends Serializer<V>> valueSerializerClass)` will cause an exception upon cache initialization.

But if an instance of the serializer is configured using either of `CacheConfigurationBuilder.withKeySerializer(Serializer keySerializer)` or `CacheConfigurationBuilder.withValueSerializer(Serializer valueSerializer)` it will work since the instantiation is done by the user code itself.

Registering a serializer that lacks such a constructor at the cache manager level will prevent it from being chosen for caches.

Custom serializer implementations could have some state that is used in the serialization/deserialization process. When configured on a persistent cache, the state of such serializers needs to be persisted across restarts.

To address these requirements you can have a `StatefulSerializer` implementation. `StatefulSerializer` is a specialized `Serializer` with an additional `init` method with the following signature:

```
public void init(StateRepository repository) {
}
```

The `StateRepository.getPersistentStateHolder(String name, Class<K> keyClass, Class<V> valueClass, Predicate<Class<?>> isClassPermitted, ClassLoader classLoader)` provides a `StateHolder` (a map like structure) that you can use to store any relevant state. Here `name` is the name of the `StateHolder` which maps objects of `keyClass` to objects of `valueClass`. The `Predicate isClassPermitted` authorizes the classes for deserialization as part of key or value deserialization. If a `Class` fails the `isClassPermitted` test, a `RuntimeException` is thrown. The deserialization uses the `ClassLoader` to resolve classes.

Note: `StateRepository.getPersistentStateHolder(String name, Class<K> keyClass, Class<V> valueClass)` has been deprecated in favour of the above method which takes in `isClassPermitted` and `classLoader` also as parameters.

The `StateRepository` is provided by the authoritative tier of the cache and hence will have the same persistence properties of that tier. For persistent caches it is highly recommended that all state is stored in these holders as the users won't have to worry about the persistence aspects of this state holder as it is taken care of by Ehcache.

- In the case of a disk persistent cache, the contents of the state holder will be persisted locally on to the disk.
- For clustered caches, the contents are persisted in the cluster itself so that other clients using the same cache can also access the contents of the state holder.

Copiers

As the on-heap store is capable of storing plain Java objects as such, it is not necessary to rely on a serialization mechanism to copy keys and values in order to provide *by value* semantics. Other forms of *copy* mechanism can be a lot more performant, such as using a *copy constructor* but it requires custom code to be able to copy user classes.

`Copier` is the Ehcache abstraction solving this: it is specific to the on-heap store.

By default, the on-heap mappings are stored *by reference*. The way to store them *by value* is to configure copier(s) on the cache for the key, value or both.

Of course, the exact semantic of *by value* in this context depends heavily on the `Copier` implementation.

How is a copier configured?

There are two places where copiers can be configured:

- at the cache level where one can use
  - `CacheConfigurationBuilder.withKeyCopier(Class<? extends Copier<K>> keyCopierClass),`
  - `CacheConfigurationBuilder.withKeyCopier(Copier<K> keyCopier),`
  - `CacheConfigurationBuilder.withValueCopier(Class<? extends Copier<V>> valueCopierClass),`
  - and `CacheConfigurationBuilder.withValueCopier(Copier<V> valueCopier).`

    which allow by *instance* or by *class* configuration.
- at the cache manager level where one can use
  - `CacheManagerBuilder.withCopier(Class<C> clazz, Class<? extends Copier<C>> copier)`

If a copier is configured directly at the cache level, it will be used, ignoring any cache manager level configuration.

If a copier is configured at the cache manager level, upon initialization, a cache with no specifically configured copier will search through its cache manager's registered list of copiers and try to find one that directly matches the cache's key or value type. If such search fails, all the registered copiers will be tried in the added order to find one that handles compatible types.

For instance, let's say you have a `Person` interface and two subclasses: `Employee` and `Customer`. If you configure your cache manager as follows:

```
CacheManagerBuilder.newCacheManagerBuilder().withCopier(Employee.class,
EmployeeCopier.class).withCopier(Person.class,
PersonCopier.class)
```

then configuring a `Cache<Long, Employee>` would make it use the `EmployeeCopier` while a `Cache<Long, Customer>` would make it use the `PersonCopier`.

A `Copier` configured at the cache level by class will not be shared to other caches when instantiated.

Note: Given the above, it is recommended to limit `Copier` registration to concrete classes and not aim for generality.

Bundled implementations

A `SerializingCopier` class exists in case you want to configure store *by value* on-heap using the configured (or default) serializer. Note that this implementation performs a serialization / deserialization on each read or write operation.

Add builder methods to section about serializing copier

The `CacheConfigurationBuilder` provides the following methods to make use of this specialized copier:

- `CacheConfigurationBuilder.withKeySerializingCopier()` for the key.
- `CacheConfigurationBuilder.withValueSerializingCopier()` for the value.

Lifecycle: instances vs class

When a `Copier` is configured by providing an *instance*, it is up to the provider of that instance to manage its lifecycle. It will need to dispose of any resource it used after it is no longer required.

When a `Copier` is configured by providing a *class* either at the cache or cache manager level, since Ehcache is responsible for creating the instance, it also is responsible for disposing of it. If the `Copier` implements `java.io.Closeable` then `close()` will be called when the cache is closed and the `Copier` no longer needed.

Writing your own Copier

Implement the following interface:

```
/**
 * Defines the contract used to copy type instances.
 * <p>
 *   The copied object's class must be preserved.  The following must always be true:
 *   <p>
 *   <code>object.getClass().equals( myCopier.copyForRead(object).getClass() )</code>
 *   <code>object.getClass().equals( myCopier.copyForWrite(object).getClass() )</code>
 *   </p>
 * </p>
 * @param <T> the type of the instance to copy
 */
```

```
public interface Copier<T> {
  /**
   * Creates a copy of the instance passed in.
   * <p>
   *   This method is invoked as a value is read from the cache.
   * </p>
   *
   * @param obj the instance to copy
   * @return the copy of the {@code obj} instance
   */
  T copyForRead(T obj);
  /**
   * Creates a copy of the instance passed in.
   * <p>
   *   This method is invoked as a value is written to the cache.
   * </p>
   *
   * @param obj the instance to copy
   * @return the copy of the {@code obj} instance
   */
  T copyForWrite(T obj);
}
```

- `T copyForRead(T obj)` is invoked when a copy must be made upon a read operation (like a cache `get()`),
- `T copyForWrite(T obj)` is invoked when a copy must be made upon a write operation (like a cache `put()`).

The separation between copying for read and for write can be useful when you want to store a lighter version of your objects into the cache.

Alternatively, you can extend from org.ehcache.impl.copy.ReadWriteCopier if copying for read and copying for write implementations are identical, in which case you only have to implement:

- `public abstract T copy(T obj)`

14 Thread Pools

Introduction to Thread Pools

Some services work asynchronously, hence they require thread pools to perform their tasks. All thread pooling facilities are centralized behind the `ExecutionService` interface.

Let's start with a bit of theory.

What ExecutionService provides

`ExecutionService` is an interface providing:

- `ScheduledExecutorService` to schedule tasks, i.e.: tasks that happen repeatedly after a configurable delay.
- Unordered `ExecutorService` to execute tasks as soon as a thread is available.
- Ordered `ExecutorService` to execute tasks as soon as a thread is available, with the guarantee that tasks are going to be executed in the order they were submitted.

Available ExecutionService implementations

There currently are two bundled implementations:

- `OnDemandExecutionService` creates a new pool each time an executor service (scheduled or not) is requested. This implementation is the default one and requires no configuration at all.
- `PooledExecutionService` keeps a configurable set of thread pools and divides them to handle all executor service requests. This implementation must be configured with a `PooledExecutionServiceConfiguration` when used.

Configuring PooledExecutionService

When you want total control of the threads used by a cache manager and its caches, you have to use a `PooledExecutionService` that itself must be configured as it does not have any defaults.

The `PooledExecutionServiceConfigurationBuilder` can be used for this purpose, and the resulting configuration it builds can simply be added to a `CacheManagerBuilder` to switch the `ExecutionService` implementation to a `PooledExecutionService`.

The builder has two interesting methods:

- `defaultPool` that is used to set the default pool. There can be only one default pool, its name does not matter, and if thread-using services do not specify a thread pool, this is the one that will be used.
- `pool` that is used to add a thread pool. There can be as many pools as you wish but services must explicitly be configured to make use of them.

Using the configured thread pools

Following is the list of services making use of `ExecutionService`:

- Disk store: disk writes are performed asynchronously.

  `OffHeapDiskStoreConfiguration` is used to configure what thread pool to use at the cache level, while `OffHeapDiskStoreProviderConfiguration` is used to configure what thread pool to use at the cache manager level.

- Write Behind: `CacheLoaderWriter` write tasks happen asynchronously.

  `DefaultWriteBehindConfiguration` is used to configure what thread pool to use at the cache level, while `WriteBehindProviderConfiguration` is used to configure what thread pool to use at the cache manager level.

- Eventing: produced events are queued and sent to the listeners by a thread pool.

  `DefaultCacheEventDispatcherConfiguration` is used to configure what thread pool to use at the cache level, while `CacheEventDispatcherFactoryConfiguration` is used to configure what thread pool to use at the cache manager level.

The different builders will make use of the right configuration class, you do not have to use those classes directly. For instance, calling `CacheManagerBuilder.withDefaultDiskStoreThreadPool(String threadPoolAlias)` actually is identical to calling `CacheManagerBuilder.using(new OffHeapDiskStoreProviderConfiguration(threadPoolAlias))`.

The thread pool to use can be configured on a service through the builders by using the methods carrying a `ThreadPool` related name. When a service is not told anything about which thread pool to use, the default thread pool is used.

Configuring Thread Pools with Code

Following are examples of describing how to configure the thread pools the different services will use.

Disk store

```
CacheManager cacheManager
    = CacheManagerBuilder.newCacheManagerBuilder()
    .using(PooledExecutionServiceConfigurationBuilder
    .newPooledExecutionServiceConfigurationBuilder()    // 1
        .defaultPool("dflt", 0, 10)
        .pool("defaultDiskPool", 1, 3)
        .pool("cache2Pool", 2, 2)
        .build())
    .with(new CacheManagerPersistenceConfiguration(new File(getStoragePath(),
        "myData")))
    .withDefaultDiskStoreThreadPool("defaultDiskPool")  // 2
    .withCache("cache1",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
            String.class,
```

```
            ResourcePoolsBuilder.newResourcePoolsBuilder()
                .heap(10, EntryUnit.ENTRIES)
                .disk(10L, MemoryUnit.MB)))
    .withCache("cache2",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
            String.class,
            ResourcePoolsBuilder.newResourcePoolsBuilder()
                .heap(10, EntryUnit.ENTRIES)
                .disk(10L, MemoryUnit.MB))
            .withDiskStoreThreadPool("cache2Pool", 2))  // 3
    .build(true);
Cache<Long, String> cache1 =
    cacheManager.getCache("cache1", Long.class, String.class);
Cache<Long, String> cache2 =
    cacheManager.getCache("cache2", Long.class, String.class);
cacheManager.close();
```

1	Configure the thread pools. Note that the default one (`dflt`) is required for the events even when no event listener is configured.
2	Tell the `CacheManagerBuilder` to use a default thread pool for all disk stores that don't explicitly specify one.
3	Tell the cache to use a specific thread pool for its disk store.

Write Behind

```
CacheManager cacheManager
    = CacheManagerBuilder.newCacheManagerBuilder()
    .using(PooledExecutionServiceConfigurationBuilder.
        newPooledExecutionServiceConfigurationBuilder()   // 1
        .defaultPool("dflt", 0, 10)
        .pool("defaultWriteBehindPool", 1, 3)
        .pool("cache2Pool", 2, 2)
        .build())
    .withDefaultWriteBehindThreadPool("defaultWriteBehindPool")  // 2
    .withCache("cache1",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
            String.class, ResourcePoolsBuilder.newResourcePoolsBuilder().heap(10,
                EntryUnit.ENTRIES))
            .withLoaderWriter(new SampleLoaderWriter<Long, String>(
                 singletonMap(41L, "zero")))
            .add(WriteBehindConfigurationBuilder
                .newBatchedWriteBehindConfiguration(1, TimeUnit.SECONDS, 3)
                .queueSize(3)
                .concurrencyLevel(1)))
    .withCache("cache2",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
            String.class, ResourcePoolsBuilder.newResourcePoolsBuilder().heap(10,
                EntryUnit.ENTRIES))
            .withLoaderWriter(new SampleLoaderWriter<Long, String>(
                singletonMap(41L, "zero")))
            .add(WriteBehindConfigurationBuilder
                .newBatchedWriteBehindConfiguration(1, TimeUnit.SECONDS, 3)
                .useThreadPool("cache2Pool")  // 3
                .queueSize(3)
                .concurrencyLevel(2)))
    .build(true);
```

```
Cache<Long, String> cache1 =
    cacheManager.getCache("cache1", Long.class, String.class);
Cache<Long, String> cache2 =
    cacheManager.getCache("cache2", Long.class, String.class);
cacheManager.close();
```

1	Configure the thread pools. Note that the default one (`dflt`) is required for the events even when no event listener is configured.
2	Tell the `CacheManagerBuilder` to use a default thread pool for all write-behind caches that don't explicitly specify one.
3	Tell the `WriteBehindConfigurationBuilder` to use a specific thread pool for its write-behind work.

Events

```
CacheManager cacheManager
    = CacheManagerBuilder.newCacheManagerBuilder()
    .using(PooledExecutionServiceConfigurationBuilder
        .newPooledExecutionServiceConfigurationBuilder()   // 1
        .pool("defaultEventPool", 1, 3)
        .pool("cache2Pool", 2, 2)
        .build())
    .withDefaultEventListenersThreadPool("defaultEventPool")    // 2
    .withCache("cache1",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
            String.class, ResourcePoolsBuilder.newResourcePoolsBuilder().heap(10,
            EntryUnit.ENTRIES))
            .add(CacheEventListenerConfigurationBuilder
                .newEventListenerConfiguration(new ListenerObject(),
                 EventType.CREATED, EventType.UPDATED)))
    .withCache("cache2",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
            String.class, ResourcePoolsBuilder.newResourcePoolsBuilder().heap(10,
                EntryUnit.ENTRIES))
            .add(CacheEventListenerConfigurationBuilder
                .newEventListenerConfiguration(new ListenerObject(),
                 EventType.CREATED, EventType.UPDATED))
            .withEventListenersThreadPool("cache2Pool"))   // 3
    .build(true);
Cache<Long, String> cache1 =
    cacheManager.getCache("cache1", Long.class, String.class);
Cache<Long, String> cache2 =
    cacheManager.getCache("cache2", Long.class, String.class);
cacheManager.close();
```

1	Configure the thread pools. Note that there is no default one so all thread-using services must be configured with explicit defaults.
2	Tell the `CacheManagerBuilder` to use a default thread pool to manage events of all caches that don't explicitly specify one.

3	Tell the `CacheEventListenerConfigurationBuilder` to use a specific thread pool for sending its events.

Configuring Thread Pools with XML

Following is an example describing how to configure the thread pools the different services will use.

```
<thread-pools>   <!-- 1 -->
  <thread-pool alias="defaultDiskPool" min-size="1" max-size="3"/>
  <thread-pool alias="defaultWriteBehindPool" min-size="1" max-size="3"/>
  <thread-pool alias="cache2Pool" min-size="2" max-size="2"/>
</thread-pools>
<event-dispatch thread-pool="defaultEventPool"/>        <!-- 2 -->
<write-behind thread-pool="defaultWriteBehindPool"/>    <!-- 3 -->
<disk-store thread-pool="defaultDiskPool"/>             <!-- 4 -->
<cache alias="cache1">
  <key-type>java.lang.Long</key-type>
  <value-type>java.lang.String</value-type>
  <resources>
    <heap unit="entries">10</heap>
    <disk unit="MB">10</disk>
  </resources>
</cache>
<cache alias="cache2">
  <key-type>java.lang.Long</key-type>
  <value-type>java.lang.String</value-type>
  <loader-writer>
    <class>org.ehcache.docs.plugs.ListenerObject</class>
    <write-behind thread-pool="cache2Pool">   <!-- 5 -->
      <batching batch-size="5">
        <max-write-delay unit="seconds">10</max-write-delay>
      </batching>
    </write-behind>
  </loader-writer>
  <listeners dispatcher-thread-pool="cache2Pool"/>  <!-- 6 -->
  <resources>
    <heap unit="entries">10</heap>
    <disk unit="MB">10</disk>
  </resources>
  <disk-store-settings thread-pool="cache2Pool"
       writer-concurrency="2"/>   <!-- 7 -->
</cache>
```

1	Configure the thread pools. Note that there is no default one.
2	Configure the default thread pool this cache manager will use to send events.
3	Configure the default thread pool this cache manager will use for write-behind work.

4	Configure the default thread pool this cache manager will use for disk stores.
5	Configure a specific write-behind thread pool for this cache.
6	Configure a specific thread pool for this cache to send its events.
7	Configure a specific thread pool for this cache's disk store.

15 Code Examples

Peeper - a simple message board

The `demo` directory in the Ehcache sources includes a sample applications with two (2) implementations demonstrating Ehcache use. Implemented as a simple browser-based web service, the sample application, Peeper, displays any messages (peeps) previously entered and accepts new peeps recording the peeps in a database. The peeps database, shared among implementations of the Peeper application, is located at `$HOME/ehcache-demo-peeper.mv.db`. This file may be safely erased while the application is not running. While running, information about the operation of Peeper application (database access, cache access, etc.) is written to the console.

While the sample application may be run, the application is *very* simplistic - the code implementing the sample is the interesting bit. Running the sample application requires the use of Gradle. This sample may be accessed from GitHub by *cloning* the Ehcache git repository:

```
# Create and/or change to a directory to hold the Ehcache git repository clone
git clone https://github.com/ehcache/ehcache3.git
```

Peeper without Caching-00-NoCache

The first sample, located in `demos/00-NoCache`, is a base Peeper application that does *not* use caching. Each peep is stored in the database and all peeps are read from the database to display the Peeper web page. To run this implementation:

```
cd ehcache3/demos/00-NoCache
../../gradlew appStart
```

This builds the necessary components, starts a Jetty web service, and displays the URL of the web server on the console. The URL will be something like `http://localhost:8080/ehcache-demos/00-NoCache/`.

While running, lines like the following are displayed to the console:

```
11:23:53.536 [800523121@qtp-1157162760-2] INFO  o.e.d.p.DataStore - Loading peeps from DB
11:24:03.226 [800523121@qtp-1157162760-2] INFO  o.e.d.p.DataStore - Adding peep into DB
11:24:03.234 [800523121@qtp-1157162760-2] INFO  o.e.d.p.DataStore - Loading peeps from DB
11:24:13.312 [800523121@qtp-1157162760-2] INFO  o.e.d.p.DataStore - Adding peep into DB
11:24:13.317 [800523121@qtp-1157162760-2] INFO  o.e.d.p.DataStore - Loading peeps from DB
11:24:41.238 [800523121@qtp-1157162760-2] INFO  o.e.d.p.DataStore - Loading peeps from DB
11:24:50.896 [800523121@qtp-1157162760-2] INFO  o.e.d.p.DataStore - Adding peep into DB
11:24:50.901 [800523121@qtp-1157162760-2] INFO  o.e.d.p.DataStore - Loading peeps from DB
11:24:56.295 [800523121@qtp-1157162760-2] INFO  o.e.d.p.DataStore - Adding peep into DB
11:24:56.298 [800523121@qtp-1157162760-2] INFO  o.e.d.p.DataStore - Loading peeps from DB
```

Note the absence of indications of interactions with a cache.

Peeper with Cache-aside Caching-01-CacheAside

The second sample, located in `demos/01-CacheAside`, is a version of the Peeper application that makes use of Ehcache. As each peep is being read from the database

(for display in the web page), it is written to an Ehcache instance. If the Peeper web page is refreshed (without adding a new peep) or a new Peeper client connects, the peeps are read from the cache (instead of the database) to form the web page. If a new peep is posted, the cache is cleared. To run this implementation:

```
cd ehcache3/demos/01-CacheAside
../../gradlew appStart
```

This builds the necessary components, starts a Jetty web service, and displays the URL of the web server on the console. The URL will be something like `http://localhost:8080/ehcache-demos/01-CacheAside/`.

While running, lines like the following are displayed to the console:

```
11:26:20.557 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Loading peeps from DB
11:26:20.572 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Filling cache with peeps
11:26:33.422 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Adding peep into DB
11:26:33.428 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Clearing peeps cache
11:26:33.431 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Loading peeps from DB
11:26:33.432 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Filling cache with peeps
11:26:50.025 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Adding peep into DB
11:26:50.027 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Clearing peeps cache
11:26:50.030 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Loading peeps from DB
11:26:50.031 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Filling cache with peeps
11:27:10.742 [139688380@qtp-965028604-0] INFO  o.e.d.p.DataStore - Getting peeps from cache
```

Note the presence of the Filling cache with peeps, Clearing peeps cache, and Getting peeps from cache lines indicating cache interactions.

XML with 107 extension

```
<ehcache:config
    xmlns:ehcache="http://www.ehcache.org/v3"
    xmlns:jcache="http://www.ehcache.org/v3/jsr107">
  <!--
    OPTIONAL
     services to be managed and lifecycled by the CacheManager
  -->
  <ehcache:service>
    <!--
      One element in another namespace, using our JSR-107 extension as an example here
    -->
    <jcache:defaults>
      <jcache:cache name="invoices" template="myDefaultTemplate"/>
    </jcache:defaults>
  </ehcache:service>
  <!--
    OPTIONAL
      A <cache> element defines a cache, identified by the mandatory 'alias' attribute,
      to be managed by the CacheManager
  -->
  <ehcache:cache alias="productCache">
    <!--
      OPTIONAL, defaults to java.lang.Object
        The FQCN of the type of keys K we'll use with the Cache<K, V>
    -->
    <ehcache:key-type copier=
        "org.ehcache.impl.copy.SerializingCopier">java.lang.Long</ehcache:key-type>
    <!--
      OPTIONAL, defaults to java.lang.Object
        The FQCN of the type of values V we'll use with the Cache<K, V>
    -->
```

```
    <ehcache:value-type copier=
        "org.ehcache.impl.copy.SerializingCopier">com.pany.domain.Product</ehcache:value-type>
    <!--
      OPTIONAL, defaults to no expiry
        Entries to the Cache can be made to expire after a given time
    -->
    <ehcache:expiry>
      <!--
        time to idle, the maximum time for an entry to remain untouched
          Entries to the Cache can be made to expire after a given time
          other options are:
             * <ttl>, time to live;
             * <class>, for a custom Expiry implementation; or
             * <none>, for no expiry
      -->
      <ehcache:tti unit="minutes">2</ehcache:tti>
    </ehcache:expiry>
    <!--
      OPTIONAL, defaults to no advice
        An eviction advisor, which lets you control what entries should only get
        evicted as last resort
        FQCN of a org.ehcache.config.EvictionAdvisor implementation
    -->
    <ehcache:eviction-advisor>com.pany.ehcache.MyEvictionAdvisor</ehcache:eviction-advisor>
    <!--
      OPTIONAL,
        Let's you configure your cache as a "cache-through",
        i.e. a Cache that uses a CacheLoaderWriter to load on misses,
        and write on mutative operations.
    -->
    <ehcache:loader-writer>
      <!--
          The FQCN implementing org.ehcache.spi.loaderwriter.CacheLoaderWriter
      -->
      <ehcache:class>com.pany.ehcache.integration.ProductCacheLoaderWriter</ehcache:class>
        <!-- Any further elements in another namespace -->
    </ehcache:loader-writer>
    <!--
        The maximal number of entries to be held in the Cache, prior to eviction starting
    -->
    <ehcache:heap unit="entries">200</ehcache:heap>
    <!--
       OPTIONAL
        Any further elements in another namespace
    -->
  </ehcache:cache>
  <!--
    OPTIONAL
      A <cache-template> defines a named template that can be used be <cache>
      definitions in this same file
      They have all the same property as the <cache> elements above
  -->
  <ehcache:cache-template name="myDefaultTemplate">
    <ehcache:expiry>
      <ehcache:none/>
    </ehcache:expiry>
    <!--
       OPTIONAL
        Any further elements in another namespace
    -->
  </ehcache:cache-template>
  <!--
    A <cache> that uses the template above by referencing the cache-template's
```

```
    name in the uses-template attribute:
  -->
  <ehcache:cache alias="customerCache" uses-template="myDefaultTemplate">
    <!--
      Adds the key and value type configuration
    -->
    <ehcache:key-type>java.lang.Long</ehcache:key-type>
    <ehcache:value-type>com.pany.domain.Customer</ehcache:value-type>
    <!--
      Overwrites the capacity limit set by the template to a new value
    -->
    <ehcache:heap unit="entries">200</ehcache:heap>
  </ehcache:cache>
</ehcache:config>
```

16 Ehcache XSDs

XSD namespaces and locations

- Core namespace: http://www.ehcache.org/v3
  - Location of the schema used in Terracotta Ehcache: http://www.ehcache.org/schema/ehcache-core-3.2.xsd
- JSR-107 namespace: http://www.ehcache.org/v3/jsr107
  - Location of the schema used in Terracotta Ehcache: http://www.ehcache.org/schema/ehcache-107-ext-3.2.xsd
- Transactions namespace: http://www.ehcache.org/v3/tx
  - Location of the schema used in Terracotta Ehcache: http://www.ehcache.org/schema/ehcache-tx-ext-3.2.xsd
- Cluster namespace: http://www.ehcache.org/v3/clustered
  - Location of the schema used in Terracotta Ehcache: http://www.ehcache.org/schema/ehcache-clustered-ext-3.2.xsd

Usage example

```
<eh:config
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:eh='http://www.ehcache.org/v3'
    xmlns:jsr107='http://www.ehcache.org/v3/jsr107'
    xsi:schemaLocation="
        http://www.ehcache.org/v3
            http://www.ehcache.org/schema/ehcache-core-3.2.xsd
        http://www.ehcache.org/v3/jsr107
            http://www.ehcache.org/schema/ehcache-107-ext-3.2.xsd">
</eh:config>
```

17 Management and Monitoring with Ehcache

Introduction

Managed objects like caches, cache managers and stores are registered into an `org.ehcache.management.ManagementRegistryService` instance.

A `ManagementRegistry` implementation has to understand the registered object and provide management and monitoring capabilities for them, including the capabilities' context.

Given a capability and a context, statistics can be collected or calls can be made.

The current `ManagementRegistry` implementation provides minimal support for `Ehcache` instances, providing a minimal set of statistics and actions via a couple of capabilities.

Making use of the ManagementRegistry

By default, a `ManagementRegistry` is automatically discovered and enabled, but can only be accessed by Ehcache internal services. If you wish to make use of it, you should create your own instance and pass it to the cache manager builder as a service:

```
CacheManager cacheManager = null;
try {
  DefaultManagementRegistryConfiguration registryConfiguration =
       new DefaultManagementRegistryConfiguration()
      .setCacheManagerAlias("myCacheManager1");   // 1
  ManagementRegistryService managementRegistry =
      new DefaultManagementRegistryService(registryConfiguration); // 2
  CacheConfiguration<Long, String> cacheConfiguration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(
    Long.class, String.class,
    ResourcePoolsBuilder.newResourcePoolsBuilder()
    .heap(1, MemoryUnit.MB).offheap(2, MemoryUnit.MB))
    .build();
  cacheManager = CacheManagerBuilder.newCacheManagerBuilder()
      .withCache("myCache", cacheConfiguration)
      .using(managementRegistry)   // 3
      .build(true);
  Object o =
    managementRegistry.withCapability("StatisticCollectorCapability")
    .call("updateCollectedStatistics",
      new Parameter("StatisticsCapability"),
      new Parameter(Arrays.asList("Cache:HitCount", "Cache:MissCount"),
        Collection.class.getName()))
    .on(Context.create("cacheManagerName", "myCacheManager1"))
    .build()
    .execute()
    .getSingleResult();
  System.out.println(o);
  Cache<Long, String> aCache = cacheManager.getCache(
     "myCache", Long.class, String.class);
  aCache.put(1L, "one");
  aCache.put(0L, "zero");
  aCache.get(1L);  // 4
  aCache.get(0L);  // 4
```

```
  aCache.get(0L);
  aCache.get(0L);
  Context context = StatsUtil.createContext(managementRegistry);  // 5
  StatisticQuery query =
      managementRegistry.withCapability("StatisticsCapability")   // 6
      .queryStatistic("Cache:HitCount")
      .on(context)
      .build();
  ResultSet<ContextualStatistics> counters = query.execute();
  ContextualStatistics statisticsContext = counters.getResult(context);
  Assert.assertThat(counters.size(), Matchers.is(1));
}
finally {
  if(cacheManager != null) cacheManager.close();
}
```

1	Optional: give a name to your cache manager by using a custom configuration
2	Create an instance of `org.ehcache.management.registry.DefaultManagementRegistryService`. This is only required because the service is used below.
3	Pass it as a service to the cache manager (if you only want to configure the ManagementRegistry, you can just pass the configuration instead)
4	Perform a few gets to increment the statistic's counter
5	Create the target statistic's context
6	Collect the get count statistic

Obviously, you may use the above technique to pass your own implementation of `ManagementRegistry`.

Capabilities and contexts

Capabilities are metadata of what the managed objects are capable of: a collection of statistic that can be queried and/or remote actions that can be called. Each capability requires a context to run in. For instance, cache-specific statistics require a cache manager name and a cache name to uniquely identify the cache on which you want to query stats or call an action.

```
CacheConfiguration<Long, String> cacheConfiguration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(
    Long.class, String.class, ResourcePoolsBuilder.heap(10))
    .build();
CacheManager cacheManager = null;
try {
  ManagementRegistryService managementRegistry =
     new DefaultManagementRegistryService();
```

```
  cacheManager = CacheManagerBuilder.newCacheManagerBuilder()
      .withCache("aCache", cacheConfiguration)
      .using(managementRegistry)
      .build(true);
  Collection<? extends Capability> capabilities =
      managementRegistry.getCapabilities();  // 1
  Assert.assertThat(capabilities.isEmpty(), Matchers.is(false));
  Capability capability = capabilities.iterator().next();
  String capabilityName = capability.getName();  // 2
  Collection<? extends Descriptor> capabilityDescriptions =
    capability.getDescriptors();  // 3
  Assert.assertThat(capabilityDescriptions.isEmpty(),
      Matchers.is(false));
  CapabilityContext capabilityContext =
      capability.getCapabilityContext();
  Collection<CapabilityContext.Attribute> attributes =
      capabilityContext.getAttributes();  // 4
  Assert.assertThat(attributes.size(), Matchers.is(2));
  Iterator<CapabilityContext.Attribute> iterator =
      attributes.iterator();
  CapabilityContext.Attribute attribute1 = iterator.next();
  Assert.assertThat(attribute1.getName(),
      Matchers.equalTo("cacheManagerName"));  // 5
  Assert.assertThat(attribute1.isRequired(), Matchers.is(true));
  CapabilityContext.Attribute attribute2 = iterator.next();
  Assert.assertThat(attribute2.getName(),
      Matchers.equalTo("cacheName"));  // 6
  Assert.assertThat(attribute2.isRequired(), Matchers.is(true));
  ContextContainer contextContainer =
    managementRegistry.getContextContainer();  // 7
  Assert.assertThat(contextContainer.getName(),
    Matchers.equalTo("cacheManagerName"));   // 8
  Assert.assertThat(contextContainer.getValue(),
    Matchers.startsWith("cache-manager-"));
  Collection<ContextContainer> subContexts =
    contextContainer.getSubContexts();
  Assert.assertThat(subContexts.size(), Matchers.is(1));
  ContextContainer subContextContainer =
    subContexts.iterator().next();
  Assert.assertThat(subContextContainer.getName(),
    Matchers.equalTo("cacheName"));   // 9
  Assert.assertThat(subContextContainer.getValue(),
    Matchers.equalTo("aCache"));
}
finally {
  if(cacheManager != null) cacheManager.close();
}
```

1	Query the `ManagementRegistry` for the registered managed objects' capabilities.
2	Each capability has a unique name you will need to refer to it.
3	Each capability has a collection of `Descriptor`s that contains the metadata of each statistic or action.
4	Each capability requires a context to which it needs to refer to.

5	The first attribute of this context is the cache manager name.
6	The second attribute of this context is the cache name. With both attributes, the capability can uniquely refer to a unique managed object.
7	Query the `ManagementRegistry` for the all the registered managed objects' contexts.
8	There is only one context here, and its name is the cache manager's name.
9	The above context has a subcontext: the cache's name.

The context containers give you all the attributes of all existing contexts. You can match the values returned by a context container to a capability's context by matching their respective names.

Actions

There are two forms of capabilities: statistics and action ones. The statistic ones offer a set of predefined statistics that can be queried at will, while the action ones offer a set of actions that can be taken upon a managed object. Examples of actions could be: clear caches, get their config or modify a config setting.

```
CacheConfiguration<Long, String> cacheConfiguration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(
    Long.class, String.class, ResourcePoolsBuilder.heap(10))
    .build();
CacheManager cacheManager = null;
try {
  ManagementRegistryService managementRegistry =
    new DefaultManagementRegistryService();
  cacheManager = CacheManagerBuilder.newCacheManagerBuilder()
      .withCache("aCache", cacheConfiguration)
      .using(managementRegistry)
      .build(true);
  Cache<Long, String> aCache =
    cacheManager.getCache("aCache", Long.class, String.class);
  aCache.put(0L, "zero");    // 1
  Context context =
      StatsUtil.createContext(managementRegistry);   // 2
  managementRegistry.withCapability("ActionsCapability") // 3
      .call("clear")
      .on(context)
      .build()
      .execute();
  Assert.assertThat(aCache.get(0L),
      Matchers.is(Matchers.nullValue()));  // 4
}
finally {
  if(cacheManager != null) cacheManager.close();
}
```

1	Put something in a cache.
2	Call the 'clear' action on the managed cache. Refer to the descriptors of the provider to get the exact list of action names and their required parameters.
3	Call the clear action on the cache.
4	Make sure that the cache is now empty.

Managing multiple cache managers

The default `ManagementRegistry` instance that is created when none are manually registered only manages a single cache manager by default, but sometimes you may want one `ManagementRegistry` to manage multiple cache managers.

`ManagementRegistry` instances are thread-safe, so one instance can be shared amongst multiple cache managers:

```
CacheConfiguration<Long, String> cacheConfiguration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(
    Long.class, String.class, ResourcePoolsBuilder.heap(10))
    .build();
CacheManager cacheManager1 = null;
CacheManager cacheManager2 = null;
try {
  SharedManagementService sharedManagementService =
    new DefaultSharedManagementService();   // 1
  cacheManager1 = CacheManagerBuilder.newCacheManagerBuilder()
      .withCache("aCache", cacheConfiguration)
      .using(new DefaultManagementRegistryConfiguration()
           .setCacheManagerAlias("myCacheManager-1"))
      .using(sharedManagementService)  // 2
      .build(true);
  cacheManager2 = CacheManagerBuilder.newCacheManagerBuilder()
      .withCache("aCache", cacheConfiguration)
      .using(new DefaultManagementRegistryConfiguration()
          .setCacheManagerAlias("myCacheManager-2"))
      .using(sharedManagementService)  // 3
      .build(true);
  Context context1 = Context.empty()
    .with("cacheManagerName", "myCacheManager-1")
    .with("cacheName", "aCache");
  Context context2 = Context.empty()
    .with("cacheManagerName", "myCacheManager-2")
    .with("cacheName", "aCache");
  Cache<Long, String> cache =
      cacheManager1.getCache("aCache", Long.class, String.class);
  cache.get(1L);//cache miss
  cache.get(2L);//cache miss
  StatisticQuery query = sharedManagementService
    .withCapability("StatisticsCapability")
    .queryStatistic("Cache:MissCount")
```

```
    .on(context1)
    .on(context2)
    .build();
  long val = 0;
  // it could be several seconds before the sampled stats
  // could become available
  // let's try until we find the correct value : 2
  do {
    ResultSet<ContextualStatistics> counters = query.execute();
    ContextualStatistics statisticsContext1 =
       counters.getResult(context1);
    Number counterContext1 = statisticsContext1.
      getStatistic("Cache:MissCount");
    // miss count is a sampled stat,
    //for example its values could be [0,1,2].
    // In the present case, only the last value is important to us,
    // the cache was eventually missed 2 times
    val = counterContext1.longValue();
  } while(val != 2);
}
finally {
  if(cacheManager2 != null) cacheManager2.close();
  if(cacheManager1 != null) cacheManager1.close();
}
```

1	Create an instance of `org.ehcache.management.SharedManagementService`
2	Pass it as a service to the first cache manager
3	Pass it as a service to the second cache manager

This way, all managed objects get registered into a common `ManagementRegistry` instance.

Rules for Statistics Calculation

This table describes the impact of each cache method on the statistics.

The statistics are:

Hit
An entry was asked for and found in the cache.

Miss
An entry was asked for and not found in the cache.

Put
An entry was added or updated.

Update
An existing entry was updated (this is a subset of put).

Removal
An entry was removed from the cache.

Expiration
An entry has expired and was therefore removed from the cache.

Eviction
An entry was evicted from the cache due to lack of space

Method	Hit	Miss	Put	Update	Remov	Expira	Evictio	Notes
clear	No	No	No	No	No	No	No	A bulk remove with no impact on statistics.
containsKey	No	No	No	No	No	No	No	Generates no hit or miss since the value isn't accessed.
forEach	Yes	No	No	No	No	Yes	No	Java 8. Will hit each entry once.
get	Yes	Yes	No	No	No	Yes	No	Hit when the entry is found, miss otherwise.
getAll	Yes	Yes	No	No	No	Yes	No	Like get but calculated per entry.
getAndPut	Yes	Yes	Yes	Yes	No	Yes	Yes	JSR107 specific. Hit and update when the entry is found, miss otherwise. Always put.
getAndRemove	Yes	Yes	No	No	Yes	Yes	No	JSR107 specific. Hit and remove when the entry is found, miss otherwise.
getAndReplace	Yes	Yes	Yes	Yes	No	Yes	Yes	JSR107 specific. Hit, put and

Method	Hit	Miss	Put	Update	Remov	Expira	Evictio	Notes
								update when the entry is found, miss otherwise.
invoke	Yes	Yes	Yes	Yes	Yes	Yes	Yes	JSR107 specific. Depends on how the EntryProcessor modifies the MutableEntry.
invokeAll	Yes	Yes	Yes	Yes	Yes	Yes	Yes	JSR107 specific. Like invoke but calculated per entry.
iterator	Yes	No	No	No	Yes	Yes	No	Will hit each iterated entry and count a removal on Iterator.remove.
loadAll	No	No	No	No	No	No	No	JSR107 specific. Background loading that has no impact on statistics.
put	No	No	Yes	Yes	No	No	Yes	Put an entry whether it already existed or not.
putAll	No	No	Yes	Yes	No	No	Yes	Like put but calculated per entry.
putIfAbsent	Yes	Yes	Yes	No	No	Yes	Yes	Will hit if the entry exists. Will put and miss if it doesn't.
remove(K)	No	No	No	No	Yes	Yes	No	Count a removal if the entry exists.

Method	Hit	Miss	Put	Update	Remov	Expira	Evictio	Notes
remove(K,V)	Yes	Yes	No	No	Yes	Yes	No	Hit if the key is found. Miss otherwise.
removeAll	No	No	No	No	Yes	No	No	One removal per entry in the cache.
removeAll(keys)	No	No	No	No	Yes	No	No	Like remove but calculated per entry.
replace(K,V)	Yes	Yes	Yes	Yes	No	Yes	Yes	Hit, put and update if the entry exists. Miss otherwise.
replace(K,O,N)	Yes	Yes	Yes	Yes	No	Yes	Yes	Hit if the entry exists.
spliterator	Yes	No	No	No	No	Yes	No	Java 8. Will hit for each iterated entry.

The statistics are provided by cache and tiers. Cache evictions and expirations are taken from the lowest (authoritative) tier.

18 Class Loading

About Class Loading

Since Ehcache is a library and supports user types both in configuration and in mapping keys or values, it must offer flexibility around class loading.

Default ClassLoader in Ehcache

The default ClassLoader from Ehcache will first try to use the thread context class loader, through Thread.currentThread().getContextClassLoader(). In case this fails to load the requested resource, it will then use the ClassLoader that loaded the Ehcache internal classes.

Handling User Types

The way to configure a ClassLoader and the scope of its use differ between Java and XML based configurations. However, regardless of the configuration method, each CacheManager and Cache is always linked individually to a specific ClassLoader instance.

Java configuration

During the configuration of the CacheManager or Cache, there are multiple extension points where a user type can be involved. This includes Serializer, CacheLoaderWriter and other similar companion objects. You can usually give an instance of these types to the configuration or an instance of the class to use. This effectively negates the need to think in terms of a class loader here.

However you can still pass a specific ClassLoader to the CacheManager configuration. You can also give a specific ClassLoader per cache if you need to, taking precedence over the one configured at the CacheManager level. These will be used at runtime (see the section "At runtime" on page 115 below).

If no class loader is specified, the default class loader from Ehcache (see the section "Default ClassLoader in Ehcache" on page 114) will be used.

XML configuration

When using XML to configure Ehcache, references to custom types are given through a fully qualified class name. This means that transforming these String names into the proper class representation may require a specific ClassLoader.

In order to support this, the XmlConfiguration constructors can take ClassLoader parameters:

public XmlConfiguration(URL url)
to use only default ClassLoader

public XmlConfiguration(URL url, final ClassLoader classLoader)
to use a specific ClassLoader at the CacheManager level

public XmlConfiguration(URL url, final ClassLoader classLoader, final Map<String, ClassLoader> cacheClassLoaders)
to use a specific ClassLoader at the CacheManager level and a map of <String, ClassLoader> which will be used to link a specific ClassLoader to a Cache by its alias

In the same way as for the Java configuration, the Cache level configuration takes precedence over the CacheManager level one. If no class loaders are specified, the default class loader from Ehcache will be used.

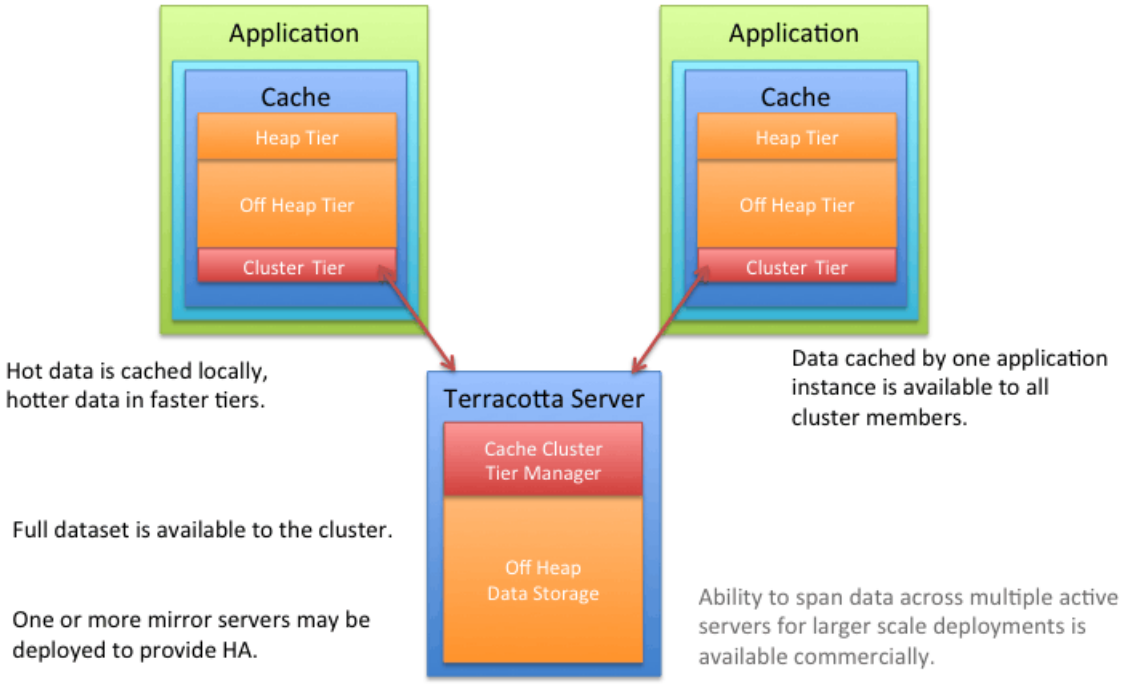
At runtime

As soon as serialization is involved with Cache keys or values, the class loader plays a role at runtime. The specific ClassLoader per Cache will be used by the serialization sub-system. It will allow deserialization to types that may not be visible by the EhcacheClassLoader.

19 Clustered Caches

Introduction

Distributed caching allows you to harness additional benefits of horizontal scale-out, without losing on low latency offered by local on-heap tiers.



To enable clustering with Terracotta, you will have to deploy a Terracotta Server configured with clustered cache storage.

You will then need to configure a cache manager to have clustering capabilities such that the caches it manages can utilize the clustered storage. Finally, any caches which should be distributed should be configured with a clustered storage tier.

Clustering Concepts

In this section we discuss some Terracotta clustering terms and concepts that you need to understand before creating cache managers and caches with clustering support.

Server off-heap resource

Server off-heap resources are storage resources defined at the server. Caches can reserve a storage area for their cluster tiers within these server off-heap resources.

Cluster Tier Manager

The Ehcache*Cluster Tier Manager* is the server-side component that gives clustering capabilities to a cache manager. Cache managers connect to it to get access to the server's storage resources so that the clustered tiers of caches defined in them can consume those

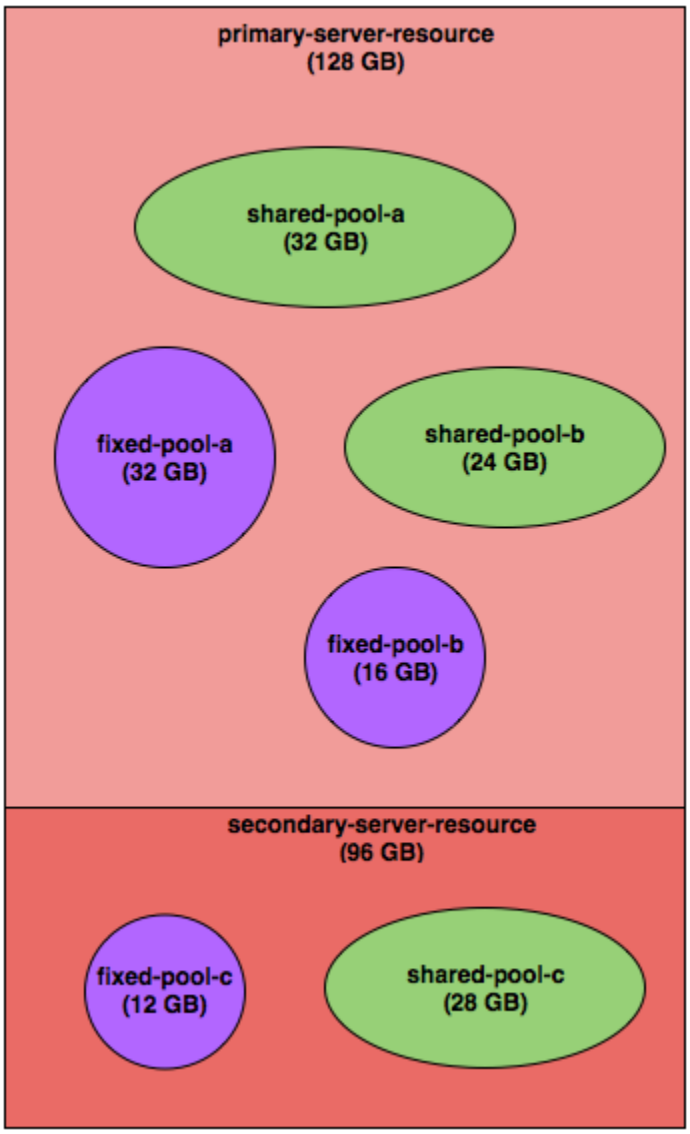
resources. An Ehcache cluster tier manager at the server side is identified by a unique identifier. Using the unique identifier of any given cluster tier manager, multiple cache managers can connect to the same cluster tier manager in order to share cache data. The cluster tier manager is also responsible for managing the storage of the cluster tier of caches, with the following different options.

Dedicated pool
Dedicated pools are a fixed-amount of storage pools allocated to the cluster tiers of caches. A dedicated amount of storage is allocated directly from server off-heap resources to these pools. And this storage space is used exclusively by a given cluster tier.

Shared pool
Shared pools are also fixed-amount storage pools, but can be shared by the cluster tiers of multiple caches. As in the case of dedicated pools, shared pools are also carved out from server off-heap resources. The storage available in these shared pools is strictly shared. In other words, no cluster tier can ask for a fixed-amount of storage from a shared pool.

Sharing of storage via shared pools does not mean that the data is shared. That is, if two caches are using a shared pool as their clustered tier, the data of each cache is still isolated but the underlying storage is shared. Consequently, when resource capacity is reached and triggers eviction, the evicted mapping can come from any of the cluster tiers sharing the pool.

Here is a pictorial representation of the concepts explained above:



Starting the Terracotta Server

Information on how to start the Terracotta Server is contained in the *Terracotta Server Administration Guide*.

You can start the Terracotta Server with the following configuration. It contains the bare minimum configuration required for the samples in the rest of the document to work.

```
<?xml version="1.0" encoding="UTF-8"?>
<tc-config xmlns="http://www.terracotta.org/config"
           xmlns:ohr="http://www.terracotta.org/config/offheap-resource">
  <services>
    <service id="resources">
      <ohr:offheap-resources>
        <ohr:resource name="primary-server-resource"
            unit="MB">128</ohr:resource>   <!-- 1 -->
        <ohr:resource name="secondary-server-resource"
```

```
            unit="MB">96</ohr:resource>    <!-- 2 -->
      </ohr:offheap-resources>
    </service>
  </services>
</tc-config>
```

The above configuration defines two named server off-heap resources:

1	An off-heap resource of 128 MB size named "primary-server-resource".
2	Another off-heap resource named "secondary-server-resource" with 96 MB capacity.

The rest of the document explains in detail how you can configure cache managers and caches to consume the server's off-heap resources.

Creating a Cache Manager with Clustering Capabilities

After starting the Terracotta Server, as described in the previous section, you can now proceed to create the cache manager. For creating the cache manager with clustering support you will need to provide the clustering service configuration. Here is a code sample that shows how to configure a cache manager with clustering service.

```
CacheManagerBuilder<PersistentCacheManager> clusteredCacheManagerBuilder =
     CacheManagerBuilder.newCacheManagerBuilder() // <1>
        .with(ClusteringServiceConfigurationBuilder.cluster(
         URI.create("terracotta://localhost:9410/my-application")) // <2>
            .autoCreate()); // <3>
PersistentCacheManager cacheManager =
   clusteredCacheManagerBuilder.build(true); // <4>
cacheManager.close(); // <5>
```

1	Returns the `org.ehcache.config.builders.CacheManagerBuilder` instance;
2	Use the `ClusteringServiceConfigurationBuilder`'s static method `.cluster(URI)` for connecting the cache manager to the clustered storage at the URI specified that returns the clustering service configuration builder instance. Sample URI provided in the example is pointing to the clustered storage instance named "my-application" on the Terracotta Server (assuming the server is running on localhost and port 9410.
3	Auto-create the clustered storage if it doesn't already exist.
4	Returns a fully initialized cache manager that can be used to create clustered caches.

5	Close the cache manager.

Cache Manager Configuration and Usage of Server Side Resources

This code sample demonstrates the usage of the concepts explained in the previous section in configuring a cache manager and clustered caches by using a broader clustering service configuration:

```
final CacheManagerBuilder<PersistentCacheManager> clusteredCacheManagerBuilder =
    CacheManagerBuilder.newCacheManagerBuilder()
        .with(ClusteringServiceConfigurationBuilder.cluster(
            URI.create("terracotta://localhost:9410/my-application")).autoCreate()
            .defaultServerResource("primary-server-resource")  // <1>
            .resourcePool("resource-pool-a", 28, MemoryUnit.MB,
                "secondary-server-resource")  // <2>
            .resourcePool("resource-pool-b", 32, MemoryUnit.MB)) // <3>
        .withCache("clustered-cache",
            CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
                String.class,  // <4>
            ResourcePoolsBuilder.newResourcePoolsBuilder()
                .with(ClusteredResourcePoolBuilder.clusteredDedicated(
                    "primary-server-resource", 32, MemoryUnit.MB))))  // <5>
        .withCache("shared-cache-1",
            CacheConfigurationBuilder.newCacheConfigurationBuilder(
                Long.class, String.class,
            ResourcePoolsBuilder.newResourcePoolsBuilder()
                .with(ClusteredResourcePoolBuilder.clusteredShared(
                    "resource-pool-a"))))  // <6>
        .withCache("shared-cache-2",
            CacheConfigurationBuilder.newCacheConfigurationBuilder(
                Long.class, String.class,
            ResourcePoolsBuilder.newResourcePoolsBuilder()
                .with(ClusteredResourcePoolBuilder.clusteredShared(
                     "resource-pool-a"))));  // <7>
final PersistentCacheManager cacheManager =
    clusteredCacheManagerBuilder.build(true);  // <8>
cacheManager.close();
```

1	`defaultServerResource(String)` on `ClusteringServiceConfigurationBuilder` instance sets the default server off-heap resource for the cache manager. From the example, cache manager sets its default server off-heap resource to "primary-server-resource" in the server.
2	Adds a resource pool for the cache manager with the specified name ("resource-pool-a") and size (28MB) consumed out of the named server off-heap resource "secondary-server-resource". A resource pool at the cache manager level maps directly to a shared pool at the server side.

3	Adds another resource pool for the cache manager with the specified name ("resource-pool-b") and size (32MB). Since the server resource identifier is not explicitly passed, this resource pool will be consumed out of default server resource provided in Step 3. This demonstrates that a cache manager with clustering support can have multiple resource pools created out of several server off-heap resources.
4	Provide the cache configuration to be created.
5	`ClusteredResourcePoolBuilder.clusteredDedicated(String , long , MemoryUnit)` allocates a dedicated pool of storage to the cache from the specified server off-heap resource. In this example, a dedicated pool of 32MB is allocated for `clustered-cache` from "primary-server-resource".
6	`ClusteredResourcePoolBuilder.clusteredShared(String)`, passing the name of the resource pool specifies that "shared-cache-1" shares the storage resources with other caches using the same resource pool ("resource-pool-a").
7	Configures another cache ("shared-cache-2") that shares the resource pool ("resource-pool-a") with "shared-cache-1".
8	Creates fully initialized cache manager with the clustered caches.

Ehcache Cluster Tier Manager Lifecycle

When configuring a cache manager to connect to a cluster tier manager there are three possible connection modes:

```
CacheManagerBuilder<PersistentCacheManager> autoCreate =
    CacheManagerBuilder.newCacheManagerBuilder()
    .with(ClusteringServiceConfigurationBuilder.cluster(
        URI.create("terracotta://localhost:9410/my-application"))
        .autoCreate()  // <1>
        .resourcePool("resource-pool", 32, MemoryUnit.MB,
            "primary-server-resource"))
    .withCache("clustered-cache",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class, String.class,
        ResourcePoolsBuilder.newResourcePoolsBuilder()
            .with(ClusteredResourcePoolBuilder.clusteredShared("resource-pool"))));
CacheManagerBuilder<PersistentCacheManager> expecting =
    CacheManagerBuilder.newCacheManagerBuilder()
    .with(ClusteringServiceConfigurationBuilder.cluster(
        URI.create("terracotta://localhost:9410/my-application"))
        .expecting()  // <2>
        .resourcePool("resource-pool", 32, MemoryUnit.MB, "primary-server-resource"))
    .withCache("clustered-cache",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class, String.class,
        ResourcePoolsBuilder.newResourcePoolsBuilder()
```

```
            .with(ClusteredResourcePoolBuilder.clusteredShared("resource-pool"))));
CacheManagerBuilder<PersistentCacheManager> configless =
    CacheManagerBuilder.newCacheManagerBuilder()
    .with(ClusteringServiceConfigurationBuilder.cluster(
        URI.create("terracotta://localhost:9410/my-application"))) // <3>
    .withCache("clustered-cache",
        CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class, String.class,
         ResourcePoolsBuilder.newResourcePoolsBuilder()
            .with(ClusteredResourcePoolBuilder.clusteredShared("resource-pool"))));
```

1	In auto-create mode if no cluster tier manager exists then one is created with the supplied configuration. If it exists and its configuration matches the supplied configuration then a connection is established. If the supplied configuration does not match then the cache manager will fail to initialize.
2	In expected mode if a cluster tier manager exists and its configuration matches the supplied configuration then a connection is established. If the supplied configuration does not match or the cluster tier manager does not exist then the cache manager will fail to initialize.
3	In config-less mode if a cluster tier manager exists then a connection is established without regard to its configuration. If it does not exist then the cache manager will fail to initialize.

Configuring a Clustered Cache

Clustered Storage Tier

```
CacheConfiguration<Long, String> config =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class, String.class,
    ResourcePoolsBuilder.newResourcePoolsBuilder()
        .heap(2, MemoryUnit.MB)  // <1>
        .with(ClusteredResourcePoolBuilder.clusteredDedicated(
            "primary-server-resource", 8, MemoryUnit.MB)))  // <2>
    .add(ClusteredStoreConfigurationBuilder.withConsistency(Consistency.STRONG))
    .build();
Cache<Long, String> cache = cacheManager.createCache("clustered-cache", config);
cache.put(42L, "All you need to know!");
```

1	Configuring the heap tier for cache.
2	Configuring the cluster tier of dedicated size from server off-heap resource using `ClusteredResourcePoolBuilder`.

The equivalent XML configuration is as follows:

```
<cache alias="clustered-cache">
  <resources>
    <heap unit="MB">10</heap>  <!-- 1 -->
    <tc:clustered-dedicated unit="MB">50</tc:clustered-dedicated> <!-- 2 -->
```

```
  </resources>
  <tc:clustered-store consistency="strong"/>
</cache>
```

1	Specify the heap tier for cache.
2	Specify the cluster tier for cache through a custom service configuration from the `clustered` namespace.

Specifying consistency level

Ehcache offers two levels of consistency:

Eventual
This consistency level indicates that the visibility of a write operation is not guaranteed when the operation returns. Other clients may still see a stale value for the given key. However this consistency level guarantees that for a mapping `(K, V1)` updated to `(K, V2)`, once a client sees `(K, V2)` it will never see `(K, V1)` again.

Strong
This consistency level provides strong visibility guarantees ensuring that when a write operation returns other clients will be able to observe it immediately. This comes with a latency penalty on the write operation required to give this guarantee.

```
CacheConfiguration<Long, String> config =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class, String.class,
        ResourcePoolsBuilder.newResourcePoolsBuilder()
            .with(ClusteredResourcePoolBuilder.clusteredDedicated(
            "primary-server-resource", 2, MemoryUnit.MB)))
    .add(ClusteredStoreConfigurationBuilder.withConsistency(Consistency.STRONG))  // <1>
    .build();
Cache<Long, String> cache = cacheManager.createCache("clustered-cache", config);
cache.put(42L, "All you need to know!"); // <2>
```

1	Specify the consistency level through the use of additional service configuration, using "strong" consistency here.
2	With the consistency used above, this `put` operation will return only when all other clients have had the corresponding mapping invalidated.

The equivalent XML configuration is as follows:

```
<cache alias="clustered-cache">
  <resources>
    <tc:clustered-dedicated unit="MB">50</tc:clustered-dedicated>
  </resources>
  <tc:clustered-store consistency="strong"/>  <!-- 1 -->
</cache>
```

1	Specify the consistency level through a custom service configuration from the `clustered` namespace.

Clustered Cache Expiry

Expiry in clustered caches work with an exception that `Expiry#getExpiryForAccess` is handled on a best effort basis for cluster tiers. It may not be as accurate as in the case of local tiers.

Clustered Unspecified Inheritance

We have included an option which allows a cache to be created inside the cache manager without having to explicitly define its cluster tier resource pool allocation. In order to use this feature the cluster tier must already have been created with either a *shared* or *dedicated* resource pool.

In this case the definition of the cluster resource is done simply with a `clustered()` resource pool. This effectively means *unspecified* and indicates you expect it to exist already. It will then inherit the clustered resource pool as it was configured when creating the cluster tier.

This option provides many benefits. The main benefit is it simplifies clustered configuration by allowing clustered resource pool configuration to be handled by one client, then all subsequent clients can inherit this configuration. In addition, it also reduces clustered pool allocation configuration errors. More importantly, sizing calculations only need to be done by one person and updated in one location. Thus any programmer can use the cache without having to worry about using matching resource pool allocations.

The example code below shows how this can be implemented.

```
CacheManagerBuilder<PersistentCacheManager> cacheManagerBuilderAutoCreate =
    CacheManagerBuilder.newCacheManagerBuilder()
        .with(ClusteringServiceConfigurationBuilder.cluster(
            URI.create("terracotta://localhost:9410/my-application"))
            .autoCreate()  // <1>
            .resourcePool("resource-pool", 32, MemoryUnit.MB, "primary-server-resource"));
final PersistentCacheManager cacheManager1 = cacheManagerBuilderAutoCreate.build(false);
cacheManager1.init();
CacheConfiguration<Long, String> cacheConfigDedicated =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class, String.class,
ResourcePoolsBuilder.newResourcePoolsBuilder()
    .with(ClusteredResourcePoolBuilder.clusteredDedicated(
        "primary-server-resource", 8, MemoryUnit.MB)))  // <2>
.add(ClusteredStoreConfigurationBuilder.withConsistency(Consistency.STRONG))
.build();
Cache<Long, String> cacheDedicated = cacheManager1.createCache(
    "my-dedicated-cache", cacheConfigDedicated);  // <3>
CacheManagerBuilder<PersistentCacheManager> cacheManagerBuilderExpecting =
    CacheManagerBuilder.newCacheManagerBuilder()
        .with(ClusteringServiceConfigurationBuilder.cluster(
            URI.create("terracotta://localhost:9410/my-application"))
            .expecting()  // <4>
            .resourcePool("resource-pool", 32, MemoryUnit.MB, "primary-server-resource"));
final PersistentCacheManager cacheManager2 = cacheManagerBuilderExpecting.build(false);
cacheManager2.init();
CacheConfiguration<Long, String> cacheConfigUnspecified =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class, String.class,
ResourcePoolsBuilder.newResourcePoolsBuilder()
    .with(ClusteredResourcePoolBuilder.clustered()))  // <5>
```

```
.add(ClusteredStoreConfigurationBuilder.withConsistency(Consistency.STRONG))
.build();
Cache<Long, String> cacheUnspecified = cacheManager2.createCache(
    "my-dedicated-cache", cacheConfigUnspecified); // <6>
```

1	Configure the first cache manager with auto create
2	Build a cache configuration for a clustered *dedicated* resource pool
3	Create cache "my-dedicated-cache" using the cache configuration
4	Configure the second cache manager as *expecting* (auto create off)
5	Build a cache configuration for a clustered *unspecified* resource pool, which will use the previously configured clustered *dedicated* resource pool.
6	Create cache with the same name "my-dedicated-cache" and use the clustered *unspecified* cache configuration

Creating a Cluster with Multiple Stripes

Server-side tasks for setting up a Multi-Stripe Cluster

- Before you create a cluster, start *at least* one server on each of the stripes which are going to be part of the cluster.
- Use the `configure` command of the cluster tool to configure a cluster, specifying the Terracotta configuration file for all required stripes.

  For details about usage of the cluster tool, refer to the cluster tool documentation in the *Terracotta Server Administration Guide*.

Client Side Code

The client-side code for connecting to a cluster will be similar to the following:

```
PersistentCacheManager cacheManager =
    CacheManagerBuilder.newCacheManagerBuilder()
    .with(ClusteringServiceConfigurationBuilder.cluster(URI.create(
       "terracotta://localhost:9410/multi-stripe-cm")).autoCreate())   // 1
    .withCache("myCache", CacheConfigurationBuilder.newCacheConfigurationBuilder(
        Long.class, String.class, ResourcePoolsBuilder.heap(10)
        .with(ClusteredResourcePoolBuilder.clusteredDedicated(
             "primary-server-resource", 10, MemoryUnit.MB))))          // 2
    .build(true);
Cache<Long, String> myCache = cacheManager.getCache("myCache",
    Long.class, String.class);
myCache.put(42L, "Success!");
System.out.println(myCache.get(42L));
```

1	Connection to a multi-stripe cluster requires a URI with the `host:port` of **all** servers that belong to **one** of the stripes.
2	The size of the resource pool on each stripe is the total pool size divided by the numbers of stripes in the cluster.

Note:

- It's recommended to check the availability of resources across all stripes in the cluster before configuring a cache, otherwise cache creation will fail on the stripes with insufficient resources.
- Creating cluster with cluster tool is a **mandatory first step**, otherwise cache creation will succeed on only one of the stripes represented by the provided URI. Creating a cluster after creating clustered resources is not supported and will result in errors.

20 Fast Restartability

Overview of Fast Restartability

The *Fast Restart* feature provides enterprise-ready crash resilience by keeping a consistent, real-time record of in-memory data on a disk. After a shutdown - planned or unplanned - the next time the application starts up, all of the data that was in memory prior to the shutdown is made available again. This reduces the load on a remote data source, such as a database serving as a System of Record (SoR), and enables applications to resume from the state prior to the shutdown. The time taken for the data to be restored from the disk depends on the quantity of data restored, and the capabilities of the disk itself.

Creating a Restartable Cache Manager

Configuring a restartable server lays the foundation for application data to be restartable.

Refer to the topic *Platform Persistence* in the section *Configuring the Terracotta Server* in the *Terracotta Server Administration Guide* for information on configuring a restartable server.

Caches must be configured as restartable in a restartable CacheManager. A non-restartable CacheManager cannot contain restartable Caches. However, a restartable CacheManager can contain restartable as well as non-restartable Caches. In the latter case, only the Cache configuration is restored upon a server restart, and not the Cache data.

The following example illustrates a restartable CacheManager creation. Only the parts which are different in a restartable CacheManager are explained.

```
EnterpriseServerSideConfigurationBuilder serverSideConfigBuilder =
    EnterpriseClusteringServiceConfigurationBuilder
    .enterpriseCluster(URI.create(
        "terracotta://localhost:9410/my-application"))     // 1
    .autoCreate()
    .defaultServerResource("primary-server-resource")
    .restartable("data-directory-name")                    // 2
PersistentCacheManager cacheManager = CacheManagerBuilder
    .newCacheManagerBuilder()
    .with(serverSideConfigBuilder)                         // 3
    .build(true);
cacheManager.close();
cacheManager.destroy();                                    // 4
```

1	EnterpriseClusteringServiceConfigurationBuilder's static method enterpriseCluster(URI) connects the cache manager to the clustered storage at the URI specified. The sample URI provided in the example points to the clustered storage instance named 'my-application' on the Terracotta Server (assuming the server is running on `localhost` and port `9410`).
2	The restartable(String) method accepts a logical data directory name to specify where the fast restart logs should be stored.

	The string specified here is a logical name, which should be present in the Terracotta Server configuration. The server configuration has a mapping of this logical name to a directory path. Using this API is an absolute MUST to create a restartable `CacheManager`, in the absence of which the`CacheManager` will be created as a non-restartable `CacheManager`.
3	CacheManagerBuilder uses serverSideConfigBuilder created in Step 1 above.
4	Calling `destroy()` on the `CacheManager` will destroy all caches contained inside the `CacheManager`, all associated metadata and the `CacheManager` itself. This can be done to free up resources and to ensure that cache contents don't re-appear on a server restart.

Creating a Restartable Cache

This is the part which makes the application data restartable. As mentioned in the section "Creating a Restartable Cache" on page 131, a restartable Cache can be configured **only** in a restartable CacheManager.

A restartable CacheConfiguration API is a lot like the regular (non-restartable) CacheConfiguration API, except that it takes the restartable variant of ClusteredResourcePool, which makes the cache contents restartable.

Creating Restartable Resource Pools

Let us do a quick recap of restartability:

- A restartable server enables you to have *restartable objects* on the server.
- A restartable CacheManager makes the *cache configuration* restartable.
- A restartable resource pool configuration makes your *cache data* restartable.

As with non-restartable resource pools, restartable resource pools can be dedicated or shared. See "Clustering Concepts" on page 118 for a refresher on resource pools.

```
ClusteredResourcePool restartableDedicatedPool =
    ClusteredRestartableResourcePoolBuilder
    .clusteredRestartableDedicated(
        "primary-server-resource", 4, MemoryUnit.MB);  // 1
ClusteredResourcePool restartableSharedPool =
    ClusteredRestartableResourcePoolBuilder
    .clusteredRestartableShared("shared-pool");        // 2
```

1	ClusteredRestartableResourcePoolBuilder's static method clusteredRestartableDedicated(String, long, MemoryUnit) configures a restartable

	dedicated pool of size 4 MB from the server's primary-server-resource. As the name suggests, this pool will be dedicated to the `Cache` using it.
2	ClusteredRestartableResourcePoolBuilder's static method clusteredRestartableShared(String) specifies a restartable shared pool with the name shared-pool. This pool can be shared by multiple caches, and no cache can exclusively reserve this pool for its sole use.

Example of a Restartability Scenario

The following example illustrates a typical scenario of a `CacheManager` containing a mix of restartable and non-restartable caches.

```
EnterpriseServerSideConfigurationBuilder serverSideConfigBuilder =
    EnterpriseClusteringServiceConfigurationBuilder
    .enterpriseCluster(connectionURI)
    .autoCreate()
    .defaultServerResource("primary-server-resource")
    .resourcePool("shared-pool", 20, MemoryUnit.MB,
        "secondary-server-resource")                                    // 1
    .restartable("data-directory-name");
PersistentCacheManager cacheManager = CacheManagerBuilder
    .newCacheManagerBuilder()
    .with(serverSideConfigBuilder)
    .build(true);
ClusteredResourcePool restartableDedicatedPool =
    ClusteredRestartableResourcePoolBuilder
    .clusteredRestartableDedicated(
        "primary-server-resource", 4, MemoryUnit.MB);
ClusteredResourcePool restartableSharedPool =
    ClusteredRestartableResourcePoolBuilder
    .clusteredRestartableShared("shared-pool");
ClusteredResourcePool nonRestartableDedicatedPool =
    ClusteredResourcePoolBuilder
    .clusteredDedicated(
        "primary-server-resource", 8, MemoryUnit.MB);                   // 2
ClusteredResourcePool nonRestartableSharedPool =
    ClusteredResourcePoolBuilder
    .clusteredShared("shared-pool");                                    // 3
Cache<Long, String> restartableDedicatedPoolCache = cacheManager
    .createCache("restartableDedicatedPoolCache",
         CacheConfigurationBuilder
        .newCacheConfigurationBuilder(Long.class, String.class,
            ResourcePoolsBuilder.newResourcePoolsBuilder()
            .with(restartableDedicatedPool)));                          // 4
Cache<Long, String> restartableSharedPoolCache = cacheManager
    .createCache("restartableSharedPoolCache",
        CacheConfigurationBuilder
        .newCacheConfigurationBuilder(Long.class, String.class,
            ResourcePoolsBuilder.newResourcePoolsBuilder()
            .with(restartableSharedPool)));                             // 5
Cache<String, Boolean> nonRestartableDedicatedPoolCache = cacheManager
    .createCache("nonRestartableDedicatedPoolCache",
         CacheConfigurationBuilder
        .newCacheConfigurationBuilder(String.class, Boolean.class,
            ResourcePoolsBuilder.newResourcePoolsBuilder()
            .with(nonRestartableDedicatedPool)));                       // 6
```

```
Cache<String, Boolean> nonRestartableSharedPoolCache = cacheManager
    .createCache("nonRestartableSharedPoolCache",
        CacheConfigurationBuilder
        .newCacheConfigurationBuilder(String.class, Boolean.class,
            ResourcePoolsBuilder.newResourcePoolsBuilder()
            .with(nonRestartableSharedPool)));                          // 7
cacheManager.close();
cacheManager.destroy();
```

1	Registers with the `CacheManager` a resource pool named `shared-pool` of size 20 MB, reserved from server's `secondary-server-resource`.
2	Creates a non-restartable dedicated resource pool of size 8 MB from server's `primary-server-resource`.
3	Specifies a non-restartable shared resource pool which could be used by multiple caches.
4	Creates a `Cache` using a restartable dedicated pool.
5	Creates a `Cache` using a restartable shared pool.
6	Creates a `Cache` using the non-restartable dedicated pool created in Step 2 above.
7	Creates a `Cache` using the non-restartable shared pool created in Step 3 above. Note here that the same pool is shared by restartable and non-restartable caches.

General Notes on Configuring Restartability

- A restartable CacheManager**must** be supplied with a data directory name using the restartable(<data-root-identifier>) API. If not, the CacheManager would be created as a non-restartable CacheManager.

  Refer to the topic *Data Directories* in the section *Configuring the Terracotta Server* in the *Terracotta Server Administration Guide* for more details.

- The data directory name specified at CacheManager should be any one of the data directories specified in the server configuration. For example, if a server is configured with data directories `root1`, `root2` and `root3`, a CacheManager could be supplied any one of these three data directories. If the data directory specified in the CacheManager configuration does not match the data directory specified in server configuration, the following Exception is thrown.

  ```
  org.terracotta.entity.ConfigurationException:
  Given identifier (specified_directory) is not present in the server's configuration;
  Cannot configure Fast Restart Store for clustered tier manager
  ```

```
(specified_cache_manager) due to invalid configuration specified by client.
```

- As with a non-restartable CacheManager, the server resource supplied in the defaultServerResource API should exist in the Terracotta Server configuration.
- As should be obvious, the specified pool size shouldn't exceed the server resource size.
- Different clients who want to connect to the same CacheManager should use the exact same configuration. Once CacheManager is created, any subsequent attempts to create the same CacheManager with a different configuration will fail.

21 Hybrid Caching

Overview of Hybrid Caching

A *full* restartable CacheManager stores **all** of its cache data in server off-heap. A *hybrid* restartable CacheManager stores only **some** (explained in "Configuring a Hybrid Cache" on page 137 section below) of its cache data in server off-heap. The off-heap tier in this scenario is *backed by* a disk, meaning that the entries in the off-heap tier are a subset of the entries in the disk, and the disk provides crash-resilience to the cache data. This has the following advantages:

- Cache size can exceed the available off-heap memory.
- Predictable low latencies are guaranteed at very large scale.
- Crash-resilience (see "Fast Restartability" on page 129) is provided to the cache data.

Ehcache provides the flexibility of configuring hybrid at the CacheManager level. Thus, we could have full as well as hybrid cache managers on the same Terracotta Server.

Note: Because of the random-access read/write pattern of hybrid, it is recommended to use hybrid with Solid State Drives (SSDs) only.

In the subsequent sections, we'll refer to a hybrid restartable CacheManager simply as a hybrid CacheManager, because a non-restartable hybrid CacheManager does not exist.

Configuring a Hybrid Cache Manager

A hybrid CacheManager configuration is very much like a full CacheManager (refer to "Cache Manager Configuration and Usage of Server Side Resources" on page 122 for details), except that instead of taking a FULL RestartableOffHeapMode type, it takes a PARTIAL RestartableOffHeapMode type. Everything else remains the same at this configuration level.

```
EnterpriseServerSideConfigurationBuilder serverSideConfigBuilder =
    EnterpriseClusteringServiceConfigurationBuilder
    .enterpriseCluster(URI.create(
        "terracotta://localhost:9410/my-application"))
    .autoCreate()
    .defaultServerResource("primary-server-resource")
    .restartable("data-directory-name")
    .withRestartableOffHeapMode(RestartableOffHeapMode.PARTIAL);   // 1
PersistentCacheManager cacheManager = CacheManagerBuilder
    .newCacheManagerBuilder()
    .with(serverSideConfigBuilder)
    .build(true);
cacheManager.close();
cacheManager.destroy();
```

1	EnterpriseClusteringServiceConfigurationBuilder's withRestartableOffHeapMode(RestartableOffHeapMode) API lets you configure a hybrid CacheManager using the value PARTIAL in enum

	RestartableOffHeapMode. The value of FULL signifies a full (non-hybrid) CacheManager. If this API is not used, the CacheManager is created as a full CacheManager by default.

Configuring a Hybrid Cache

Hybrid caches can be configured by specifying a dataPercent value at the ResourcePool configuration level. dataPercent signifies the percentage of cache data to be kept in the off-heap tier. If the cache data exceeds this limit, it goes to the disk, provided that the metadata is large enough to store this information; else cache eviction will be triggered.

Note:

- Hybrid caches only support dedicated resource pools. Thus, it would be illegal to create a hybrid cache using shared restartable resource pools.
- Conceptually, a hybrid Cache with a dataPercent of 100 is not equal to a full Cache.

Let us consider an example configuration where a Cache uses a ResourcePool of size 1 GB. Consider the following:

Data Percent value	Description
0 (default value)	This is a 'Pure Hybrid' configuration where all cache data will reside on disk and all metadata will reside in off-heap. The ResourcePool will be used only to store metadata, and can contain metadata for approximately 26 million entries, irrespective of the Cache key type. Evictions will happen when the entries exceed this number.
50	512 MB of the ResourcePool will be used to store cache entries, and the remaining 512 MB to store metadata. The number of entries in the cached data portion is not predictable and depends on Cache key size, data size, operations performed on the Cache etc. The number of entries in the metadata can be computed as approximately 8 million. Evictions will happen when the entries exceed this number.
95	972.8 MB of the ResourcePool will be used to store cache entries, and the remaining 51.2 MB to store metadata. As mentioned above - the number of entries in the cached data portion is not predictable and depends on Cache key size, data size, operations performed on the Cache etc. The number of entries in the metadata can be computed as approximately 0.8 million.

Data Percent value	Description
	Evictions will happen when the entries exceed this number.

Example of a Hybrid Scenario

```
EnterpriseServerSideConfigurationBuilder serverSideConfigBuilder =
    EnterpriseClusteringServiceConfigurationBuilder
    .enterpriseCluster(connectionURI.resolve("/cacheManager"))
    .autoCreate()
    .defaultServerResource("primary-server-resource")
    .restartable("data-directory-name")
    .withRestartableOffHeapMode(RestartableOffHeapMode.PARTIAL);           // 1
PersistentCacheManager cacheManager = CacheManagerBuilder
    .newCacheManagerBuilder()
    .with(serverSideConfigBuilder)
    .build(true);
ClusteredResourcePool pool1 = ClusteredRestartableResourcePoolBuilder
    .clusteredRestartableDedicated(
    "primary-server-resource", 4, MemoryUnit.MB, 25);                      // 2
ClusteredResourcePool pool2 = ClusteredRestartableResourcePoolBuilder
    .clusteredRestartableDedicated(
    "primary-server-resource", 12, MemoryUnit.MB, 50);                     // 3
Cache<Long, String> restartableDedicatedPoolCache = cacheManager
    .createCache("restartableDedicatedPoolCache", CacheConfigurationBuilder
        .newCacheConfigurationBuilder(Long.class, String.class,
            ResourcePoolsBuilder.newResourcePoolsBuilder().with(pool1)));  // 4
Cache<Long, String> restartableSharedPoolCache = cacheManager
    .createCache("restartableSharedPoolCache", CacheConfigurationBuilder
        .newCacheConfigurationBuilder(Long.class, String.class,
            ResourcePoolsBuilder.newResourcePoolsBuilder().with(pool2)));  // 5
cacheManager.close();
cacheManager.destroy();
```

1	Creates a hybrid CacheManager.
2	Creates a pool of size 4 MB from primary-server-resource, with a dataPercent value of 25.
3	Creates a pool of size 12 MB from primary-server-resource, with a dataPercent value of 50.
4	Creates a Cache using pool1 created from Step 2 above.
5	Creates a Cache using pool2 created from Step 3 above.

General Notes on Configuring Hybrid

- Permissible values of dataPercent are 0 through 99, both values included.
- dataPercent values are not allowed for full caches. If an attempt to configure a full Cache with any dataPercent value is made, the following Exception is thrown:

```
org.terracotta.entity.ConfigurationException:
dataPercent attribute has no meaning for a clustered tier manager
with FULL restartable offheap mode
```

22 Migrating Code from Ehcache v2

This guide provides sample code snippets to help migrate Ehcache v2 code to Ehcache v10 code.

Per Mapping Expiry

Per mapping expiry covers use cases where a subset of mappings have different expiration settings than the ones configured at the cache level.

Ehcache 2.x Code

Here we are creating a cache manager that has a default time-to-live (TTL) expiry.

Before adding, we verify the expiry and set it on the Element only when different than the Cache expiry.

```
int defaultCacheTTLInSeconds = 20;
CacheManager cacheManager = initCacheManager();
CacheConfiguration cacheConfiguration = new CacheConfiguration().name("cache")
    .maxEntriesLocalHeap(100)
    .timeToLiveSeconds(defaultCacheTTLInSeconds);    // 1
cacheManager.addCache(new Cache(cacheConfiguration));
Element element = new Element(10L, "Hello");
int ttlInSeconds = getTimeToLiveInSeconds((Long)element.getObjectKey(),
   (String)element.getObjectValue());               // 2
if (ttlInSeconds != defaultCacheTTLInSeconds) {     // 3
  element.setTimeToLive(ttlInSeconds);
}
cacheManager.getCache("cache").put(element);
System.out.println(cacheManager.getCache("cache").get(10L).getObjectValue());
sleep(2100);                                        // 4
// Now the returned element should be null, as the mapping is expired.
System.out.println(cacheManager.getCache("cache").get(10L));
```

1	Expiry duration defined at the cache level.
2	Compute the mapping expiry using the helper method getTimeToLiveInSeconds.
3	Only setting the computed expiry on element if other than default expiry.
4	Waiting for 2.1 seconds - assuming 2 seconds is the custom expiry duration - to get the mapping to be expired.

Corresponding Ehcache 10.x Code

Here we are creating a cache manager with a cache configuration specifying a custom expiry, having dedicated logic in the methods called during the lifecycle of added and updated mappings.

```
CacheManager cacheManager = initCacheManager();
CacheConfigurationBuilder<Long, String> configuration =
    CacheConfigurationBuilder.newCacheConfigurationBuilder(Long.class,
         String.class, ResourcePoolsBuilder
        .heap(100))
        .withExpiry(new Expiry<Long, String>() {        // 1
          @Override
          public Duration getExpiryForCreation(Long key, String value) {
            return getTimeToLiveDuration(key, value);   // 2
          }
          @Override
          public Duration getExpiryForAccess(Long key, ValueSupplier<?
            extends String> value) {
            return null;  // Keeping the existing expiry
          }
          @Override
          public Duration getExpiryForUpdate(Long key, ValueSupplier<?
            extends String> oldValue, String newValue) {
            return null;  // Keeping the existing expiry
          }
        });
cacheManager.createCache("cache", configuration);
Cache<Long, String> cache = cacheManager.getCache("cache", Long.class,
String.class);
cache.put(10L, "Hello");
System.out.println(cache.get(10L));
sleep(2100);                                            // 3
// Now the returned value should be null, as mapping is expired.
System.out.println(cache.get(10L));
```

1	Defining custom expiry to be called during the lifecycle of added mappings.
2	During mapping creation, defining expiry duration using the helper method getTimeToLiveDuration.
3	Waiting for 2.1 seconds - assuming 2 seconds is the custom expiry duration - to get the mapping to be expired.

So to migrate the former Ehcache per mapping expiry code to the current version of Ehcache, move the expiry computation logic to the getExpiryForCreation method of the created custom expiry.