Ehcache Developer Guide

Version 2.10

April 2015

This document applies to Ehcache Version 2.10 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: EHC-DG-210-20150415

Table of Contents

1 Key Classes and Methods

About the Key Classes

Ehcache consists of a *CacheManager*, which manages logical data sets represented as *Caches*. A Cache object contains *Elements*, which are essentially name-value pairs. You can use Cache objects to hold any kind of data that you want to keep in memory, not just data that you want to cache.

Caches are physically implemented, either in-memory or on disk . The logical representations of these components are actualized mostly through the following classes:

- CacheManager
- Cache
- Element

These classes form the core of the Ehcache API. The methods provided by these classes are largely responsible for providing programmatic access to a cache or in-memory data store.

CacheManager

The CacheManager class is used to manage caches. Creation of, access to, and removal of caches is controlled by a named CacheManager.

CacheManager Creation Modes

CacheManager supports two creation modes: singleton and instance. The two types can exist in the same JVM. However, multiple CacheManagers with the same name are not allowed to exist in the same JVM. CacheManager() constructors creating non-Singleton CacheManagers can violate this rule, causing a NullPointerException. If your code might create multiple CacheManagers of the same name in the same JVM, avoid this error by using the static CacheManager.create() methods, which always return the named (or default unnamed) CacheManager if it already exists in that JVM. If the named (or default unnamed) CacheManager does not exist, the CacheManager.create() methods create it.

For singletons, calling CacheManager.create(...) returns the existing singleton CacheManager with the configured name (if it exists) or creates the singleton based on the passed-in configuration.

To work from configuration, use the CacheManager.newInstance(...) method, which parses the passed-in configuration to either get the existing named CacheManager or create that CacheManager if it doesn't exist.

To review, the behavior of the CacheManager creation methods is as follows:

- `CacheManager.newInstance(Configuration configuration)` – Create a new CacheManager or return the existing one named in the configuration.
- `CacheManager.create()` – Create a new singleton CacheManager with default configuration, or return the existing singleton. This is the same as `CacheManager.getInstance()`.
- `CacheManager.create(Configuration configuration)` – Create a singleton CacheManager with the passed-in configuration, or return the existing singleton.
- `new CacheManager(Configuration configuration)` – Create a new CacheManager, or throw an exception if the CacheManager named in the configuration already exists or if the parameter (configuration) is null.

Note that in instance-mode (non-singleton), where multiple CacheManagers can be created and used concurrently in the same JVM, each CacheManager requires its own configuration.

If the Caches under management use the disk store, the disk-store path specified in each CacheManager configuration should be unique. This is because when a new CacheManager is created, a check is made to ensure that no other CacheManagers are using the same disk-store path. Depending upon your persistence strategy, Ehcache will automatically resolve a disk-store path conflict, or it will let you know that you must explicitly configure the disk-store path.

If managed caches use only the memory store, there are no special considerations.

See the Javadoc for Ehcache for more information on these methods, including options for passing in configuration. For examples, see the code samples in "Creating a CacheManager" on page 12.

Cache

A Cache is a thread-safe logical representation of a set of data elements, analogous to a cache region in many caching systems. Once a reference to a cache is obtained (through a CacheManager), logical actions can be performed. The physical implementation of these actions is relegated to the stores. For more information about the stores, see “Configuring Storage Tiers” in the *Configuration Guide* for Ehcache.

Caches are instantiated from configuration or programmatically using one of the Cache() constructors. Certain cache characteristics, such as Automatic Resource Control (ARC)-related sizing, and pinning, must be set using configuration.

Cache methods can be used to get information about the cache (for example, getCacheManager(), isNodeBulkLoadEnabled(), and isSearchable()), or perform certain cache-wide operations (for example, flush, load, initialize, and dispose).

The methods provided in the Cache class also allow you to work with cache elements (for example, get, set, remove, and replace) as well as get information about the them (for example, isExpired, isPinned).

Element

An element is an atomic entry in a cache. It has a key, a value, and a record of accesses. Elements are put into and removed from caches. They can also expire and be removed by the cache, depending on the cache settings.

There is an API for Objects in addition to the one for Serializable. Non-serializable Objects can be stored only in heap. If an attempt is made to persist them, they are discarded with a DEBUG-level log message but no error.

The APIs are identical except for the return methods from Element: getKeyValue() and getObjectValue() are used by the Object API in place of getKey() and getValue().

2 Basic Caching

Creating a CacheManager

All usages of the Ehcache API start with the creation of a CacheManager. The following code snippets illustrate various ways to create one.

Singleton versus Instance

The following creates a singleton CacheManager using defaults, then list caches.

```
CacheManager.create();
String[] cacheNames = CacheManager.getInstance().getCacheNames();
```

The following creates a CacheManager instance using defaults, then list caches.

```
CacheManager.newInstance();
String[] cacheNames = manager.getCacheNames();
```

The following creates two CacheManagers, each with a different configuration, and list the caches in each.

```
CacheManager manager1 = CacheManager.newInstance("src/config/ehcache1.xml");
CacheManager manager2 = CacheManager.newInstance("src/config/ehcache2.xml");
String[] cacheNamesForManager1 = manager1.getCacheNames();
String[] cacheNamesForManager2 = manager2.getCacheNames();
```

Loading a Configuration

When a CacheManager is created, it creates caches found in a provided configuration.

The following creates a CacheManager based on the configuration defined in the ehcache.xml file in the classpath.

```
CacheManager manager = CacheManager.newInstance();
```

The following creates a CacheManager based on a specified configuration file.

```
CacheManager manager = CacheManager.newInstance("src/config/ehcache.xml");
```

The following creates a CacheManager from a configuration resource in the classpath.

```
URL url = getClass().getResource("/anotherconfigurationname.xml");
CacheManager manager = CacheManager.newInstance(url);
```

The following creates a CacheManager from a configuration in an InputStream.

```
InputStream fis = new FileInputStream(new File
("src/config/ehcache.xml").getAbsolutePath());
try {
  CacheManager manager = CacheManager.newInstance(fis);
} finally {
  fis.close();
}
```

Adding and Removing Caches Programmatically

Adding Caches Programmatically

You are not limited to using caches that are placed in the CacheManager configuration. A new cache based on the default configuration can be added to a CacheManager very simply:

```
manager.addCache(cacheName);
```

For example, the following adds a cache called *testCache* to CacheManager called *singletonManager* . The cache is configured using defaultCache from the CacheManager configuration.

```
CacheManager singletonManager = CacheManager.create();
singletonManager.addCache("testCache");
Cache test = singletonManager.getCache("testCache");
```

As shown below, you can also create a new cache with a specified configuration and add the cache to a CacheManager. Note that when you create a new cache, it is not usable until it has been added to a CacheManager.

```
CacheManager singletonManager = CacheManager.create();
Cache memoryOnlyCache = new Cache("testCache", 5000, false, false, 5, 2);
singletonManager.addCache(memoryOnlyCache);
Cache test = singletonManager.getCache("testCache");
```

Below is another way to create a new cache with a specified configuration. This example creates a cache called *testCache* and adds it CacheManager called *manager* .

```
//Create a singleton CacheManager using defaults
CacheManager manager = CacheManager.create();
//Create a Cache specifying its configuration.
Cache testCache = new Cache(
  new CacheConfiguration("testCache", maxEntriesLocalHeap)
    .memoryStoreEvictionPolicy(MemoryStoreEvictionPolicy.LFU)
    .eternal(false)
    .timeToLiveSeconds(60)
    .timeToIdleSeconds(30)
    .diskExpiryThreadIntervalSeconds(0)
    .persistence(new PersistenceConfiguration().strategy(Strategy.LOCALTEMPSWAP)));
  manager.addCache(testCache);
```

For a full list of parameters for a new Cache, see the Cache constructor at http://ehcache.org/xref/net/sf/ehcache/Cache.html.

Removing Caches Programmatically

The following removes the cache called *sampleCache1* :

```
CacheManager singletonManager = CacheManager.create();
singletonManager.removeCache("sampleCache1");
```

Performing Basic Cache Operations

The following examples refer to *manager* , which is a reference to a CacheManager that contains a cache called *sampleCache1* .

Obtaining a reference to a Cache

The following obtains a Cache called *sampleCache1* , which has been preconfigured in the configuration file

```
Cache cache = manager.getCache("sampleCache1");
```

Putting an Element in Cache

The following puts an element into a cache

```
Cache cache = manager.getCache("sampleCache1");
Element element = new Element("key1", "value1");
cache.put(element);
```

Updating and Element in Cache

The following updates an element in a cache. Even though cache.put( ) is used, Ehcache knows there is an existing element, and considers the put operation as an update for the purpose of notifying cache listeners.

```
Cache cache = manager.getCache("sampleCache1");
cache.put(new Element("key1", "value1"));
//This updates the entry for "key1"
cache.put(new Element("key1", "value2"));
```

Getting an Element from Cache

The following gets a Serializable value from an element with a key of *key1* .

```
Cache cache = manager.getCache("sampleCache1");
Element element = cache.get("key1");
Serializable value = element.getValue();
```

The following gets a NonSerializable value from an element with a key of *key1* .

```
Cache cache = manager.getCache("sampleCache1");
Element element = cache.get("key1");
Object value = element.getObjectValue();
```

Removing an Element from Cache

The following removes an element with a key of *key1* .

```
Cache cache = manager.getCache("sampleCache1");
cache.remove("key1");
```

Obtaining Cache Sizes

The following gets the number of elements currently in the cache.

```
Cache cache = manager.getCache("sampleCache1");
int elementsInMemory = cache.getSize();
```

The following gets the number of elements currently in the MemoryStore.

```
Cache cache = manager.getCache("sampleCache1");
long elementsInMemory = cache.getMemoryStoreSize();
```

The following gets the number of elements currently in the DiskStore.

```
Cache cache = manager.getCache("sampleCache1");
long elementsInMemory = cache.getDiskStoreSize();
```

Shut down the CacheManager

You should shut down a CacheManager after use. It does have a shut-down hook, but it is a best practice to shut it down in your code.

The following shuts down the singleton CacheManager:

```
CacheManager.getInstance().shutdown();
```

The following shuts down a CacheManager instance, assuming you have a reference to the CacheManager called *manager* :

```
manager.shutdown();
```

For additional examples, see CacheManagerTest at http://ehcache.org/xref-test/net/sf/ehcache/CacheManagerTest.html.

JUnit Tests

Ehcache comes with a comprehensive JUnit test suite, which not only tests the code, but shows you how to use the Ehcache API.

You can browse the available tests here: http://ehcache.org/xref-test/index.html. The unit tests are also available in the src.zip in the Ehcache tarball.

3 Cache Usage Patterns

Supported Usage Patterns

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

- "Cache-aside" on page 18 (or direct manipulation)
- "Cache-as-sor" on page 19 (a combination of read-through and write-through or write-behind patterns)
- "Read-through" on page 20
- "Write-through" on page 20
- "Write-behind" on page 21 (or write-back)
- "Copy cache" on page 21

cache-aside

With the cache-aide 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 with the system-of-record. This results in code that often looks like the following pseudo-code:

```
public class MyDataAccessClass
{
  private final Ehcache cache;
  public MyDataAccessClass(Ehcache cache)
  {
    this.cache = cache;
  }
  /* read some data, check cache first, otherwise read from sor */
  public V readSomeData(K key)
  {
     Element element;
     if ((element = cache.get(key)) != null) {
         return element.getValue();
     }
      // note here you should decide whether your cache
     // will cache 'nulls' or not
     if (value = readDataFromDataStore(key)) != null) {
         cache.put(new Element(key, value));
     }
     return value;
  }
  /* write some data, write to sor, then update cache */
  public void writeSomeData(K key, V value)
```

```
  {
     writeDataToDataStore(key, value);
     cache.put(new Element(key, value);
  }
```

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 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)
- Choice of write-through or write-behind strategies on a per-cache basis (use only configuration)
- Allows the cache to solve the "thundering-herd" problem

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

- Less directly visible code-path

cache-as-sor example

```
public class MyDataAccessClass
{
private final Ehcache cache;
public MyDataAccessClass(Ehcache cache)
{
   cache.registerCacheWriter(new MyCacheWriter());
   this.cache = new SelfPopulatingCache(cache);
}
/* read some data - notice the cache is treated as an SOR.
* the application code simply assumes the key will always be available
*/
public V readSomeData(K key)
{
   return cache.get(key);
}
/* write some data - notice the cache is treated as an SOR, it is
* the cache's responsibility to write the data to the SOR.
*/
public void writeSomeData(K key, V value)
{
   cache.put(new Element(key, value);
}
/**
* Implement the CacheEntryFactory that allows the cache to provide
* the read-through strategy
*/
```

```
private class MyCacheEntryFactory implements CacheEntryFactory
{
   public Object createEntry(Object key) throws Exception
   {
       return readDataFromDataStore(key);
   }
}
/**
* Implement the CacheWriter interface which allows the cache to provide
* the write-through or write-behind strategy.
*/
private class MyCacheWriter implements CacheWriter
   public CacheWriter clone(Ehcache cache) throws CloneNotSupportedException;
   {
       throw new CloneNotSupportedException();
   }
    public void init() { }
   void dispose() throws CacheException { }
    void write(Element element) throws CacheException;
   {
       writeDataToDataStore(element.getKey(), element.getValue());
   }
    void writeAll(Collection<Element> elements) throws CacheException
   {
       for (Element element : elements) {
           write(element);
       }
   }
    void delete(CacheEntry entry) throws CacheException
   {
       deleteDataFromDataStore(element.getKey());
   }
    void deleteAll(Collection<CacheEntry> entries) throws CacheException
   {
       for (Element element : elements) {
           delete(element);
       }
   }
}
}
```

read-through

The read-through pattern mimics the structure of the cache-aside pattern when reading data. The difference is that you must implement the CacheEntryFactory interface to instruct the cache how to read objects on a cache miss, and you must wrap the Cache instance with an instance of SelfPopulatingCache.

write-through

The write-through pattern mimics the structure of the cache-aside pattern when writing data. The difference is that you must implement the CacheWriter interface and configure the cache for write-through mode.

A write-through cache writes data to the system-of-record in the same thread of execution. Therefore, in the common scenario of using a database transaction in context of the thread, the write to the database is covered by the transaction in scope. For more details (including configuration settings) about using the write-through pattern, see "Write-Through and Write-Behind Caches" on page 55.

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 in the same thread of execution, write-behind queues the data for write at a later time.

The consequences of the change from write-through to write-behind are that the data write using write-behind will occur outside of the scope of the transaction.

This often-times means that a new transaction must be created to commit the data to the system-of-record. That transaction is separate from the main transaction. For more details (including configuration settings) about using the write-behind pattern, see "Write-Through and Write-Behind Caches" on page 55.

Copy Cache

A copy cache can have two behaviors: it can copy Element instances it returns, when `copyOnRead` is true and copy elements it stores, when `copyOnWrite` to true.

A copy-on-read cache can be useful when you can't let multiple threads access the same Element instance (and the value it holds) concurrently. For example, where the programming model doesn't allow it, or you want to isolate changes done concurrently from each other.

Copy on write also lets you determine exactly what goes in the cache and when (i.e., when the value that will be in the cache will be in state it was when it actually was put in cache). *All mutations to the value, or the element, after the put operation will not be reflected in the cache*.

A concrete example of a copy cache is a Cache configured for XA. It will always be configured `copyOnRead` and `copyOnWrite` to provide proper transaction isolation and clear transaction boundaries (the state the objects are in at commit time is the state making it into the cache). By default, the copy operation will be performed using standard Java object serialization. For some applications, however, this might not be good (or fast) enough. You can configure your own CopyStrategy, which will be used to perform these copy operations. For example, you could easily implement use cloning rather than Serialization.

For more information about copy caches, see “Passing Copies Instead of References” in the *Configuration Guide* for Ehcache.

4 Searching a Cache

About Searching

The Search API allows you to execute arbitrarily complex queries against caches. The development of alternative indexes on values provides the ability for data to be looked up based on multiple criteria instead of just keys.

Note: Terracotta BigMemory Go and BigMemory Max products use indexing. The Search API queries open-source Ehcache using a direct search method. For more information about indexing, see "Best Practices for Optimizing Searches" on page 34.

Searchable attributes can be extracted from both keys and values. Keys, values, or summary values (Aggregators) can all be returned. Here is a simple example: Search for 32-year-old males and return the cache values.

```
Results results = cache.createQuery().includeValues()
  .addCriteria(age.eq(32).and(gender.eq("male"))).execute();
```

You can formulate queries using the Search API.

```
// BigMemory Search API:
Attribute<Integer> age = cache.getSearchAttribute("age");
Person.createQuery().addCriteria(age.gt(30)).includeValues().execute();
```

Before creating a query, the cache configuration must be prepared as described in "Making a Cache Searchable" on page 24.

- For more information about creating queries using the Search API, see "Creating a Query" on page 29.

What is Searchable?

Searches can be performed against Element keys and values, but they must be treated as attributes. Some Element keys and values are directly searchable and can simply be added to the search index as attributes. Some Element keys and values must be made searchable by extracting attributes with supported search types out of the keys and values. It is the attributes themselves that are searchable.

Making a Cache Searchable

Caches can be made searchable, on a per cache basis, either by configuration or programmatically.

By Configuration

Caches are made searchable by adding a `<searchable/>` tag to the cache definition in the ehcache.xml file.

```
<cache name="cache2" maxBytesLocalHeap="16M" eternal="true"
maxBytesLocalOffHeap="256M">
    <persistence strategy="localRestartable"/>
```

```
    <searchable/>
</cache>
```

This configuration will scan keys and values and, if they are of supported search types, add them as attributes called “key” and “value,” respectively. If you do not want automatic indexing of keys and values, you can disable it using:

```
<cache name="cacheName" ...>
    <searchable keys="false" values="false">
       ...
    </searchable>
</cache>
```

You might want to do this if you have a mix of types for your keys or values. The automatic indexing will throw an exception if types are mixed.

If you think that you will want to add search attributes after the cache is initialized, you can explicitly indicate the dynamic search configuration. Set the `allowDynamicIndexing` attribute to “true” to enable use of the Dynamic Attributes extractor. For more information about the Dynamic Attributes extractor, see "Defining Attributes" on page 26.

```
<cache name="cacheName" ...>
    <searchable allowDynamicIndexing="true">
       ...
    </searchable>
</cache>
```

Often keys or values will not be directly searchable and instead you will need to extract searchable attributes from the keys or values. The following example shows a more typical case. Attribute extractors are explained in more detail in "Defining Attributes" on page 26.

```
<cache name="cache3" maxEntriesLocalHeap="10000" eternal="true"
maxBytesLocalOffHeap="10G">
    <persistence strategy="localRestartable"/>
    <searchable>
        <searchAttribute name="age" class="net.sf.ehcache.search.
        TestAttributeExtractor"/>
        <searchAttribute name="gender" expression="value.getGender()"/>
    </searchable>
</cache>
```

Programmatically

The following example shows how to programmatically create the cache configuration with search attributes.

```
Configuration cacheManagerConfig = new Configuration();
CacheConfiguration cacheConfig = new CacheConfiguration("myCache", 0).eternal(true);
Searchable searchable = new Searchable();
cacheConfig.addSearchable(searchable);
// Create attributes to use in queries.
searchable.addSearchAttribute(new SearchAttribute().name("age"));
// Use an expression for accessing values.
searchable.addSearchAttribute(new SearchAttribute()
    .name("first_name")
    .expression("value.getFirstName()"));
searchable.addSearchAttribute(new SearchAttribute()
     .name("last_name")
 .expression("value.getLastName()"));
```

```
searchable.addSearchAttribute(new SearchAttribute()
     .name("zip_code")
 .className("net.sf.ehcache.search.TestAttributeExtractor"));
cacheManager = new CacheManager(cacheManagerConfig);
cacheManager.addCache(new Cache(cacheConfig));
Ehcache myCache = cacheManager.getEhcache("myCache");
// Now create the attributes and queries, then execute.
...
```

To learn more about the Search API, see the net.sf.ehcache.search* packages in the Ehcache Javadoc.

Defining Attributes

In addition to configuring a cache to be searchable, you must define the attributes to be used in searches.

Attributes are extracted from keys or values during search by using AttributeExtractors. An extracted attribute must be one of the following types:

- Boolean
- Byte
- Character
- Double
- Float
- Integer
- Long
- Short
- String
- java.util.Date
- java.sql.Date
- Enum

These types correspond to the AttributeType enum specified by the Ehcache Javadoc at http://ehcache.org/apidocs/2.10/net/sf/ehcache/search/attribute/AttributeType.html.

Type name matching is case sensitive. For example, Double resolves to the java.lang.Double class type, and double is interpreted as the primitive double type.

Search API Example

```
<searchable>
<searchAttribute name="age" type="Integer"/>
</searchable>
```

If an attribute cannot be found or is of the wrong type, an AttributeExtractorException is thrown on search execution. !

Note: On the first use of an attribute, the attribute type is detected, validated against supported types, and saved automatically. Once the type is established, it cannot be changed. For example, if an integer value was initially returned for attribute named "Age" by the attribute extractor, it is an error for the extractor to return a float for this attribute later on.

Well-known Attributes

The parts of an Element that are well-known attributes can be referenced by some predefined, well-known names. If a key and/or value is of a supported search type, it is added automatically as an attribute with the name "key" or "value." These well-known attributes have the convenience of being constant attributes made available in the Query class. For example, the attribute for "key" can be referenced in a query by `Query.KEY`. For even greater readability, statically import so that, in this example, you would use `KEY`.

Well-known Attribute Name	Attribute Constant
key	Query.KEY
value	Query.VALUE

Reflection Attribute Extractor

The ReflectionAttributeExtractor is a built-in search attribute extractor that uses JavaBean conventions and also understands a simple form of expression. Where a JavaBean property is available and it is of a searchable type, it can be declared:

```
<cache>
  <searchable>
    <searchAttribute name="age"/>
  </searchable>
</cache>
```

The expression language of the ReflectionAttributeExtractor also uses method/value dotted expression chains. The expression chain must start with "key", "value", or "element". From the starting object, a chain of method calls or field names follows. Method calls and field names can be freely mixed in the chain:

```
<cache>
  <searchable>
    <searchAttribute name="age" expression="value.person.getAge()"/>
  </searchable>
</cache>
<cache>
  <searchable>
     <searchAttribute name="name" expression="element.toString()"/>
  </searchable>
</cache>
```

Note: The method and field name portions of the expression are case-sensitive.

Custom Attribute Extractor

In more complex situations, you can create your own attribute extractor by implementing the AttributeExtractor interface. The interface's attributeFor( ) method returns the attribute value for the element and attribute name you specify.

Note: These examples assume there are previously created Person objects containing attributes such as name, age, and gender.

Provide your extractor class:

```
<cache name="cache2" maxEntriesLocalHeap="0" eternal="true">
  <persistence strategy="none"/>
  <searchable>
     <searchAttribute name="age" class="net.sf.ehcache.search.TestAttributeExtractor"/>
  </searchable>
</cache>
```

A custom attribute extractor could be passed an Employee object to extract a specific attribute:

```
returnVal = employee.getdept();
```

If you need to pass state to your custom extractor, specify properties:

```
<cache>
  <searchable>
    <searchAttribute name="age"
    class="net.sf.ehcache.search.TestAttributeExtractor"
    properties="foo=this,bar=that,etc=12" />
  </searchable>
</cache>
```

If properties are provided, the attribute extractor implementation must have a public constructor that accepts a single java.util.Properties instance.

Dynamic Attributes Extractor

The DynamicAttributesExtractor provides flexibility by allowing the search configuration to be changed after the cache is initialized. This is done with one method call, at the point of element insertion into the cache. The DynamicAttributesExtractor method returns a map of attribute names to index and their respective values. This method is called for every Ehcache.put( ) and replace( ) invocation.

Assuming that we have previously created Person objects containing attributes such as name, age, and gender, the following example shows how to create a dynamically searchable cache and register the DynamicAttributesExtractor:

```
Configuration config = new Configuration();
config.setName("default");
CacheConfiguration cacheCfg = new CacheConfiguration("PersonCache");
cacheCfg.setEternal(true);
cacheCfg.terracotta(new TerracottaConfiguration().clustered(true));
Searchable searchable = new Searchable().allowDynamicIndexing(true);
cacheCfg.addSearchable(searchable);
config.addCache(cacheCfg);
CacheManager cm = new CacheManager(config);
Ehcache cache = cm.getCache("PersonCache");
final String attrNames[] = {"first_name", "age"};
```

```
// Now you can register a dynamic attribute extractor to index
// the cache elements, using a subset of known fields
cache.registerDynamicAttributesExtractor(new DynamicAttributesExtractor() {
    Map<String, Object> attributesFor(Element element) {
        Map<String, Object> attrs = new HashMap<String, Object>();
        Person value = (Person)element.getObjectValue();
        // For example, extract first name only
        String fName = value.getName() == null ? null : value.getName().
        split("\\s+")[0];
        attrs.put(attrNames[0], fName);
        attrs.put(attrNames[1], value.getAge());
        return attrs;
    }
});
// Now add some data to the cache
cache.put(new Element(10, new Person("John Doe", 34, Person.Gender.MALE)));
```

Given the code above, the newly put element would be indexed on values of name and age fields, but not gender. If, at a later time, you would like to start indexing the element data on gender, you would need to create a new DynamicAttributesExtractor instance that extracts that field for indexing.

Dynamic Search Rules

- To use the DynamicAttributesExtractor, the cache must be configured to be searchable and dynamically indexable. For information about making a cache searchable, see "Making a Cache Searchable" on page 24.
- A dynamically searchable cache must have a dynamic extractor registered BEFORE data is added to it. (This is to prevent potential races between extractor registration and cache loading which might result in an incomplete set of indexed data, leading to erroneous search results.)
- Each call on the DynamicAttributesExtractor method replaces the previously registered extractor, because there can be at most one extractor instance configured for each such cache.
- If a dynamically searchable cache is initially configured with a predefined set of search attributes, this set of attributes is always be queried for extracted values, regardless of whether or not a dynamic search attribute extractor has been configured.
- The initial search configuration takes precedence over dynamic attributes, so if the dynamic attribute extractor returns an attribute name already used in the initial searchable configuration, an exception is thrown.

Creating a Query

Ehcache uses a fluent, object-oriented query API, following the principles of a Domain-Specific Language (DSL), which should be familiar to Java programmers. For example:

```
Query query = cache.createQuery().addCriteria(age.eq(35)).includeKeys().end();
Results results = query.execute();
```

Using Attributes in Queries

If declared and available, the well-known attributes are referenced by their names or the convenience attributes are used directly:

```
Results results = cache.createQuery().addCriteria(Query.KEY.eq(35)).execute();
Results results = cache.createQuery().addCriteria(Query.VALUE.lt(10)).execute();
```

Other attributes are referenced by the names in the configuration:

```
Attribute<Integer> age = cache.getSearchAttribute("age");
Attribute<String> gender = cache.getSearchAttribute("gender");
Attribute<String> name = cache.getSearchAttribute("name");
```

Expressions

A Query is built up using Expressions. Expressions can include logical operators such as <and> and <or>, and comparison operators such as <ge> (>=), <between>, and <like>. The configuration, `addCriteria(...)`, is used to add a clause to a query. Adding a further clause automatically "<and>s" the clauses.

```
query = cache.createQuery().includeKeys()
             .addCriteria(age.le(65))
     .add(gender.eq("male"))
     .end();
```

Both logical and comparison operators implement the Criteria interface. To add a criterion with a different logical operator, explicitly nest it within a new logical operator Criteria object. For example, to check for age = 35 or gender = female:

```
query.addCriteria(new Or(age.eq(35),
             gender.eq(Gender.FEMALE))
            );
```

More complex compound expressions can be created through additional nesting. For a complete list of expressions, see the Expression Javadoc at http://ehcache.org/xref/net/sf/ehcache/search/expression/package-frame.html.

List of Operators

Operators are available as methods on attributes, so they are used by adding a ".". For example, "lt" means "less than" and is used as `age.lt(10)`, which is a shorthand way of saying `age LessThan(10)`.

Shorthand	Criteria Class	Description
and	And	The Boolean AND logical operator.
between	Between	A comparison operator meaning between two values.
eq	EqualTo	A comparison operator meaning Java "equals to" condition.

Shorthand	Criteria Class	Description
gt	GreaterThan	A comparison operator meaning greater than.
ge	GreaterThanOrEqual	A comparison operator meaning greater than or equal to.
in	InCollection	A comparison operator meaning in the collection given as an argument.
lt	LessThan	A comparison operator meaning less than.
le	LessThanOrEqual	A comparison operator meaning less than or equal to.
ilike	ILike	A regular expression matcher. “?” and “*” may be used. Note that placing a wildcard in front of the expression will cause a table scan. ILike is always case insensitive.
isNull	IsNull	Tests whether the value of an attribute with given name is null.
notNull	NotNull	Tests whether the value of an attribute with given name is NOT null.
not	Not	The Boolean NOT logical operator,
ne	NotEqualTo	A comparison operator meaning not the Java “equals to” condition,
or	Or	The Boolean OR logical operator.

Note: For Strings, the operators are case-insensitive.

Making Queries Immutable

By default, a query can be executed, modified, and re-executed. If end( ) is called, the query is made immutable.

Obtaining and Organizing Query Results

Queries return a Results object that contains a list of objects of class Result. Each element in the cache that a query finds is represented as a Result object. For example, if a query finds 350 elements, there will be 350 Result objects. However, if no keys or attributes are included but aggregators are included, there is exactly one Result present.

A Result object can contain:

- The Element key - when `includeKeys()` is added to the query,
- The Element value - when `includeValues()` is added to the query,
- Predefined attribute(s) extracted from an Element value - when `includeAttribute(...)` is added to the query. To access an attribute from a Result, use `getAttribute(Attribute<T> attribute)`.
- Aggregator results - Aggregator results are summaries computed for the search. They are available through Result.getAggregatorResults, which returns a list of Aggregators in the same order in which they were used in the query.

Aggregators

Aggregators are added with `query.includeAggregator(\<attribute\>.\<aggregator\>)`. For example, to find the sum of the age attribute:

```
query.includeAggregator(age.sum());
```

For a complete list of aggregators, see Aggregators in the Ehcache Javadoc.

Ordering Results

Query results can be ordered in ascending or descending order by adding an addOrderBy clause to the query. This clause takes as parameters the attribute to order by and the ordering direction. For example, to order the results by ages in ascending order:

```
query.addOrderBy(age, Direction.ASCENDING);
```

Grouping Results

Query query results can be grouped similarly to using an SQL GROUP BY statement. The GroupBy feature provides the option to group results according to specified attributes. You can add an addGroupBy clause to the query, which takes as parameters the attributes to group by. For example, you can group results by department and location:

```
Query q = cache.createQuery();
Attribute<String> dept = cache.getSearchAttribute("dept");
Attribute<String> loc = cache.getSearchAttribute("location");
q.includeAttribute(dept);
q.includeAttribute(loc);
q.addCriteria(cache.getSearchAttribute("salary").gt(100000));
q.includeAggregator(Aggregators.count());
q.addGroupBy(dept, loc);
```

The GroupBy clause groups the results from includeAttribute( ) and allows aggregate functions to be performed on the grouped attributes. To retrieve the attributes that are associated with the aggregator results, you can use:

```
String dept = singleResult.getAttribute(dept);
String loc = singleResult.getAttribute(loc);
```

GroupBy Rules

Grouping query results adds another step to the query. First, results are returned, and second the results are grouped. Note the following rules and considerations:

- In a query with a GroupBy clause, any attribute specified using includeAttribute( ) should also be included in the GroupBy clause.
- Special KEY or VALUE attributes cannot be used in a GroupBy clause. This means that includeKeys( ) and includeValues( ) cannot be used in a query that has a GroupBy clause.
- Adding a GroupBy clause to a query changes the semantics of any aggregators passed in, so that they apply only within each group.
- As long as there is at least one aggregation function specified in a query, the grouped attributes are not required to be included in the result set, but they are typically requested anyway to make result processing easier.
- An addCriteria clause applies to all results prior to grouping.
- If OrderBy is used with GroupBy, the ordering attributes are limited to those listed in the GroupBy clause.

Limiting the Size of Results

By default, a query can return an unlimited number of results. For example, the following query will return all keys in the cache.

```
Query query = cache.createQuery();
query.includeKeys();
query.execute();
```

If too many results are returned, it could cause an OutOfMemoryError The maxResults clause is used to limit the size of the results. For example, to limit the above query to the first 100 elements found:

```
Query query = cache.createQuery();
query.includeKeys();
query.maxResults(100);
query.execute();
```

Note: When maxResults is used with GroupBy, it limits the number of groups.

When you are done with the results, call the discard( ) method to free up resources.

For additional information about managing large result sets, see the topics that relate to pagination in "Best Practices for Optimizing Searches" on page 34.

Interrogating Results

To determine what a query returned, use one of the interrogation methods on Results.:

- hasKeys( )
- hasValues( )
- hasAttributes( )
- hasAggregators( )

Best Practices for Optimizing Searches

- Construct searches by including only the data that is actually required.
  - Only use includeKeys( ) and/or includeAttribute( ) if those values are required for your application logic.
  - If you do not need values or attributes, be careful not to burden your queries with unnecessary work. For example, if result.getValue( ) is not called in the search results, do not use includeValues( ) in the query.
  - Consider if it would be sufficient to get attributes or keys on demand. For example, instead of running a search query with includeValues( ) and then result.getValue( ), run the query for keys and include cache.get( ) for each individual key.

  Note: The includeKeys( ) and includeValues( ) methods have lazy deserialization, meaning that keys and values are deserialized only when result.getKey( ) or result.getValue( ) is called. However, calls to includeKeys() and includeValues( ) do take time, so consider carefully when constructing your queries.
- Searchable keys and values are automatically indexed by default. If you are not including them in your query, turn off automatic indexing with the following:

```
<cache name="cacheName" ...>
  <searchable keys="false" values="false"/>
  ...
  </searchable>
</cache>
```

- Limit the size of the result set. Depending on your use case, you might consider maxResults, an Aggregator, or pagination:
- If getting a subset of all the possible results quickly is more important than receiving all the results, consider using query.maxResults(int number_of_results). Sometimes maxResults is useful where the result set is ordered such that the items you want most are included within the maxResults.

- If all you want is a summary statistic, use a built-in Aggregator function, such as count(). For details, see the net.sf.ehcache.search.aggregator package in the Ehcache Javadoc.
- Make your search as specific as possible.
  - Queries with iLike criteria and fuzzy (wildcard) searches might take longer than more specific queries.
  - If you are using a wildcard, try making it the trailing part of the string instead of the leading part (`"321*"` instead of `"*123"`).

    Tip: If you want leading wildcard searches, you should create a <searchAttribute> with the string value reversed in it, so that your query can use the trailing wildcard instead.
- When possible, use the query criteria "Between" instead of "LessThan" and "GreaterThan", or "LessThanOrEqual" and "GreaterThanOrEqual". For example, instead of using `le(startDate)` and `ge(endDate)`, try `not(between(startDate,endDate))`.
- Index dates as integers. This can save time and can also be faster if you have to do a conversion later on.

Concurrency Considerations

Unlike cache operations, which have selectable concurrency control or transactions, queries are asynchronous and search results are "eventually consistent" with the caches.

Index Updating

Although indexes are updated synchronously, their state lags slightly behind that of the cache. The only exception is when the updating thread performs a search.

For caches with concurrency control, an index does not reflect the new state of the cache until:

- The change has been applied to the cluster.
- For a cache with transactions, when commit has been called.

Query Results

Unexpected results might occur if:

- A search returns an Element reference that no longer exists.
- Search criteria select an Element, but the Element has been updated.
- Aggregators, such as sum( ), disagree with the same calculation done by redoing the calculation yourself by re-accessing the cache for each key and repeating the calculation.

- A value reference refers to a value that has been removed from the cache, and the cache has not yet been reindexed. If this happens, the value is null but the key and attributes supplied by the stale cache index are non-null. Because values in a cache are also allowed to be null, you cannot tell whether your value is null because it has been removed from the cache after the index was last updated or because it is a null value.

Recommendations

Because the state of the cache can change between search executions, the following is recommended:

- Add all of the aggregators you want for a query at once, so that the returned aggregators are consistent.
- Use null guards when accessing a cache with a key returned from a search.

Options for Working with Nulls

The Search API supports using the presence of a null as search criteria.

```
myQuery.addCriteria(cache.getAttribute("middle_name").isNull());
```

It also supports using the absence of a null as search criteria:

```
myQuery.addCriteria(cache.getAttribute("middle_name").notNull());
```

Which is equivalent to:

```
myQuery.addCriteria(cache.getAttribute("middle_name").isNull().not());
```

Alternatively, you can call constructors to set up equivalent logic:

```
Criteria isNull = new IsNull("middle_name");
Criteria notNull = new NotNull("middle_name");
```

5 Using Explicit Locking

About Explicit Locking

Ehcache contains an implementation which provides for explicit locking, using read and write locks.

With explicit locking, it is possible to get more control over Ehcache 's locking behavior to allow business logic to apply an atomic change with guaranteed ordering across one or more keys in one or more caches. It can therefore be used as a custom alternative to XA Transactions or Local transactions.

With that power comes a caution. It is possible to create deadlocks in your own business logic using this API.

Code Sample for Explicit Locking

The following is a simple example that shows how to use explicit locking.

```
String key = "123";
Foo val = new Foo();
cache.acquireWriteLockOnKey(key);
try {
    cache.put(new Element(key, val));
} finally {
    cache.releaseWriteLockOnKey(key);
}
...sometime later
String key = "123";
cache.acquireWriteLockOnKey(key);
try {
    Object cachedVal = cache.get(key).getValue();
    cachedVal.setSomething("abc");
    cache.put(new Element(key, cachedVal));
 } finally {
cache.releaseWriteLockOnKey(key);
 }
```

How Locking Works

A READ lock does not prevent other READers from also acquiring a READ lock and reading. A READ lock cannot be obtained if there is an outstanding WRITE lock. It will queue.

A WRITE lock cannot be obtained while there are outstanding READ locks. It will queue.

In each case the lock should be released after use to avoid locking problems. The lock release should be in a “finally” block.

If before each read you acquire a READ lock and then before each write you acquire a WRITE lock, then an isolation level akin to READ_COMMITTED is achieved.

The Locking API

The following methods are available on Cache and Ehcache.

```
/**
* Acquires the proper read lock for a given cache key
*
* @param key - The key that retrieves a value that you want to protect via
* locking.
*/
public void acquireReadLockOnKey(Object key) {
   this.acquireLockOnKey(key, LockType.READ);
}
/**
* Acquires the proper write lock for a given cache key
*
* @param key - The key that retrieves a value that you want to protect via
* locking.
*/
public void acquireWriteLockOnKey(Object key) {
   this.acquireLockOnKey(key, LockType.WRITE);
}
/**
* Try to get a read lock on a given key. If can't get it in timeout millis
* then return a boolean telling that it didn't get the lock
*
* @param key - The key that retrieves a value that you want to protect via
* locking.
* @param timeout - millis until giveup on getting the lock
* @return whether the lock was awarded
* @throws InterruptedException
*/
public boolean tryReadLockOnKey(Object key, long timeout) throws
InterruptedException {
   Sync s = getLockForKey(key);
   return s.tryLock(LockType.READ, timeout);
}
/**
* Try to get a write lock on a given key. If can't get it in timeout millis
* then return a boolean telling that it didn't get the lock
*
* @param key - The key that retrieves a value that you want to protect via
* locking.
* @param timeout - millis until giveup on getting the lock
* @return whether the lock was awarded
* @throws InterruptedException
*/
public boolean tryWriteLockOnKey(Object key, long timeout) throws
InterruptedException {
   Sync s = getLockForKey(key);
   return s.tryLock(LockType.WRITE, timeout);
}
/**
* Release a held read lock for the passed in key
*
* @param key - The key that retrieves a value that you want to protect via
* locking.
*/
public void releaseReadLockOnKey(Object key) {
   releaseLockOnKey(key, LockType.READ);
```

```
}
/**
* Release a held write lock for the passed in key
*
* @param key - The key that retrieves a value that you want to protect via
* locking.
*/
public void releaseWriteLockOnKey(Object key) {
   releaseLockOnKey(key, LockType.WRITE);
}
/**
* Returns true if a read lock for the key is held by the current thread
*
* @param key
* @return true if a read lock for the key is held by the current thread
*/
boolean isReadLockedByCurrentThread(Object key);
/**
* Returns true if a write lock for the key is held by the current thread
*
* @param key
* @return true if a write lock for the key is held by the current thread
*/
boolean isWriteLockedByCurrentThread(Object key);
```

Supported Topologies

Except as noted in the "The Locking API" on page 39, the locking API supports the standalone and distributed cache topologies. It does not support the replicated topology.

6 Blocking and Self Populating Caches

About Blocking and Self-Populating Caches

The net.sf.ehcache.constructs package contains some applied caching classes which use the core classes to solve everyday caching problems. Two of these are BlockingCache and SelfPopulatingCache.

Blocking Cache

Imagine you have a very busy web site with thousands of concurrent users. Rather than being evenly distributed in what they do, they tend to gravitate to popular pages. These pages are not static, they have dynamic data which goes stale in a few minutes. Or imagine you have collections of data which go stale in a few minutes. In each case the data is extremely expensive to calculate. If each request thread asks for the same thing, that is a lot of work. Now, add a cache. Get each thread to check the cache; if the data is not there, go and get it and put it in the cache.

Now, imagine that there are so many users contending for the same data that in the time it takes the first user to request the data and put it in the cache, ten other users have done the same thing. The upstream system, whether a JSP or velocity page, or interactions with a service layer or database are doing ten times more work than they need to. Enter the BlockingCache. It is blocking because all threads requesting the same key wait for the first thread to complete. Once the first thread has completed the other threads simply obtain the cache entry and return. The BlockingCache can scale up to very busy systems. Each thread can either wait indefinitely, or you can specify a timeout using the `timeoutMillis` constructor argument.

For more information, see the Javadoc for BlockingCache.

SelfPopulatingCache

Sometimes, you want to use the BlockingCache, but the requirement to always release the lock results in complicated code. You also want to think about what you are doing without thinking about the caching. Enter the SelfPopulatingCache.

SelfPopulatingCache is synonymous with *pull-through cache*, which is a common caching term. However, SelfPopulatingCache is always used in addition to a BlockingCache.

SelfPopulatingCache uses a CacheEntryFactory which, given a key, knows how to populate the entry.

Note: JCache inspired getWithLoader and getAllWithLoader directly in Ehcache, which work with a CacheLoader may be used as an alternative to SelfPopulatingCache.

For more information, see the Javadoc for SelfPopulatingCache.

7 Transaction Support

About Transaction Support

Transactions are supported in versions of Ehcache 2.0 and higher. The 2.3.x or lower releases only support XA. However since ehcache 2.4 support for both Global Transactions with xa_strict and xa modes, and Local Transactions with local mode has been added.

All or Nothing

If a cache is enabled for transactions, all operations on it must happen within a transaction context otherwise a TransactionException will be thrown.

Change Visibility

The isolation level offered by Ehcache is READ_COMMITTED. Ehcache can work as an XAResource, in which case, full two-phase commit is supported. Specifically:

- All mutating changes to the cache are transactional including put(), remove(), putWithWriter(), removeWithWriter(), and removeAll().
- Mutating changes are not visible to other transactions in the local JVM or across the cluster until COMMIT has been called.
- Until then, reads such as by cache.get(...) by other transactions return the old copy. Reads do not block.

Transactional Modes

Transactional modes enable you to perform atomic operations on your caches and other data stores.

- local — When you want your changes across multiple caches to be performed atomically. Use this mode when you need to update your caches atomically. That is, you can have all your changes be committed or rolled back using a straightforward API. This mode is most useful when a cache contains data calculated from other cached data.
- xa — Use this mode when you cache data from other data stores, such as a DBMS or JMS, and want to do it in an atomic way under the control of the JTA API ("Java Transaction API") but without the overhead of full two-phase commit. In this mode, your cached data can get out of sync with the other resources participating in the transactions in case of a crash. Therefore, only use this mode if you can afford to live with stale data for a brief period of time.
- xa_strict — Similar to "xa" but use it only if you need strict XA disaster recovery guarantees. In this mode, the cached data can never get out of sync with the other resources participating in the transactions, even in case of a crash. However, to get that extra safety the performance decreases significantly.

Requirements for Transactional Caches

The objects you store in your transactional cache must:

- Implement java.io.Serializable—This is required to store cached objects when the cache is distributed in a Terracotta cluster, and it is also required by the copy-on-read / copy-on-write mechanism used to implement isolation.
- Override `equals` and `hashcode` — These must be overridden because the transactional stores rely on element value comparison. See: ElementValueComparator and the `elementValueComparator` configuration setting.

Configuring Transactional Cache

Transactions are enabled on a cache-by-cache basis with the `transactionalMode` cache attribute. The allowed values are:

- xa_strict
- xa
- local
- off

The default value is "off." In the following example, "xa_strict" is enabled.

```
<cache name="xaCache"
   maxEntriesLocalHeap="500"
   eternal="false"
   timeToIdleSeconds="300"
   timeToLiveSeconds="600"
   diskExpiryThreadIntervalSeconds="1"
   transactionalMode="xa_strict">
 </cache>
```

Transactional Caches with Spring

Note the following when using Spring:

- If you access the cache from an @Transactional Spring-annotated method, begin/commit/rollback statements are not required in application code because they are emitted by Spring.
- Both Spring and Ehcache need to access the transaction manager internally, and therefore you must inject your chosen transaction manager into Spring's PlatformTransactionManager as well as use an appropriate lookup strategy for Ehcache.
- The Ehcache default lookup strategy might not be able to detect your chosen transaction manager. For example, it cannot detect the WebSphere transaction

manager. For additional information, see the "Transaction Managers" on page 48.

- Configuring a <tx:method> with `read-only=true` could be problematic with certain transaction managers such as WebSphere.

Working with Global Transactions

Global Transactions are supported by Ehcache . Ehcache can act as an {XAResouce} to participate in JTA transactions under the control of a Transaction Manager. This is typically provided by your application server. However you can also use a third party transaction manager such as Bitronix.

To use Global Transactions, select either “xa_strict” or “xa” mode.

Implementation

Global transactions support is implemented at the Store level, through XATransactionStore and JtaLocalTransactionStore. The former decorates the underlying MemoryStore implementation, augmenting it with transaction isolation and two-phase commit support through an <XAResouce> implementation. The latter decorates a LocalTransactionStore-decorated cache to make it controllable by the standard JTA API instead of the proprietary TransactionController API.

During its initialization, the cache does a lookup of the Transaction Manager using the provided TransactionManagerLookup implementation. Then, using the TransactionManagerLookup.register(XAResouce), the newly created XAResource is registered. The store is automatically configured to copy every element read from the cache or written to it. Cache is copy-on-read and copy-on-write.

Failure Recovery

In support of the JTA specification, only *prepared* transaction data is recoverable. Prepared data is persisted onto the cluster and locks on the memory are held. This means that non-clustered caches cannot persist transaction data. Therefore, recovery errors after a crash might be reported by the transaction manager.

Recovery

At any time after something went wrong, an XAResource might be asked to recover. Data that has been prepared might either be committed or rolled back during recovery. XA data that has not yet been prepared is discarded. The recovery guarantee differs depending on the XA mode.

xa Mode

With “xa” mode, the cache doesn't get registered as an {XAResource} with the Transaction Manager but merely can follow the flow of a JTA transaction by registering a JTA {Synchronization}. The cache can end up inconsistent with the other resources if

there is a JVM crash in the mutating node. In this mode, some inconsistency might occur between a cache and other XA resources (such as databases) after a crash. However, the cache data remains consistent because the transaction is still fully atomic on the cache itself.

xa_strict Mode

With “xa_strict” mode, the cache always responds to the Transaction Manager's recover calls with the list of prepared XIDs of failed transactions. Those transaction branches can then be committed or rolled back by the Transaction Manager. This mode supports the basic XA mechanism of the JTA standard.

Sample Applications

The following sample applications showing how to use XA with a variety of technologies.

XA Sample Application

This sample application uses the JBoss application server. It shows an example using User managed transactions. Although most people use JTA from within a Spring or EJB container rather than managing it themselves, this sample application is useful as a demonstration. The following snippet from our SimpleTX servlet shows a complete transaction.

```
Ehcache cache = cacheManager.getEhcache("xaCache");
UserTransaction ut = getUserTransaction();
printLine(servletResponse, "Hello...");
try {
   ut.begin();
   int index = serviceWithinTx(servletResponse, cache);
   printLine(servletResponse, "Bye #" + index);
   ut.commit();
} catch(Exception e) {
   printLine(servletResponse,
       "Caught a " + e.getClass() + "! Rolling Tx back");
   if(!printStackTrace) {
       PrintWriter s = servletResponse.getWriter();
       e.printStackTrace(s);
       s.flush();
   }
   rollbackTransaction(ut);
}
```

You can download the source code for the sample from Terracotta Forge. The readme file explains how to set up the sample application.

XA Banking Application

This application represents a real-world scenario. A Web application reads <account transfer> messages from a queue and tries to execute the transfers.

With transaction mode enabled, failures are rolled back so that the cached account balance is always the same as the true balance summed from the database. This sample

is a Spring-based web application running in a Jetty container. It has (embedded) the following components:

- A message broker (ActiveMQ)
- 2 databases (embedded Derby XA instances)
- 2 caches (transactional Ehcache)

All XA Resources are managed by Atomikos Transaction Manager. Transaction demarcation is done using Spring AOP's @Transactional annotation. You can run it with: `mvn clean jetty:run`. Then point your browser at: http://localhost:9080. To see what happens without XA transactions: `mvn clean jetty:run -Dxa=no`

You can download the source code for the sample from Terracotta Forge. The readme file explains how to set up the sample application.

Transaction Managers

Automatically Detected Transaction Managers

Ehcache automatically detects and uses the following transaction managers in the order shown below:

- GenericJNDI (e.g. GlassFish, JBoss, JTOM and any others that register themselves in JNDI at the standard location of java:/TransactionManager
- WebLogic (since 2.4.0)
- Bitronix
- Atomikos

No configuration is required. They work out-of-the-box. The first found is used.

Configuring a Transaction Manager

If your transaction manager is not in the list above or you want to change the priority, provide your own lookup class based on an implementation of net.sf.ehcache.transaction.manager.TransactionManagerLookup and specify it in place of the DefaultTransactionManagerLookup in ehcache.xml as shown below.

```
<transactionManagerLookup
  class= "com.mycompany.transaction.manager.MyTransactionManagerLookupClass"
  properties="" propertySeparator=":"/>
```

Another option is to provide a different location for the JNDI lookup by passing the `jndiName` property to the DefaultTransactionManagerLookup. The example below provides the proper location for the transaction manager in GlassFish v3:

```
<transactionManagerLookup
  class="net.sf.ehcache.transaction.manager.DefaultTransactionManagerLookup"
  properties="jndiName=java:appserver/TransactionManager" propertySeparator=";"/>
```

Working with Local Transactions

Local Transactions allow single-phase commit across multiple cache operations, across one or more caches, and in the same CacheManager.This lets you apply multiple changes to a CacheManager all in your own transaction. If you also want to apply changes to other resources, such as a database, open a transaction to them and manually handle commit and rollback to ensure consistency.

Local transactions are not controlled by a transaction manager. Instead there is an explicit API where a reference is obtained to a TransactionController for the CacheManager using cacheManager.getTransactionController() and the steps in the transaction are called explicitly. The steps in a local transaction are:

- `transactionController.begin()` - This marks the beginning of the local transaction on the current thread. The changes are not visible to other threads or to other transactions.
- `transactionController.commit()` - Commits work done in the current transaction on the calling thread.
- `transactionController.rollback()` - Rolls back work done in the current transaction on the calling thread. The changes done since begin are not applied to the cache. These steps should be placed in a try-catch block which catches TransactionException. If any exceptions are thrown, rollback() should be called. Local Transactions has its own exceptions that can be thrown, which are all subclasses of CacheException. They are:
- `TransactionException` - a general exception
- `TransactionInterruptedException` - if Thread.interrupt() was called while the cache was processing a transaction.
- `TransactionTimeoutException` - if a cache operation or commit is called after the transaction timeout has elapsed.

Introductory Video

Ludovic Orban, the primary author of Local Transactions, presents an introductory video on Local Transactions.

Configuration

Local transactions are configured as follows:

```
<cache name="sampleCache"
   ...
   transactionalMode="local"
 </cache>
```

Isolation Level

As with the other transaction modes, the isolation level is READ_COMMITTED.

Transaction Timeouts

If a transaction cannot complete within the timeout period, a TransactionTimeoutException is thrown. To return the cache to a consistent state, call transactionController.rollback(). Because TransactionController is at the level of the CacheManager, a default timeout can be set which applies to all transactions across all caches in a CacheManager. The default is 15 seconds. To change the defaultTimeout:

```
transactionController.setDefaultTransactionTimeout(int defaultTransactionTimeoutSeconds)
```

The countdown starts when begin() is called. You might have another local transaction on a JDBC connection and you might be making multiple changes. If you think it might take longer than 15 seconds for an individual transaction, you can override the default when you begin the transaction with:

```
transactionController.begin(int transactionTimeoutSeconds) {
```

Sample Code

The following example shows a transaction that performs multiple operations across two caches.

```
CacheManager cacheManager = CacheManager.getInstance();
try {
   cacheManager.getTransactionController().begin();
    cache1.put(new Element(1, "one"));
   cache2.put(new Element(2, "two"));
   cache1.remove(4);
    cacheManager.getTransactionController().commit();
} catch (CacheException e) {
   cacheManager.getTransactionController().rollback()
}
```

Performance Considerations

Managing Contention

If two transactions attempt to perform a cache operation on the same element, the following rules apply:

- The first transaction gets access
- The following transactions block on the cache operation until either the first transaction completes or the transaction timeout occurs.

Note: When an element is involved in a transaction, it is replaced with a new element with a marker that is locked, along with the transaction ID.

What Granularity of Locking is Used?

Ehcache uses soft locks stored in the Element itself and is on a key basis.

Performance Comparisons

Any transactional cache adds an overhead, which is significant for writes and nearly negligible for reads. Compared to `transactionalMode="off"`, the time it takes to perform writes will be noticeably slower with either “xa” or “local” specified, and “xa_strict” will be the slowest.

Within the modes the relative time take to perform writes, where off = 1, is as follows:

- off - no overhead
- xa_strict - 20 times slower
- xa - 3 times slower
- local - 3 times slower The relative read performance is:
- off - no overhead
- xa_strict - 20 times slower
- xa - 30% slower
- local - 30% slower

Use “xa_strict” only when full guarantees are required, otherwise use one of the other modes.

Potential Timeouts in a Transactional Cache

Why Do Some Threads Regularly Time Out and Throw an Exception?

In transactional caches, write locks are in force whenever an element is updated, deleted, or added. With concurrent access, these locks cause some threads to block and appear to deadlock. Eventually the deadlocked threads time out (and throw an exception) to avoid being stuck in a deadlock condition.

Transactions in Write-Behind and Write-Through Caches

If your transaction-enabled cache is being used with a writer, write operations are queued until transaction commit time. A solely write-through approach would have its potential XAResource participate in the same transaction.

Write-behind is supported, however it should probably not be used with an XA transactional cache because the operations would never be part of the same transaction. Your writer would also be responsible for obtaining a new transaction.

Using Write-through with a non-XA resource would also work, but there is no guarantee the transaction will succeed after the write operations have been executed. On the other hand, any exception thrown during these write operations would

cause the transaction to be rolled back by having UserTransaction.commit() throw a RollbackException.

Support for Other Transaction Systems

Is IBM WebSphere Transaction Manager supported?

Mostly. The "xa_strict" mode is not supported due to each version of WebSphere being a custom implementation. That is, it has no stable interface to implement against. However, "xa", which uses TransactionManager callbacks, and "local" modes are supported.

When using Spring, make sure your configuration is set up correctly with respect to the PlatformTransactionManager and the WebSphere TM.

To confirm that Ehcache will succeed, try to manually register a com.ibm.websphere.jtaextensions.SynchronizationCallback in the com.ibm.websphere.jtaextensions.ExtendedJTATransaction. Get java:comp/websphere/ExtendedJTATransaction from JNDI, cast that to com.ibm.websphere.jtaextensions.ExtendedJTATransaction and call the registerSynchronizationCallbackForCurrentTran method. If you succeed, Ehcache should too.

Are Hibernate Transactions Supported?

Ehcache is a "transactional" cache for Hibernate purposes. The net.sf.ehcache.hibernate.EhCacheRegionFactory supports Hibernate entities configured with `<cache usage="transactional"/>`.

How Do I Make WebLogic 10 Work with a Transactional Cache?

WebLogic uses an optimization that is not supported by the Ehcache implementation. By default WebLogic 10 spawns threads to start the transaction on each XAResource in parallel. Because we need transaction work to be performed on the same Thread, you must turn off this optimization by setting the `parallel-xa-enabled` option to false in your domain configuration:

```
<jta>
   ...
   <checkpoint-interval-seconds>300</checkpoint-interval-seconds>
   <parallel-xa-enabled>false</parallel-xa-enabled>
   <unregister-resource-grace-period>30</unregister-resource-grace-period>
   ...
</jta></p>
```

How Do I Make Atomikos Work with a Cache in "xa" Mode?

Atomikos has a bug, which makes the "xa" mode's normal transaction termination mechanism unreliable. There is an alternative termination mechanism built in that transaction mode that is automatically enabled when `net.sf.ehcache.transaction.xa.alternativeTerminationMode` is set to true or when Atomikos is detected as the controlling transaction manager.

This alternative termination mode has strict requirement on the way threads are used by the transaction manager and Atomikos's default settings will not work unless you configure the following property as shown below:

```
com.atomikos.icatch.threaded_2pc=false
```

8 Write-Through and Write-Behind Caches

About Write-Through and Write-Behind Caches

Write-through caching is a caching pattern where writes to the cache cause writes to an underlying resource. The cache acts as a facade to the underlying resource. With this pattern, it often makes sense to read through the cache too.

Write-behind caching uses the same client API; however, the write happens asynchronously.

While file systems or a web-service clients can underlie the facade of a write-through cache, the most common underlying resource is a database. To simplify the discussion, we will use the database as the example resource.

Potential Benefits of Write-Behind

The major benefit of write-behind is database offload. This can be achieved in a number of ways:

- **Time shifting** - moving writes to a specific time or time interval. For example, writes could be batched up and written overnight, or at 5 minutes past the hour, to avoid periods of peak contention.
- **Rate limiting** - spreading writes out to flatten peaks. Say a point-of-sale (POS) network has an end-of-day procedure where data gets written to a central server. All POS nodes in the same time zone will write all at once. A very large peak will occur. Using rate limiting, writes could be limited to 100 TPS to whittle down the queue of writes over several hours
- **Conflation** - consolidate writes to create fewer transactions. For example, a value in a database row is updated by 5 writes, incrementing it from 10 to 20 to 31 to 40 to 45. Using conflation, the 5 transactions are replaced by one to update the value from 10 to 45.

These benefits must be weighed against the limitations and constraints imposed.

Limitations & Constraints of Write-Behind

Transaction Boundaries

If the cache participates in a JTA transaction, which means it is an XAResource, then the cache can be made consistent with the database. A write to the database, and a commit or rollback, happens with the transaction boundary. In write-behind, the write to the resource happens after the write to the cache. The transaction boundary is the write to the outstanding queue, not the write behind. In write-through mode, commit can get called and both the cache and the underlying resource can get committed at once. Because the database is being written to outside of the transaction, there is always a risk that a failure on the eventual write will occur. While this can be mitigated with retry counts and delays, compensating actions may be required.

Time Delay

The obvious implication of asynchronous writes is that there is a delay between when the cache is updated and when the database is updated. This introduces an inconsistency between the cache and the database, where the cache holds the correct value and the database will be eventually consistent with the cache.

The data passed into the CacheWriter methods is a snapshot of the cache entry at the time of the write to operation. A read against the database will result in incorrect data being loaded.

Applications Tolerant of Inconsistency

The application must be tolerant of inconsistent data. The following examples illustrate this requirement:

- The database is logging transactions and only appends are done.
- Reading is done by a part of the application that does not write, so there is no way that data can be corrupted. The application is tolerant of delays. For example, a news application where the reader displays the articles that are written.

Note if other applications are writing to the database, then a cache can often be inconsistent with the database.

Time Synchronization Across Nodes

Ideally node times should be synchronized. The write-behind queue is generally written to the underlying resource in timestamp order, based on the timestamp of the cache operation, although there is no guaranteed ordering. The ordering will be more consistent if all nodes are using the same time. This can easily be achieved by configuring your system clock to synchronize with a time authority using Network Time Protocol.

No Ordering Guarantees

The items on the write-behind queue are generally in order, but this isn't guaranteed. In certain situations, the items can be processed out of order. Additionally, when batching is used, write and delete collections are aggregated separately and can be processed inside the CacheWriter in a different order than the order that was used by the queue. Your application must be tolerant of item reordering or you need to compensate for this in your implementation of the CacheWriter. Possible examples are:

- Working with versioning in the cache elements.

  You may have to explicitly version elements. Auto-versioning is off by default and is effective only for unclustered MemoryStore caches. Distributed caches or caches that use off-heap or disk stores cannot use auto-versioning. To enable auto-versioning, set the system property `net.sf.ehcache.element.version.auto` (it is false by default). Note that if this property is turned on for one of the ineligible cache types, auto-versioning will silently fail.

- Verifications with the underlying resource to check if the scheduled write-behind operation is still relevant.

Introductory Video

Alex Snaps the primary author of Write Behind, presents an introductory video on Write Behind.

Using a Combined Read-Through and Write-Behind Cache

For applications that are not tolerant of inconsistency, the simplest solution is for the application to always read through the same cache that it writes through. Provided all database writes are through the cache, consistency is guaranteed.

The following aspects of read-through with write-behind should be considered:

Lazy Loading

The entire data set does not need to be loaded into the cache on startup. A read-through cache uses a CacheLoader that loads data into the cache on demand. In this way the cache can be populated lazily.

Caching of a Partial Dataset

If the entire dataset cannot fit in the cache, then some reads will miss the cache and fall through to the CacheLoader which will in turn hit the database. If a write has occurred but has not yet hit the database due to write-behind, then the database will be inconsistent. The simplest solution is to ensure that the entire dataset is in the cache. This then places some implications on cache configuration in the areas of expiry and eviction.

Eviction

Eviction or flushing of elements, occurs when the maximum elements for the cache have been exceeded. Be sure to size the cache appropriately to avoid eviction or flushing. See "Sizing Storage Tiers" in the *Configuration Guide* for Ehcache.

Expiry

Even if all of the dataset can fit in the cache, it could be evicted if elements expire. Consequently, you should set both the `timeToLive` and `timeToIdle` properties to eternal ("0") to prevent this from happening.

Write-Behind Sample Application

A sample web application for a raffle is available, which fully demonstrates how to use write behind. You can also check out the Ehcache Raffle application, that demonstrates Cache Writers and Cache Loaders from github.com. github

Configuring a Cache Writer

There are many configuration options for a cache writer. For a full list of configuration properties, see the Javadoc for the CacheWriterConfiguration class.

Below is an example of how to configure the cache writer in XML:

```
<cache name="writeThroughCache1" ... >
<cacheWriter writeMode="write-behind" maxWriteDelay="8" rateLimitPerSecond="5"
        writeCoalescing="true" writeBatching="true" writeBatchSize="20"
        retryAttempts="2" retryAttemptDelaySeconds="2">
   <cacheWriterFactory class="com.company.MyCacheWriterFactory"
                  properties="just.some.property=test; another.property=test2"
            propertySeparator=";"/>
</cacheWriter>
</cache>
```

Further examples:

```
<cache name="writeThroughCache2" ... >
 <cacheWriter/>
</cache>
<cache name="writeThroughCache3" ... >
 <cacheWriter writeMode="write-through" notifyListenersOnException="true"
      maxWriteDelay="30" rateLimitPerSecond="10" writeCoalescing="true"
      writeBatching="true" writeBatchSize="8" retryAttempts="20"
      retryAttemptDelaySeconds="60"/>
</cache>
<cache name="writeThroughCache4" ... >
 <cacheWriter writeMode="write-through" notifyListenersOnException="false"
      maxWriteDelay="0" rateLimitPerSecond="0" writeCoalescing="false"
      writeBatching="false" writeBatchSize="1" retryAttempts="0"
      retryAttemptDelaySeconds="0">
 <cacheWriterFactory class="net.sf.ehcache.writer.WriteThroughTestCacheWriterFactory"/>
 </cacheWriter>
</cache>
<cache name="writeBehindCache5" ... >
 <cacheWriter writeMode="write-behind" notifyListenersOnException="true"
      maxWriteDelay="8" rateLimitPerSecond="5" writeCoalescing="true"
      writeBatching="false" writeBatchSize="20"
      retryAttempts="2" retryAttemptDelaySeconds="2">
 <cacheWriterFactory class="net.sf.ehcache.writer.WriteThroughTestCacheWriterFactory"
                properties="just.some.property=test; another.property=test2"
         propertySeparator=";"/>
 </cacheWriter>
</cache>
```

As shown below, this configuration can also be achieved through the Cache constructor in Java:

```
Cache cache = new Cache(
new CacheConfiguration("cacheName", 10)
.cacheWriter(new CacheWriterConfiguration()
.writeMode(CacheWriterConfiguration.WriteMode.WRITE-BEHIND)
.maxWriteDelay(8)
.rateLimitPerSecond(5)
.writeCoalescing(true)
.writeBatching(true)
.writeBatchSize(20)
.retryAttempts(2)
```

```
.retryAttemptDelaySeconds(2)
.cacheWriterFactory(new CacheWriterConfiguration.CacheWriterFactoryConfiguration()
   .className("com.company.MyCacheWriterFactory")
   .properties("just.some.property=test; another.property=test2")
   .propertySeparator(";"))));
```

Instead of relying on a CacheWriterFactoryConfiguration to create a CacheWriter, it is also possible to explicitly register a CacheWriter instance from within Java code. This allows you to refer to local resources like database connections or file handles.

```
Cache cache = manager.getCache("cacheName");
MyCacheWriter writer = new MyCacheWriter(jdbcConnection);
cache.registerCacheWriter(writer);
```

CacheWriterFactory Attributes

The CacheWriterFactory supports the following attributes:

All modes

- `write-mode` [write-through | write-behind] - Whether to run in write-behind or write-through mode. The default is write-through.

write-through mode only

- `notifyListenersOnException` - Whether to notify listeners when an exception occurs on a store operation. Defaults to false. If using cache replication, set this attribute to "true" to ensure that changes to the underlying store are replicated.

write-behind mode only

- `writeBehindMaxQueueSize` - The maximum number of elements allowed per queue, or per bucket (if the queue has multiple buckets). "0" means unbounded (default). When an attempt to add an element is made, the queue size (or bucket size) is checked, and if full then the operation is blocked until the size drops by one. Note that elements or a batch currently being processed (and coalesced elements) are not included in the size value. Programmatically, this attribute can be set with:

```
net.sf.ehcache.config.CacheWriterConfiguration.setWriteBehindMaxQueueSize()
```

- `writeBehindConcurrency` - The number of thread-bucket pairs on the node for the given cache (default is 1). Each thread uses the settings configured for write-behind. For example, if `rateLimitPerSecond` is set to 100, each thread-bucket pair will perform up to 100 operations per second. In this case, setting `writeBehindConcurrency="4"` means that up to 400 operations per second will occur on the node for the given cache. Programmatically, this attribute can be set with:

```
net.sf.ehcache.config.CacheWriterConfiguration.setWriteBehindConcurrency()
```

- `maxWriteDelaySeconds` - The maximum number of seconds to wait before writing behind. Defaults to 0. If set to a value greater than 0, it permits operations to build up in the queue to enable effective coalescing and batching optimizations.

- `rateLimitPerSecond` - The maximum number of store operations to allow per second.
- `writeCoalescing` - Whether to use write coalescing. Defaults to false. When set to true, if multiple operations on the same key are present in the write-behind queue, then only the latest write is done (the others are redundant). This can dramatically reduce load on the underlying resource.
- `writeBatching` - Whether to batch write operations. Defaults to false. If set to true, storeAll and deleteAll will be called rather than store and delete being called for each key. Resources such as databases can perform more efficiently if updates are batched to reduce load.
- `writeBatchSize` - The number of operations to include in each batch. Defaults to 1. If there are less entries in the write-behind queue than the batch size, the queue length size is used. Note that batching is split across operations. For example, if the batch size is 10 and there were 5 puts and 5 deletes, the CacheWriter is invoked. It does not wait for 10 puts or 10 deletes.
- `retryAttempts` - The number of times to attempt writing from the queue. Defaults to 1.
- `retryAttemptDelaySeconds` - The number of seconds to wait before retrying.

API

CacheLoaders are exposed for API use through the `cache.getWithLoader(...)` method.

CacheWriters are exposed with cache.putWithWriter(...) and cache.removeWithWriter(...) methods. The code below show the method signature for the `cache.putWithWriter(...)` method. For the complete API, see the Cache Javadoc.

```
/**
* Put an element in the cache writing through a CacheWriter. If no CacheWriter
* has been set for the cache, then this method has the same effect as
* cache.put().
*
* Resets the access statistics on the element, which would be the case if
* it has previously been gotten from a cache, and is now being put back.
*
* Also notifies the CacheEventListener, if the writer operation succeeds, that:
*
* - the element was put, but only if the Element was actually put.
* - if the element exists in the cache, that an update has occurred, even if
* the element would be expired if it was requested
*
*
* @param element An object. If Serializable it can fully participate in
* replication and the DiskStore.
* @throws IllegalStateException    if the cache is not
*    {@link net.sf.ehcache.Status#STATUS_ALIVE}
* @throws IllegalArgumentException if the element is null
* @throws CacheException
*/
```

```
void putWithWriter(Element element) throws IllegalArgumentException,
IllegalStateException, CacheException;
```

SPI

The write-through SPI is the CacheWriter interface. Implementers perform writes to the underlying resource in their implementation.

```
/**
* A CacheWriter is an interface used for write-through and write-behind caching
* to an underlying resource.
* <p/>
* If configured for a cache, CacheWriter's methods will be called on a cache
* operation. A cache put will cause a CacheWriter write
* and a cache remove will cause a writer delete.
* <p>
* Implementers should create an implementation which handles storing and
* deleting to an underlying resource.
* </p>
* <h4>Write-Through</h4>
* In write-through mode, the cache operation will occur and the writer
* operation will occur before CacheEventListeners are notified. If the
* write operation fails an exception will be thrown. This can result in
* a cache which is inconsistent with the underlying resource.
* To avoid this, the cache and the underlying resource should be configured
* to participate in a transaction. In the event of a failure,
* a rollback can return all components to a consistent state.
* <p/>
* <h4>Write-Behind</h4>
* In write-behind mode, writes are written to a write-behind queue. They are
* written by a separate execution thread in a configurable
* way. When used with Terracotta Server Array, the queue is highly available.
* In addition, any node in the cluster may perform the write-behind operation.
* <p/>
* <h4>Creation and Configuration</h4>
* CacheWriters can be created using the CacheWriterFactory.
* <p/>
* The manner upon which a CacheWriter is actually called is determined by the
* {@link net.sf.ehcache.config.CacheWriterConfiguration} that is set up
* for a cache using the CacheWriter.
* <p/>
* See the CacheWriter chapter in the documentation for more information on
* how to use writers.
*
* @author Greg Luck
* @author Geert Bevin
* @version $Id: $
*/
public interface CacheWriter {
/**
* Creates a clone of this writer. This method will only be called by
* ehcache before a cache is initialized.
* <p/>
* Implementations should throw CloneNotSupportedException if they do not
* support clone but that will stop them from being used with defaultCache.
*
* @return a clone
* @throws CloneNotSupportedException if the extension could not be cloned.
*/
public CacheWriter clone(Ehcache cache) throws CloneNotSupportedException;
 /**
```

```
* Notifies writer to initialise themselves.
* <p/>
* This method is called during the Cache's initialise method after it has
* changed its status to alive. Cache operations are legal in this method.
*
* @throws net.sf.ehcache.CacheException
*/
void init();
/**
* Providers may be doing all sorts of exotic things and need to be able
* to clean up on dispose.
* <p/>
* Cache operations are illegal when this method is called. The cache itself
* is partly disposed when this method is called.
*/
void dispose() throws CacheException;
/**
* Write the specified value under the specified key to the underlying store.
* This method is intended to support both key/value creation and value
* update for a specific key.
*
* @param element the element to be written
*/
void write(Element element) throws CacheException;
/**
* Write the specified Elements to the underlying store. This method is
* intended to support both insert and update.
* If this operation fails (by throwing an exception) after a partial success,
* the convention is that entries which have been written successfully are
* to be removed from the specified mapEntries, indicating that the write
* operation for the entries left in the map has failed or has not been
* attempted.
*
* @param elements the Elements to be written
*/
void writeAll(Collection<Element> elements) throws CacheException;
 /**
* Delete the cache entry from the store
*
* @param entry the cache entry that is used for the delete operation
*/
void delete(CacheEntry entry) throws CacheException;
 /**
* Remove data and keys from the underlying store for the given collection
* of keys, if present. If this operation fails * (by throwing an exception)
* after a partial success, the convention is that keys which have been erased
* successfully are to be removed from the specified keys, indicating that the
* erase operation for the keys left in the collection has failed or has not
* been attempted.
*
* @param entries the entries that have been removed from the cache
*/
void deleteAll(Collection<CacheEntry> entries) throws CacheException;
  /**
 * This method will be called whenever an Element couldn't be handled by the
 * writer and all of the
 * {@link net.sf.ehcache.config.CacheWriterConfiguration#getRetryAttempts()
 * retryAttempts} have been tried.
 * <p>When batching is enabled, all of the elements in the failing batch will
 * be passed to this method.
 * <p>Try to not throw RuntimeExceptions from this method. Should an Exception
 * occur, it will be logged, but the element will still be lost.
 * @param element the Element that triggered the failure, or one of the elements
```

```
 * in the batch that failed.
 * @param operationType the operation we tried to execute
 * @param e the RuntimeException thrown by the Writer when the last retry attempt
 * was being executed
 */
void throwAway(Element element, SingleOperationType operationType,
   RuntimeException e);
}
```

Monitoring the Size of Write-Behind Queue

Use the method net.sf.ehcache.statistics.LiveCacheStatistics#getWriterQueueLength() to get the length of the queue. This method returns the number of elements on the local queue (in all local buckets) that are waiting to be processed, or -1 if no write-behind queue exists.

Note that elements or a batch currently being processed (and coalesced elements) are not included in the returned value.

Handling Exceptions that Occur After a Writer is Called

Once all retry attempts have been executed, on exception the element (or all elements of that batch) will be passed to the net.sf.ehcache.writer.CacheWriter#throwAway() method. The user can then act one last time on the element that failed to write.

A reference to the last thrown RuntimeException, and the type of operation that failed to execute for the element, are received. Any Exception thrown from that method will simply be logged and ignored. The element will be lost forever. It is important that implementers are careful about proper Exception handling in that last method.

A handy pattern is to use an eternal cache (potentially using a writer, so it is persistent) to store failed operations and their element. Users can monitor that cache and manually intervene on those errors at a later point.

9 Cache Loaders

About Cache Loaders

A CacheLoader is an interface that specifies load() and loadAll() methods with a variety of parameters. CacheLoaders are incorporated into the core Ehcache classes and can be configured in ehcache.xml. CacheLoaders are invoked in the following Cache methods:

- getWithLoader (synchronous)
- getAllWithLoader (synchronous)
- load (asynchronous)
- loadAll (asynchronous)

The methods will invoke a CacheLoader if there is no entry for the key or keys requested. By implementing CacheLoader, an application form of loading can take place. The get... methods follow the pull-through cache pattern. The load... methods are useful as cache warmers.

CacheLoaders are similar to the CacheEntryFactory used in SelfPopulatingCache, however SelfPopulatingCache is a decorator to Ehcache.

CacheLoaders can be set either declaratively in the ehcache.xml configuration file or programmatically. If a CacheLoader is set, it becomes the default CacheLoader. Some of the methods invoking loaders enable an override CacheLoader to be passed in as a parameter. More than one CacheLoader can be registered, in which case the loaders form a chain which are executed in order. If a loader returns null, the next in chain is called.

Declarative Configuration

The `cacheLoaderFactory` element specifies a CacheLoader, which can be used both asynchronously and synchronously to load objects into a cache.

```
<cache ...>
 <cacheLoaderFactory class="com.example.ExampleCacheLoaderFactory"
                             properties="type=int,startCounter=10"/>
</cache>
```

More than one `cacheLoaderFactory` element can be added, in which case the loaders form a chain which are executed in order. If a loader returns null, the next in chain is called.

Implementing a CacheLoaderFactory and CacheLoader

CacheLoaderFactory is an abstract factory for creating CacheLoaders. Implementers should provide their own concrete factory, extending this abstract factory. It can then

be configured in ehcache.xml. The factory class needs to be a concrete subclass of the abstract factory class CacheLoaderFactory, which is reproduced below:

```
/**
* An abstract factory for creating cache loaders. Implementers should provide
* their own concrete factory extending this factory.
*
* There is one factory method for JSR107 Cache Loaders and one for Ehcache ones.
* The Ehcache loader is a sub interface of the JSR107 Cache Loader.
*
* Note that both the JCache and Ehcache APIs also allow the CacheLoader to be set
* programmatically.
* @author Greg Luck
*/
public abstract class CacheLoaderFactory {
/**
* Creates a CacheLoader using the JSR107 creational mechanism.
* This method is called from {@link net.sf.ehcache.jcache.JCacheFactory}
*
* @param environment the same environment passed into
* {@link net.sf.ehcache.jcache.JCacheFactory}.
* This factory can extract any properties it needs from the environment.
* @return a constructed CacheLoader
*/
public abstract net.sf.jsr107cache.CacheLoader createCacheLoader(Map
environment);
/**
* Creates a CacheLoader using the Ehcache configuration mechanism at the time
* the associated cache is created.
*
* @param properties implementation specific properties. These are configured as
* comma separated name value pairs in ehcache.xml
* @return a constructed CacheLoader
*/
public abstract net.sf.ehcache.loader.CacheLoader createCacheLoader(Properties
properties);
/**
* @param cache the cache this extension should hold a reference to,
* and to whose lifecycle it should be bound.
* @param properties implementation specific properties configured as delimiter
*  separated name value pairs in ehcache.xml
* @return a constructed CacheLoader
*/
public abstract CacheLoader createCacheLoader(Ehcache cache, Properties
properties);
}
```

The factory creates a concrete implementation of the CacheLoader interface, which is reproduced below. A CacheLoader is bound to the lifecycle of a cache, so that the init() method is called during cache initialization, and dispose() is called on disposal of a cache.

```
/**
* Extends JCache CacheLoader with load methods that take an argument in addition
* to a key
* @author Greg Luck
*/
public interface CacheLoader extends net.sf.jsr107cache.CacheLoader {
/**
* Load using both a key and an argument.
*
* JCache will call through to the load(key) method, rather than this method,
* where the argument is null.
```

```
*
* @param key      the key to load the object for
* @param argument can be anything that makes sense to the loader
* @return the Object loaded
* @throws CacheException
*/
Object load(Object key, Object argument) throws CacheException;
/**
* Load using both a key and an argument.
*
* JCache will use the loadAll(key) method where the argument is null.
*
* @param keys     the keys to load objects for
* @param argument can be anything that makes sense to the loader
* @return a map of Objects keyed by the collection of keys passed in.
* @throws CacheException
*/
Map loadAll(Collection keys, Object argument) throws CacheException;
/**
* Gets the name of a CacheLoader
*
* @return the name of this CacheLoader
*/
String getName();
/**
* Creates a clone of this extension. This method will only be called by Ehcache
* before a cache is initialized.
*
* Implementations should throw CloneNotSupportedException if they do not support
* clone, but that will stop them from being used with defaultCache.
*
* @return a clone
* @throws CloneNotSupportedException if the extension could not be cloned.
*/
public CacheLoader clone(Ehcache cache) throws CloneNotSupportedException;
/**
* Notifies providers to initialise themselves.
*
* This method is called during the Cache's initialise method after it has changed
* it's status to alive. Cache operations are legal in this method.
*
* @throws net.sf.ehcache.CacheException
*/
void init();
/**
* Providers may be doing all sorts of exotic things and need to be able to clean
* up on dispose.
*
* Cache operations are illegal when this method is called. The cache itself is
* partly disposed when this method is called.
*
* @throws net.sf.ehcache.CacheException
*/
void dispose() throws net.sf.ehcache.CacheException;
/**
* @return the status of the extension
*/
public Status getStatus();
}
```

The implementations need to be placed in the classpath accessible to ehcache. For details on how the loading of these classes will be done, see "Class Loading" on page 99.

Programmatic Configuration

The following methods on Cache allow runtime interrogation, registration and unregistration of loaders:

```
/**
* Register a {@link CacheLoader} with the cache. It will then be tied into the
* cache lifecycle.
*
* If the CacheLoader is not initialised, initialise it.
*
* @param cacheLoader A Cache Loader to register
*/
public void registerCacheLoader(CacheLoader cacheLoader) {
registeredCacheLoaders.add(cacheLoader);
}
/**
* Unregister a {@link CacheLoader} with the cache. It will then be detached
* from the cache lifecycle.
*
* @param cacheLoader A Cache Loader to unregister
*/
public void unregisterCacheLoader(CacheLoader cacheLoader) {
registeredCacheLoaders.remove(cacheLoader);
}
/**
* @return the cache loaders as a live list
*/
public List<CacheLoader> getRegisteredCacheLoaders() {
return registeredCacheLoaders;
}
```

10 Cache Manager Event Listeners

About CacheManager Event Listeners

CacheManager event listeners allow implementers to register callback methods that will be executed when a CacheManager event occurs. CacheManager listeners implement the CacheManagerEventListener interface. The events include:

- Adding a Cache
- Removing a Cache

Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

Configuring a Cache Manager Event Listener

One CacheManagerEventListenerFactory and hence one CacheManagerEventListener can be specified per CacheManager instance. The factory is configured as below:

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

The entry specifies a CacheManagerEventListenerFactory which will be used to create a CacheManagerEventListener, which is notified when Caches are added or removed from the CacheManager. The attributes of a CacheManagerEventListenerFactory are:

- `class` — a fully qualified factory class name.
- `properties` — comma-separated properties having meaning only to the factory.

Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing. If no class is specified, or there is no cacheManagerEventListenerFactory element, no listener is created. There is no default.

Implementing a CacheManager Event Listener Factory and CacheManager Event Listener

CacheManagerEventListenerFactory is an abstract factory for creating CacheManager listeners. Implementers should provide their own concrete factory extending this abstract factory. It can then be configured in ehcache.xml.

The factory class needs to be a concrete subclass of the abstract factory CacheManagerEventListenerFactory, which is reproduced below:

```
/**
* An abstract factory for creating {@link CacheManagerEventListener}s.
* Implementers should provide their own concrete factory extending this
* factory. It can then be configured in ehcache.xml.
```

```
*
*/
public abstract class CacheManagerEventListenerFactory {
/**
* Create a CacheManagerEventListener
*
* @param properties implementation specific properties.
* These are configured as comma-separated name value pairs in ehcache.xml.
* Properties may be null.
* @return a constructed CacheManagerEventListener
*/
public abstract CacheManagerEventListener
       createCacheManagerEventListener(Properties properties);
}
```

The factory creates a concrete implementation of CacheManagerEventListener, which is reproduced below:

```
/**
* Allows implementers to register callback methods that will be executed when
* a CacheManager event occurs.
* The events include:
*
* adding a Cache
* removing a Cache
*
*
* Callbacks to these methods are synchronous and unsynchronized. It is the
* responsibility of the implementer to safely handle the potential
* performance and thread safety issues depending on what their listener
* is doing.
*/
public interface CacheManagerEventListener {
/**
* Called immediately after a cache has been added and activated.
*
* Note that the CacheManager calls this method from a synchronized method.
* Any attempt to call a synchronized method on CacheManager from this method
* will cause a deadlock.
*
* Note that activation will also cause a CacheEventListener status change
* notification from {@link net.sf.ehcache.Status#STATUS_UNINITIALISED} to
* {@link net.sf.ehcache.Status#STATUS_ALIVE}. Care should be taken on processing
* that notification because:
* <ul>
* <li>the cache will not yet be accessible from the CacheManager.
* <li>the addCaches methods whih cause this notification are synchronized on the
* CacheManager. An attempt to call
* {@link net.sf.ehcache.CacheManager#getCache(String)} will cause a deadlock.
* </ul>
* The calling method will block until this method returns.
*
* @param cacheName the name of the Cache the operation relates to
* @see CacheEventListener
*/
void notifyCacheAdded(String cacheName);
/**
* Called immediately after a cache has been disposed and removed. The calling
* method will block until this method returns.
*
* Note that the CacheManager calls this method from a synchronized method.
* Any attempt to call a synchronized method on CacheManager from this method
* will cause a deadlock.
*
```

```
* Note that a {@link CacheEventListener} status changed will also be triggered.
* Any attempt from that notification to access CacheManager will also result in
* a deadlock.
* @param cacheName the name of the Cache the operation relates to
*/
void notifyCacheRemoved(String cacheName);
}
```

The implementations need to be placed in the classpath accessible to Ehcache. Ehcache uses the ClassLoader returned by Thread.currentThread().getContextClassLoader() to load classes.

11 Cache Event Listeners

About Cache Event Listeners

Cache listeners allow implementers to register callback methods that will be executed when a cache event occurs. Cache listeners implement the CacheEventListener interface. The events include:

- An Element has been put
- An Element has been updated. Updated means that an Element exists in the Cache with the same key as the Element being put.
- An Element has been removed
- An Element expires, either because `timeToLive` or `timeToIdle` have been reached.

Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing. Listeners are guaranteed to be notified of events in the order in which they occurred. Elements can be put or removed from a Cache without notifying listeners by using the putQuiet() and removeQuiet() methods.

Configuring a Cache Event Listener

Cache event listeners are configured per cache. Each cache can have multiple listeners. Each listener is configured by adding a cacheEventListenerFactory element as follows:

```
<cache ...>
  <cacheEventListenerFactory class="" properties="" listenFor=""/>
  ...
</cache>
```

The entry specifies a CacheEventListenerFactory that creates a CacheEventListener, which then receives notifications. The attributes of a CacheEventListenerFactory are:

- `class` — a fully qualified factory class name.
- `properties` — optional comma-separated properties having meaning only to the factory.
- `listenFor` — describes which events will be delivered in a clustered environment (defaults to “all”).

  These are the possible values:

  - "all" — the default is to deliver all local and remote events.
  - "local" — deliver only events originating in the current node.
  - "remote" — deliver only events originating in other nodes (for BigMemory Max only).

Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

Implementing a Cache Event Listener Factory and Cache Event Listener

A CacheEventListenerFactory is an abstract factory for creating cache event listeners. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml. The following example demonstrates how to create an abstract CacheEventListenerFactory:

```
/**
* An abstract factory for creating listeners. Implementers should provide
* their own concrete factory extending this factory. It can then be configured
* in ehcache.xml
*
*/
public abstract class CacheEventListenerFactory {
/**
* Create a CacheEventListener
*
* @param properties implementation specific properties. These are configured
* as comma-separated name-value pairs in ehcache.xml
* @return a constructed CacheEventListener
*/
public abstract CacheEventListener createCacheEventListener(Properties properties);
}
```

The following example demonstrates how to create a concrete implementation of the CacheEventListener interface:

```
/**
* Allows implementers to register callback methods that will be executed when
* a cache event occurs.
* The events include:
* <ol>
* <li>put Element
* <li>update Element
* <li>remove Element
* <li>an Element expires, either because timeToLive or timeToIdle has been
* reached.
* </ol>
*
* Callbacks to these methods are synchronous and unsynchronized. It is the
* responsibility of the implementer to safely handle the potential performance
* and thread safety issues depending on what their listener is doing.
*
* Events are guaranteed to be notified in the order in which they occurred.
*
* Cache also has putQuiet and removeQuiet methods which do not notify listeners.
*
*/
public interface CacheEventListener extends Cloneable {
/**
* Called immediately after an element has been removed. The remove method will
* block until this method returns.
```

```
*
* Ehcache does not check for
*
*
* As the {@link net.sf.ehcache.Element} has been removed, only what was the
* key of the element is known.
*
*
* @param cache   the cache emitting the notification
* @param element just deleted
*/
void notifyElementRemoved(final Ehcache cache, final Element element) throws
  CacheException;
/**
* Called immediately after an element has been put into the cache. The
* {@link net.sf.ehcache.Cache#put(net.sf.ehcache.Element)} method
* will block until this method returns.
*
* Implementers may wish to have access to the Element's fields, including value,
* so the element is provided. Implementers should be careful not to modify the
* element. The effect of any modifications is undefined.
*
* @param cache   the cache emitting the notification
* @param element the element which was just put into the cache.
*/
void notifyElementPut(final Ehcache cache, final Element element) throws
  CacheException;
/**
* Called immediately after an element has been put into the cache and the element
* already existed in the cache. This is thus an update.
*
* The {@link net.sf.ehcache.Cache#put(net.sf.ehcache.Element)} method
* will block until this method returns.
*
* Implementers may wish to have access to the Element's fields, including value,
* so the element is provided. Implementers should be careful not to modify the
* element. The effect of any modifications is undefined.
*
* @param cache   the cache emitting the notification
* @param element the element which was just put into the cache.
*/
void notifyElementUpdated(final Ehcache cache, final Element element) throws
  CacheException;
/**
* Called immediately after an element is found to be expired. The
* {@link net.sf.ehcache.Cache#remove(Object)} method will block until this
* method returns.
* As the {@link Element} has been expired, only what was the key of the element
* is known.
*
* Elements are checked for expiry in Ehcache at the following times:
* <ul>
* <li>When a get request is made
* <li>When an element is spooled to diskStore in accordance with a MemoryStore
* eviction policy
* <li>In the DiskStore when the expiry thread runs, which by default is
* {@link net.sf.ehcache.Cache#DEFAULT_EXPIRY_THREAD_INTERVAL_SECONDS}
* </ul>
* If an element is found to be expired, it is deleted and this method is
* notified.
*
* @param cache   the cache emitting the notification
* @param element the element that has just expired
```

```
*
*    Deadlock Warning: expiry will often come from the DiskStore
*    expiry thread. It holds a lock to the DiskStore at the time the
*    notification is sent. If the implementation of this method calls into a
*    synchronized Cache method and that subsequently calls into
*    DiskStore a deadlock will result. Accordingly implementers of this method
*    should not call back into Cache.
*/
void notifyElementExpired(final Ehcache cache, final Element element);
/**
* Give the replicator a chance to cleanup and free resources when no longer
* needed
*/
void dispose();
/**
* Creates a clone of this listener. This method will only be called by Ehcache
* before a cache is initialized.
*
* This may not be possible for listeners after they have been initialized.
* Implementations should throw CloneNotSupportedException if they do not support
* clone.
* @return a clone
* @throws CloneNotSupportedException if the listener could not be cloned.
*/
public Object clone() throws CloneNotSupportedException;
}
```

Two other methods are also available:

- `void notifyElementEvicted(Ehcache cache, Element element)`

  Called immediately after an element is evicted from the cache. Eviction, which happens when a cache entry is deleted from a store, should not be confused with removal, which is a result of calling `Cache.removeElement(Element)`.

- `void notifyRemoveAll(Ehcache cache)`

  Called during `Ehcache.removeAll()` to indicate that all elements have been removed from the cache in a bulk operation. The usual `notifyElementRemoved(net.sf.ehcache.Ehcache, net.sf.ehcache.Element)` is not called. Only one notification is emitted because performance considerations do not allow for serially processing notifications where potentially millions of elements have been bulk deleted.

The implementations need to be placed in the classpath accessible to Ehcache. For details on how the loading of these classes will be done, see "Class Loading" on page 99.

Adding a Listener Programmatically

To add a listener programmatically, follow this example:

```
cache.getCacheEventNotificationService().registerListener(myListener);
```

12 Cache Exception Handlers

About Exception Handlers

By default, most cache operations will propagate a runtime CacheException on failure. An interceptor, using a dynamic proxy, may be configured so that a CacheExceptionHandler can be configured to intercept Exceptions. Errors are not intercepted.

Caches with ExceptionHandling configured are of type Ehcache. To get the exception handling behavior they must be referenced using CacheManager.getEhcache(), not CacheManager.getCache(), which returns the underlying undecorated cache.

Exception handlers are configured per cache. Each cache can have at most one exception handler. You can set CacheExceptionHandlers either declaratively in the ehcache.xml configuration file, or programmatically.

Declarative Configuration

To configure an exception handler declaratively, add the `cacheExceptionHandlerFactory` element to ehcache.xml as shown in the following example:

```
<cache ...>
 <cacheExceptionHandlerFactory
   class="net.sf.ehcache.exceptionhandler.CountingExceptionHandlerFactory"
   properties="logLevel=FINE"/>
</cache>
```

Implementing a Cache Exception Handler Factory and Cache Exception Handler

A CacheExceptionHandlerFactory is an abstract factory for creating cache exception handlers. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml.

Note: Your implementations need to be placed in the classpath accessible to Ehcache. For information about how class loading is handled, see "Class Loading" on page 99.

The factory class needs to be a concrete subclass of the abstract factory class CacheExceptionHandlerFactory, which is reproduced below.

```
/**
* An abstract factory for creating <code>CacheExceptionHandler</code>s at
* configuration time, in ehcache.xml.
* <p/>
* Extend to create a concrete factory
*
*/
```

```
public abstract class CacheExceptionHandlerFactory {
/**
* Create an <code>CacheExceptionHandler</code>
*
* @param properties implementation specific properties. These are configured
*                   as comma separated name value pairs in ehcache.xml
* @return a constructed CacheExceptionHandler
*/
public abstract CacheExceptionHandler createExceptionHandler(Properties properties);
}
```

The factory creates a concrete implementation of the `CacheExceptionHandler` interface, which is reproduced below:

```
/**
* A handler which may be registered with an Ehcache, to handle exception on
* Cache operations.
*
* Handlers may be registered at configuration time in ehcache.xml, using a
* CacheExceptionHandlerFactory, or set at runtime (a strategy).
*
* If an exception handler is registered, the default behaviour of throwing the
* exception will not occur. The handler method on Exception will be called.
* Of course, if the handler decides to throw the exception, it will propagate
* up through the call stack. If the handler does not, it won't.
*
* Some common Exceptions thrown, and which therefore should be considered when
* implementing this class, are listed below:
* <ul>
* <li>{@link IllegalStateException} if the cache is not
* {@link net.sf.ehcache.Status#STATUS_ALIVE}
* <li>{@link IllegalArgumentException} if an attempt is made to put a null
* element into a cache
* <li>{@link net.sf.ehcache.distribution.RemoteCacheException} if an issue
*  occurs in remote synchronous replication
* <li>
* <li>
* </ul>
*
*/
public interface CacheExceptionHandler {
/**
* Called if an Exception occurs in a Cache method. This method is not
* called if an Error occurs.
*
* @param Ehcache   the cache in which the Exception occurred
* @param key       the key used in the operation, or null if the operation
* does not use a key or the key was null
* @param exception the exception caught
*/
void onException(Ehcache ehcache, Object key, Exception exception);
}
```

Programmatic Configuration

The following example shows how to add exception handling to a cache, and then add the cache back into cache manager so that all clients obtain the cache handling decoration.

```
CacheManager cacheManager = ...
```

```
Ehcache cache = cacheManger.getCache("exampleCache");
ExceptionHandler handler = new ExampleExceptionHandler(...);
cache.setCacheLoader(handler);
Ehcache proxiedCache = ExceptionHandlingDynamicCacheProxy.createProxy(cache);
cacheManager.replaceCacheWithDecoratedCache(cache, proxiedCache);
```

13 Cache Decorators

About Cache Decorators

Ehcache uses the Ehcache interface, of which Cache is an implementation. It is possible and encouraged to create Ehcache decorators that are backed by a Cache instance, implement Ehcache and provide extra functionality.

The Decorator pattern is one of the well known Gang of Four patterns.

Decorated caches are accessed from the CacheManager using `CacheManager.getEhcache(String name)`. Note that, for backward compatibility, `CacheManager.getCache(String name)` has been retained. However only `CacheManager.getEhcache(String name)` returns the decorated cache.

Built-in Decorators

BlockingCache

This is a Blocking decorator for Ehcache that allows concurrent read access to elements already in the cache. If the element is null, other reads will block until an element with the same key is put into the cache. This decorator is useful for constructing read-through or self-populating caches. BlockingCache is used by CachingFilter.

SelfPopulatingCache

A self-populating decorator for Ehcache that creates entries on demand. Clients of the cache simply call it without needing knowledge of whether the entry exists in the cache. If null, the entry is created. The cache is designed to be refreshed. Refreshes operate on the backing cache, and do not degrade performance of get calls.

SelfPopulatingCache extends BlockingCache. Multiple threads attempting to access a null element will block until the first thread completes. If refresh is being called the threads do not block - they return the stale data. This is very useful for engineering highly scalable systems.

Caches with Exception Handling

Caches with exception handlers are decorated. For information about adding an exception handler to a cache, see "Cache Exception Handlers" on page 81.

Creating a Decorator

You can add decorators to a cache either declaratively in the ehcache.xml configuration file, or programmatically.

Declarative Creation

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

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

The properties will be parsed according to the delimiter (default is comma ",") and passed to the concrete factory's createDecoratedEhcache(Ehcache cache, Properties properties) method along with the reference to the owning cache.

It is configured as per the following example:

```
<cacheDecoratorFactory
     class="com.company.SomethingCacheDecoratorFactory"
     properties="property1=36 ..." />
```

Note that decorators can be configured against the defaultCache. This is very useful for frameworks like Hibernate that add caches based on the configuration of the defaultCache.

Programmatic Creation

Cache decorators are created as follows:

```
BlockingCache newBlockingCache = new BlockingCache(cache);
```

The class must implement Ehcache.

Adding Decorated Caches to a CacheManager

Having created a decorator programmatically, it is generally useful to put it in a place where multiple threads can access it. Note that decorators created via configuration in ehcache.xml have already been added to the CacheManager.

Using CacheManager.replaceCacheWithDecoratedCache()

A built-in way is to replace the Cache in CacheManager with the decorated one. This is achieved as in the following example:

```
cacheManager.replaceCacheWithDecoratedCache(cache, newBlockingCache);
```

The CacheManager.replaceCacheWithDecoratedCache( ) method requires that the decorated cache be built from the underlying cache from the same name.

Note that any overridden Ehcache methods will take on new behaviors without casting, as per the normal rules of Java. Casting is only required for new methods that the decorator introduces.

Any calls to get the cache out of the CacheManager now return the decorated one.

A word of caution. This method should be called in an appropriately synchronized init style method before multiple threads attempt to use it. All threads must be referencing the same decorated cache. An example of a suitable init method is found in CachingFilter:

```
/**
* The cache holding the web pages. Ensure that all threads for a given cache
* name are using the same instance of this.
*/
private BlockingCache blockingCache;
/**
* Initialises blockingCache to use
*
* @throws CacheException The most likely cause is that a cache has not been
*                        configured in Ehcache's configuration file ehcache.xml
*                        for the filter name
*/
public void doInit() throws CacheException {
  synchronized (this.getClass()) {
    if (blockingCache == null) {
      final String cacheName = getCacheName();
      Ehcache cache = getCacheManager().getEhcache(cacheName);
      if (!(cache instanceof BlockingCache)) {
        //decorate and substitute
        BlockingCache newBlockingCache = new BlockingCache(cache);
        getCacheManager().replaceCacheWithDecoratedCache(cache, newBlockingCache);
      }
      blockingCache = (BlockingCache) getCacheManager().getEhcache(getCacheName());
    }
  }
}
Ehcache blockingCache = singletonManager.getEhcache("sampleCache1");
```

The returned cache will exhibit the decorations.

Using CacheManager.addDecoratedCache()

Sometimes you want to add a decorated cache but retain access to the underlying cache.

The way to do this is to create a decorated cache and then call cache.setName(new_name) and then add it to CacheManager with CacheManager.addDecoratedCache().

```
/**
* Adds a decorated {@link Ehcache} to the CacheManager. This method neither
* creates the memory/disk store nor initializes the cache. It only adds the
* cache reference to the map of caches held by this cacheManager.
*
* It is generally required that a decorated cache, once constructed, is made
* available to other execution threads. The simplest way of doing this is to
* either add it to the cacheManager with a different name or substitute the
* original cache with the decorated one.
*
* This method adds the decorated cache assuming it has a different name.
* If another cache (decorated or not) with the same name already exists,
* it will throw {@link ObjectExistsException}. For replacing existing
* cache with another decorated cache having same name, please use
* {@link #replaceCacheWithDecoratedCache(Ehcache, Ehcache)}
*
* Note that any overridden Ehcache methods by the decorator will take on
* new behaviours without casting. Casting is only required for new methods
```

```
* that the decorator introduces. For more information see the well known
* Gang of Four Decorator pattern.
*
* @param decoratedCache
* @throws ObjectExistsException
*             if another cache with the same name already exists.
*/
public void addDecoratedCache(Ehcache decoratedCache) throws ObjectExistsException {
```

14 Cache Extensions

About Cache Extensions

Cache extensions are a general-purpose mechanism to allow generic extensions to a cache. Cache extensions are tied into the cache lifecycle. For that reason, this interface has the lifecycle methods.

Cache extensions are created using the CacheExtensionFactory, which has a createCacheCacheExtension() method that takes as a parameter a Cache and properties. It can thus call back into any public method on Cache, including, of course, the load methods. Cache extensions are suitable for timing services, where you want to create a timer to perform cache operations. (Another way of adding Cache behavior is to decorate a cache. For an example of, "Blocking and Self Populating Caches" on page 41.)

Because a CacheExtension holds a reference to a Cache, the CacheExtension can do things such as registering a CacheEventListener or even a CacheManagerEventListener, all from within a CacheExtension, creating more opportunities for customization.

Declarative Configuration

Cache extensions are configured per cache. Each cache can have zero or more.

You configure a cache extension by adding a `cacheExceptionHandlerFactory` element as shown in the following example:

```
<cache ...>
    <cacheExtensionFactory
    class="com.example.FileWatchingCacheRefresherExtensionFactory"
    properties="refreshIntervalMillis=18000, loaderTimeout=3000,
    flushPeriod=whatever, someOtherProperty=someValue ..."/>
</cache>
```

Implementing a Cache Extension Factory and Cache Extension

A CacheExtensionFactory is an abstract factory for creating cache extension. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml. The factory class needs to be a concrete subclass of the abstract factory class CacheExtensionFactory, which is reproduced below:

```
/**
* An abstract factory for creating CacheExtensions. Implementers should
* provide their own concrete factory extending this factory.
* It can then be configured in ehcache.xml.
*
*/
public abstract class CacheExtensionFactory {
/**
* @param cache the cache this extension should hold a reference to, and to
* whose lifecycle it should be bound.
```

```
* @param properties implementation specific properties configured as
* delimiter separated name value pairs in ehcache.xml
*/
public abstract CacheExtension createCacheExtension(Ehcache cache,
Properties properties);
}
```

The factory creates a concrete implementation of the `CacheExtension` interface, which is reproduced below:

```
/**
* This is a general purpose mechanism to allow generic extensions to a Cache.
*
* CacheExtensions are tied into the Cache lifecycle. For that reason this
* interface has the lifecycle methods.
*
* CacheExtensions are created using the CacheExtensionFactory which has a
* createCacheCacheExtension() method which takes as a parameter a Cache and
* properties. It can thus call back into any public method on Cache, including,
* of course, the load methods.
*
* CacheExtensions are suitable for timing services, where you want to create a
* timer to perform cache operations. The other way of adding Cache behaviour is
* to decorate a cache.
* See {@link net.sf.ehcache.constructs.blocking.BlockingCache}
* for an example of how to do this.
*
* Because a CacheExtension holds a reference to a Cache, the CacheExtension
* can do things such as registering a CacheEventListener or even a
* CacheManagerEventListener, all from within a CacheExtension,
* creating more opportunities for customisation.
*
*/
public interface CacheExtension {
/**
* Notifies providers to initialise themselves.
*
* This method is called during the Cache's initialise method after it
* has changed its status to alive. Cache operations are legal in
* this method.
*
* @throws CacheException
*/
void init();
/**
* Providers may be doing all sorts of exotic things and need to be able to
* clean up on dispose.
*
* Cache operations are illegal when this method is called. The cache
* itself is partly disposed when this method is called.
*
* @throws CacheException
*/
void dispose() throws CacheException;
/**
* Creates a clone of this extension. This method will only be called by
* Ehcache before a cache is initialized.
*
* Implementations should throw CloneNotSupportedException if they do not
* support clone but that will stop them from being used with defaultCache.
*
* @return a clone
* @throws CloneNotSupportedException if the extension could not be cloned.
```

```
*/
public CacheExtension clone(Ehcache cache) throws CloneNotSupportedException;
/**
* @return the status of the extension
*/
public Status getStatus();
}
```

The implementations need to be placed in the classpath accessible to ehcache. For details on how the loading of these classes will be done, see "Class Loading" on page 99.

Programmatic Configuration

Cache extensions can also be programmatically added to a Cache as shown below:

```
TestCacheExtension testCacheExtension = new TestCacheExtension(cache, ...);
testCacheExtension.init();
cache.registerCacheExtension(testCacheExtension);
```

15 Cache Eviction Algorithms

About Cache Eviction Algorithms

A cache eviction algorithm is a way of deciding which element to evict when the cache is full. In Ehcache , the memory store and the off-heap store might be limited in size. When these stores get full, elements are evicted. The eviction algorithms determine which elements are evicted. The default algorithm is Least Recently Used (LRU).

What happens on eviction depends on the cache configuration. If a disk store is configured, the evicted element is flushed to disk; otherwise it is removed. The disk store size by default is unbounded. But a maximum size can be set as described in "Sizing the Storage Tiers" in the *Configuration Guide* forEhcache . If the disk store is full, then adding an element causes an existing element to be evicted.

Note: The disk store eviction algorithm is not configurable. It uses LFU.

Built-in Memory Store Eviction Algorithms

The idea here is, given a limit on the number of items to cache, how to choose the thing to evict that gives the best result.

In 1966 Laszlo Belady showed that the most efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This is a theoretical result that is unimplementable without domain knowledge. The Least Recently Used (LRU) algorithm is often used as a proxy. In general, it works well because of the locality of reference phenomenon and is the default in most caches.

A variation of LRU is the default eviction algorithm in Ehcache .

Ehcache provides three eviction algorithms to choose from for the memory store.

Least Recently Used (LRU)

This is the default and is a variation on the Least Frequently Used algorithm.

The oldest element is the Less Recently Used element. The last-used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.

This algorithm takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

If probabilistic eviction does not suit your application, a true Least Recently Used deterministic algorithm is available by setting `java -Dnet.sf.ehcache.use.classic.lru=true`.

Least Frequently Used (LFU)

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

If cache-element usage follows a Pareto distribution, this algorithm might give better results than LRU.

LFU is an algorithm unique to the Ehcache API. It takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

First In First Out (FIFO)

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

This algorithm is used if the use of an element makes it less likely to be used in the future. An example here would be an authentication cache.

It takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

Plugging in Your own Eviction Algorithm

Ehcache allows you to plug in your own eviction algorithm using Cache.setMemoryStoreEvictionPolicy(Policy policy). You can utilize any Element metadata, which makes possible some very interesting approaches. For example, you might evict an element if it has been hit more than ten times.

```
/**
* Sets the eviction policy strategy. The Cache will use a policy at startup.
* There are three policies which can be configured: LRU, LFU and FIFO. However
* many other policies are possible. That the policy has access to the whole
* element enables policies based on the key, value, metadata, statistics, or a
* combination of any of the above.
*
* It is safe to change the policy of a store at any time. The new policy takes
* effect immediately.
*
* @param policy the new policy
*/
public void setMemoryStoreEvictionPolicy(Policy policy) {
  memoryStore.setEvictionPolicy(policy);
}
```

A Policy must implement the following interface:

```
public interface Policy {
  /**
   * @return the name of the Policy. Inbuilt examples are LRU, LFU and FIFO.
   */
```

```
  String getName();
  /**
   * Finds the best eviction candidate based on the sampled elements. What
   * distinguishes this approach from the classic data structures approach is
   * that an Element contains metadata (e.g. usage statistics) which can be
   * used for making policy decisions, while generic data structures do not.
   * It is expected that implementations will take advantage of that
   * metadata.
   *
   * @param sampledElements this should be a random subset of the population
   * @param justAdded we probably never want to select the element just added.
   * It is provided so that it can be ignored if selected. May be null.
   * @return the selected Element
   */
  Element selectedBasedOnPolicy(Element[] sampledElements, Element justAdded);
  /**
   * Compares the desirableness for eviction of two elements
   *
   * @param element1 the element to compare against
   * @param element2 the element to compare
   * @return true if the second element is preferable for eviction to the
   * first element under ths policy
   */
  boolean compare(Element element1, Element element2);
}
```

Disk Store Eviction Algorithm

The disk store uses the Least Frequently Used algorithm to evict an element when the store it is full.

16 Class Loading

About Class Loading

Class loading, within the plethora of environments that Ehcache can be running, could be complex. But with Ehcache, all class loading is done in a standard way in one utility class: ClassLoaderUtil.

Plugin Class Loading

Ehcache allows plugins for events and distribution. These are loaded and created as follows:

```
/**
 * Creates a new class instance. Logs errors along the way. Classes are loaded
 * using the Ehcache standard classloader.
 *
 * @param className a fully qualified class name
 * @return null if the instance cannot be loaded
 */
public static Object createNewInstance(String className) throws CacheException {
    Class clazz;
    Object newInstance;
    try {
    clazz = Class.forName(className, true, getStandardClassLoader());
    } catch (ClassNotFoundException e) {
    //try fallback
    try {
        clazz = Class.forName(className, true, getFallbackClassLoader());
    } catch (ClassNotFoundException ex) {
        throw new CacheException("Unable to load class " + className +
                    ". Initial cause was " + e.getMessage(), e);
    }
    }
    try {
    newInstance = clazz.newInstance();
    } catch (IllegalAccessException e) {
    throw new CacheException("Unable to load class " + className +
                ". Initial cause was " + e.getMessage(), e);
    } catch (InstantiationException e) {
    throw new CacheException("Unable to load class " + className +
                ". Initial cause was " + e.getMessage(), e);
    }
    return newInstance;
}
/**
 * Gets the ClassLoader that all classes in ehcache, and extensions,
 * should use for classloading. All ClassLoading in Ehcache should use this
 * one. This is the only thing that seems to work for all of the class
 * loading situations found in the wild.
 * @return the thread context class loader.
 */
public static ClassLoader getStandardClassLoader() {
    return Thread.currentThread().getContextClassLoader();
}
/**
 * Gets a fallback ClassLoader that all classes in ehcache, and
 * extensions, should use for classloading. This is used if the
```

```
 * context class loader does not work.
 * @return the ClassLoaderUtil.class.getClassLoader(); */
public static ClassLoader getFallbackClassLoader() {
    return ClassLoaderUtil.class.getClassLoader();
}
```

If this does not work for some reason, a CacheException is thrown with a detailed error message.

Loading of ehcache.xml Resources

If the configuration is otherwise unspecified, Ehcache looks for a configuration in the following order:

- Thread.currentThread().getContextClassLoader().getResource("/ehcache.xml")
- ConfigurationFactory.class.getResource("/ehcache.xml")
- ConfigurationFactory.class.getResource("/ehcache-failsafe.xml")

Ehcache uses the first configuration found. Note the use of “/ehcache.xml”, which requires that ehcache.xml be placed at the root of the classpath (i.e., not in any package).