
TERRACOTTA

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

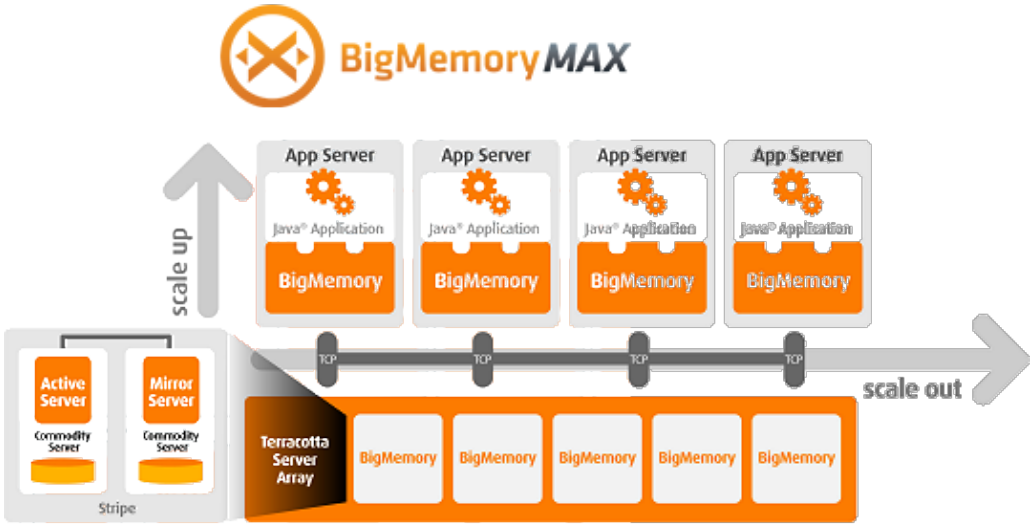
Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Terracotta 4.0 Documentation

Welcome to BigMemory Max

BigMemory Max lets you manage ALL your data in memory — across distributed servers.



With BigMemory Max, your in-memory capacity is limited only by how much RAM you have in your data center.

With BigMemory Max, you get:

- Unlimited in-memory data management across distributed servers
- Data consistency guarantees
- Advanced monitoring, search, and management for in-memory data

To get started, download BigMemory Max. Then check out the Get Started guide for instructions on installing your license key, configuring BigMemory Max, and connecting your apps to your new BigMemory Max data stores.

If you are at the end of a free trial or would like to purchase a commercial license for BigMemory Max, contact us.

Hello, World!

The first step to using BigMemory is to set up one or more instances of Ehcache. BigMemory uses Ehcache as its main programming interface.

- Download and unpack the BigMemory Max download kit.
- Add the license key (terracotta-license.key) to your classpath.
- Add the following jars in the BigMemory Max download kit to your classpath:
    - `common/lib/bigmemory-<version>.jar`
    - `apis/ehcache/lib/ehcache-ee-<version>.jar`
    - `apis/ehcache/lib/slf4j-api-<version>.jar`
    - `apis/ehcache/lib/slf4j-jdk-<version>.jar` - only some versions include this jar

Create Configuration File

Create a basic configuration file, name it "ehcache.xml" and put it in your classpath:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
         name="HelloWorldConfig">
  <cache name="hello-world" maxBytesLocalHeap="64M"/>
</ehcache>
```

This tells BigMemory that you have a data store called "hello-world" and that it can use a maximum of 64 megabytes of heap in the local Java Virtual Machine.

Create HelloWorld.java

Create and compile a Java class called HelloWorld:

```
import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Element;

public class HelloWorld {

    public static void main(final String[] args) {

        // Create a cache manager
        final CacheManager cacheManager = new CacheManager();

        // create the data store called "hello-world"
        final Cache dataStore = cacheManager.getCache("hello-world");

        // create a key to map the data to
        final String key = "greeting";

        // Create a data element
        final Element putGreeting = new Element(key, "Hello, World!");

        // Put the element into the data store
        dataStore.put(putGreeting);
```

```
        // Retrieve the data element
        final Element getGreeting = dataStore.get(key);

        // Print the value
        System.out.println(getGreeting.getObjectValue());
    }
}
```

Execute

When you run the program in a terminal, you will see BigMemory print out its license and startup info, then the string "Hello, World!".

Next Step

Next Step: Basic Create, Read, Update and Delete (CRUD) ›

Basic CRUD

Now that you've created your first instance of BigMemory with the Ehcache interface, let's exercise its basic CRUD functions.

Create Configuration File

Create a new configuration file called "ehcache-crud.xml" in your classpath and add a new cache called "crud."

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
     name="CRUDConfig" maxBytesLocalHeap="64M">
  <cache name="hello-world"/>
  <cache name="crud"/>
</ehcache>
```

(Notice that the maxByteLocalHeap setting has been moved to the top-level <ehcache> element—BigMemory gives you the ability to sculpt the memomory profile of your individual data sets as well as apply bulk settings to all data sets.)

Create Crud.java

Create and compile a Java class called Crud:

```
import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Element;

public class Crud {
    public static void main(final String[] args) {
        // Create a cache manager using the factory method
        // AND specify the new configuration file
        final CacheManager cacheManager =
              CacheManager.newInstance(
                Crud.class.getResource("/ehcache-crud.xml"));

        // Get the "crud" cache from the cache manager...
        final Cache dataStore = cacheManager.getCache("crud");

        // Set up the first data element...
        final String myKey = "My Key";
        final String myData = "My Data";
        final Element createElement = new Element(myKey, myData);

        // CREATE data using the put(Element) method...
        dataStore.put(createElement);
        System.out.println("Created data: " + createElement);

        // READ data using the get(Object) method...
        Element readElement = dataStore.get(myKey);
        System.out.println("Read data:    " + readElement);

        // Check to make sure the data is the same...
        if (! myData.equals(readElement.getObjectValue())) {
            throw new RuntimeException("My data doesn't match!");
```

```
        }

        // UPDATE data by mapping a new value to the same key...
        final String myNewData = "My New Data";
        final Element updateElement = new Element(myKey, myNewData);
        dataStore.put(updateElement);
        System.out.println("Updated data: " + updateElement);

        // Test to see that the data is updated...
        readElement = dataStore.get(myKey);
        if (! myNewData.equals(readElement.getObjectValue())) {
            throw new RuntimeException("My data doesn't match!");
        }

        // DELETE data using the remove(Object) method...
        final boolean wasRemoved = dataStore.remove(myKey);
        System.out.println("Removed data: " + wasRemoved);
        if (! wasRemoved) {
            throw new RuntimeException("My data wasn't removed!");
        }

        // Be polite and release the CacheManager resources...
        cacheManager.shutdown();
    }
}
```

Execute

When you run the Crud program in a terminal, you should see output like this:

```
Created data: [ key = My Key, value=My Data, version=1, hitCount=0,

          CreationTime = 1361401376761, LastAccessTime = 1361401376761 ]

Read data:    [ key = My Key, value=My Data, version=1, hitCount=1,

          CreationTime = 1361401376761, LastAccessTime = 1361401376775 ]

Updated data: [ key = My Key, value=My New Data, version=1, hitCount=0,

          CreationTime = 1361401376776, LastAccessTime = 1361401376776 ]

Removed data: true
```

Next Step

Next Step: Search ›

Search

Now that you've experimented with BigMemory's basic CRUD operations, let's take a look at some of the search capabilities.

We will create a data set of Person objects that has some searchable attributes, including height, weight and body mass index.

Create Configuration File

Create a new configuration file called "ehcache-search.xml" in your classpath and add a new cache called "search."

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
     name="SearchConfig" maxBytesLocalHeap="64M">
  <cache name="hello-world"/>
  <cache name="crud"/>
  <cache name="search">
    <searchable>
        <searchAttribute name="height"/>
        <searchAttribute name="weight"/>
        <searchAttribute name="bodyMassIndex" />
    </searchable>
  </cache>
</ehcache>
```

Create Search.java

Create and compile a Java class called Search:

```
import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Element;
import net.sf.ehcache.search.Attribute;
import net.sf.ehcache.search.Query;
import net.sf.ehcache.search.Result;
import net.sf.ehcache.search.Results;

public class Search {

  public static void main(final String[] args) {

    // Create a cache manager using the factory method...
    final CacheManager cacheManager = CacheManager.newInstance(Crud.class
        .getResource("/ehcache-search.xml"));

    // Retrieve the "search" data set...
    final Cache dataSet = cacheManager.getCache("search");

    // Create some searchable objects...
    final Person janeAusten = new Person(1, "Jane", "Austen", 5 * 12 + 7, 130);
    final Person charlesDickens = new Person(2, "Charles", "Dickens",
        5 * 12 + 9, 160);
    final Person janeDoe = new Person(3, "Jane", "Doe", 5 * 12 + 5, 145);
```

```
    // Put the searchable objects into the data set...
    dataSet.put(new Element(janeAusten.getId(), janeAusten));
    dataSet.put(new Element(charlesDickens.getId(), charlesDickens));
    dataSet.put(new Element(janeDoe.getId(), janeDoe));

    // Fetch the Body Mass Index (BMI) attribute...
    Attribute<Object> bmi = dataSet.getSearchAttribute("bodyMassIndex");

    // Create a query for all people with a BMI greater than 23...
    final Query query = dataSet.createQuery().addCriteria(bmi.gt(23F))
        .includeValues();

    // Execute the query...
    final Results results = query.execute();

    // Print the results...
    for (Result result : results.all()) {
      System.out.println(result.getValue());
    }
  }

  public static class Person {
    private final String firstName;
    private final String lastName;
    private final long id;
    private final int height;
    private final int weight;

    public Person(long id, final String firstName, final String lastName,
        final int height, final int weight) {
      this.id = id;
      this.firstName = firstName;
      this.lastName = lastName;
      this.height = height;
      this.weight = weight;
    }

    public String getFirstName() {
      return firstName;
    }

    public String getLastName() {
      return lastName;
    }

    public int getHeight() {
      return height;
    }

    public int getWeight() {
      return weight;
    }

    public float getBodyMassIndex() {
      return ((float) weight / ((float) height * (float) height)) * 703;
    }

    public long getId() {
      return id;
    }

    public String toString() {
```

```
      return "[id=" + id + ", firstName=" + firstName + ", lastName="
          + lastName + ", height=" + height + " in" + ", weight=" + weight
          + " lbs" + ", bmi=" + getBodyMassIndex() + "]";
    }
  }

}
```

Execute

When you run the Search program in a terminal, you should see output like this:

```
[id=2, firstName=Charles, lastName=Dickens, height=69 in, weight=160 lbs, bmi=23.625288]

[id=3, firstName=Jane, lastName=Doe, height=65 in, weight=145 lbs, bmi=24.126627]
```

Next Step

Next Step: Sorting Results ›

Sorting Results

Now that you've experimented with searching through data sets, let's try sorting those results.

Create Configuration File

Create a new configuration file called "ehcache-sort.xml" in your classpath and add a new cache called "sort."

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
     name="SortConfig" maxBytesLocalHeap="64M">
  <cache name="hello-world"/>
  <cache name="crud"/>
  <cache name="search">
    <searchable>
        <searchAttribute name="height"/>
        <searchAttribute name="weight"/>
        <searchAttribute name="bodyMassIndex" />
    </searchable>
  </cache>
  <cache name="sort">
    <searchable>
        <searchAttribute name="height"/>
        <searchAttribute name="weight"/>
    </searchable>
  </cache>
</ehcache>
```

Create Sort.java

Create and compile a Java class called Sort:

```
import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Element;
import net.sf.ehcache.search.Attribute;
import net.sf.ehcache.search.Direction;
import net.sf.ehcache.search.Query;
import net.sf.ehcache.search.Result;
import net.sf.ehcache.search.Results;

public class Sort {

  public static void main(final String[] args) {

    // Create a cache manager using the factory method...
    final CacheManager cacheManager = CacheManager.newInstance(Sort.class
        .getResource("/ehcache-sort.xml"));

    // Retrieve the "sort" data set...
    final Cache dataSet = cacheManager.getCache("sort");

    // Create some objects...
    final Person janeAusten = new Person(1, "Jane", "Austen", 5 * 12 + 7, 130);
    final Person charlesDickens = new Person(2, "Charles", "Dickens",
        5 * 12 + 9, 160);
    final Person janeDoe = new Person(3, "Jane", "Doe", 5 * 12 + 5, 145);
```

```
    final Person alexSmith = new Person(4, "Alex", "Smith", 5 * 12 + 5, 160);

    // Put the objects into the data set...
    dataSet.put(new Element(janeAusten.getId(), janeAusten));
    dataSet.put(new Element(charlesDickens.getId(), charlesDickens));
    dataSet.put(new Element(janeDoe.getId(), janeDoe));
    dataSet.put(new Element(alexSmith.getId(), alexSmith));

    // Fetch the height and weight attributes that we'll use in the query...
    Attribute<Integer> height = dataSet.getSearchAttribute("height");
    Attribute<Integer> weight = dataSet.getSearchAttribute("weight");

    // Create a query for all people with a height greater than 5' 0", ordered
    // first by height, then by weight... this will retrieve the entire data
    // set, but we'll get to see how the results are sorted.
    final Query query = dataSet.createQuery().addCriteria(height.gt(5 * 12))
        .addOrderBy(height, Direction.DESCENDING)
        .addOrderBy(weight, Direction.DESCENDING).includeValues();

    // Execute the query...
    final Results results = query.execute();

    // Print the results...
    for (Result result : results.all()) {
      System.out.println(result.getValue());
    }
  }

  public static class Person {
    private final String firstName;
    private final String lastName;
    private final long id;
    private final int height;
    private final int weight;

    public Person(final long id, final String firstName, final String lastName,
        final int height, final int weight) {
      this.id = id;
      this.firstName = firstName;
      this.lastName = lastName;
      this.height = height;
      this.weight = weight;
    }

    public String getFirstName() {
      return firstName;
    }

    public String getLastName() {
      return lastName;
    }

    public int getHeight() {
      return height;
    }

    public int getWeight() {
      return weight;
    }

    public float getBodyMassIndex() {
      return ((float) weight / ((float) height * (float) height)) * 703;
```

```
    }

    public long getId() {
      return id;
    }

    public String toString() {
      return "[id=" + id + ", firstName=" + firstName + ", lastName="
          + lastName + ", height=" + height + " in" + ", weight=" + weight
          + " lbs" + ", bmi=" + getBodyMassIndex() + "]";
    }
  }
}
```

Execute

When you run the Sort program in a terminal, you should see output like this:

```
[id=2, firstName=Charles, lastName=Dickens, height=69 in, weight=160 lbs, bmi=23.625288]

[id=1, firstName=Jane, lastName=Austen, height=67 in, weight=130 lbs, bmi=20.358654]

[id=4, firstName=Alex, lastName=Smith, height=65 in, weight=160 lbs, bmi=26.622484]

[id=3, firstName=Jane, lastName=Doe, height=65 in, weight=145 lbs, bmi=24.126627]
```

The results are in descending order by height. The last two results both have the same height, so they are further ordered by weight.

Next Step

Next Step: Grouping Results ›

Grouping Results

In addition to sorting results, the Ehcache search interface also provides grouping and aggregation functionality. Let's try modifying our sample code to exercise some of these features.

Create Configuration File

Create a new configuration file called "ehcache-group.xml" in your classpath and add a new cache called "group." We're going to add a "Gender" attribute to our Person class, so we'll also add it as a searchAttribute in the configuration.

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
     name="SampleConfig" maxBytesLocalHeap="64M">
  <cache name="hello-world"/>
  <cache name="crud"/>
  <cache name="search">
    <searchable>
        <searchAttribute name="height"/>
        <searchAttribute name="weight"/>
        <searchAttribute name="bodyMassIndex" />
    </searchable>
  </cache>
  <cache name="sort">
    <searchable>
        <searchAttribute name="height"/>
        <searchAttribute name="weight"/>
    </searchable>
  </cache>
  <cache name="group">
    <searchable>
        <searchAttribute name="height"/>
        <searchAttribute name="gender"/>
        <searchAttribute name="bmi" expression="value.getBodyMassIndex()"/>
    </searchable>
  </cache>
</ehcache>
```

Create Group.java

Create and compile a Java class called Group:

```
import java.util.List;
import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Element;
import net.sf.ehcache.search.Attribute;
import net.sf.ehcache.search.Direction;
import net.sf.ehcache.search.Query;
import net.sf.ehcache.search.Result;
import net.sf.ehcache.search.Results;
import net.sf.ehcache.search.aggregator.Aggregators;

public class Group {
  public static void main(final String[] args) {
```

Create Group.java

```
// Create a cache manager using the factory method...
final CacheManager cacheManager = CacheManager.newInstance(Group.class
    .getResource("/ehcache-group.xml"));

// Retrieve the "sort" data set...
final Cache dataSet = cacheManager.getCache("group");

// Create some objects...
final Person janeAusten = new Person(1, "Jane", "Austen", "Female",
    5 * 12 + 7, 130);
final Person charlesDickens = new Person(2, "Charles", "Dickens", "Male",
    5 * 12 + 9, 160);
final Person janeDoe = new Person(3, "Jane", "Doe", "Female", 5 * 12 + 5,
    145);
final Person alexSmith = new Person(4, "Alex", "Smith", "Other",
    5 * 12 + 5, 160);

// Put the objects into the data set...
dataSet.put(new Element(janeAusten.getId(), janeAusten));
dataSet.put(new Element(charlesDickens.getId(), charlesDickens));
dataSet.put(new Element(janeDoe.getId(), janeDoe));
dataSet.put(new Element(alexSmith.getId(), alexSmith));

// Fetch the search attributes that we'll use in our query and when fetching
// our results...
Attribute<Integer> height = dataSet.getSearchAttribute("height");
Attribute<String> gender = dataSet.getSearchAttribute("gender");
Attribute<Float> bmi = dataSet.getSearchAttribute("bmi");

// Create a query object. (This query is designed to select the entire data
// set.)
final Query query = dataSet.createQuery().addCriteria(height.gt(5 * 12));

// Group by the gender attribute so we can perform aggregations on each
// gender...
query.addGroupBy(gender);
query.includeAttribute(gender);

// Include an aggregation of the average height and bmi of each gender in
// the results...
query.includeAggregator(Aggregators.average(height));
query.includeAggregator(Aggregators.average(bmi));

// Include the count of the members of each gender
query.includeAggregator(Aggregators.count());

// Make the results come back sorted by gender in alphabetical order
query.addOrderBy(gender, Direction.ASCENDING);

// Execute the query...
final Results results = query.execute();

// Print the results...
for (Result result : results.all()) {
  String theGender = result.getAttribute(gender);
  List<Object> aggregatorResults = result.getAggregatorResults();
  Float avgHeight = (Float) aggregatorResults.get(0);
  Float avgBMI = (Float) aggregatorResults.get(1);
  Integer count = (Integer) aggregatorResults.get(2);
  System.out.println("Gender: " + theGender + "; count: " + count
      + "; average height: " + avgHeight + "; average BMI: " + avgBMI);
}
```

```
  }

  public static class Person {
    private final String firstName;
    private final String lastName;
    private final long id;
    private final int height;
    private final int weight;
    private String gender;

    public Person(final long id, final String firstName, final String lastName,
        final String gender, final int height, final int weight) {
      this.id = id;
      this.firstName = firstName;
      this.lastName = lastName;
      this.gender = gender;
      this.height = height;
      this.weight = weight;
    }

    public String getFirstName() {
      return firstName;
    }

    public String getLastName() {
      return lastName;
    }

    public String getGender() {
      return gender;
    }

    public int getHeight() {
      return height;
    }

    public int getWeight() {
      return weight;
    }

    public float getBodyMassIndex() {
      return ((float) weight / ((float) height * (float) height)) * 703;
    }

    public long getId() {
      return id;
    }

    public String toString() {
      return "[id=" + id + ", firstName=" + firstName + ", lastName="
          + lastName + ", height=" + height + " in" + ", weight=" + weight
          + " lbs" + ", bmi=" + getBodyMassIndex() + "]";
    }
  }
}
```

Execute

When you run the Group program in a terminal, you should see output like this:

```
Gender: Female; count: 2; average height: 66.0; average BMI: 22.242641

Gender: Male; count: 1; average height: 69.0; average BMI: 23.625288

Gender: Other; count: 1; average height: 65.0; average BMI: 26.622484
```

Next Step

Next Step: Adding the Server Array ›

Adding the Terracotta Server Array

The Terracotta Server Array is an array of in-memory data servers that plug in seamlessly to provide applications with high-performance, low and predictable latency data access to terabytes of in-memory data and enterprise capabilities like high availability, scalability, fault tolerance and more.

Let's try starting up a Terracotta server and connecting our data sets to it.

Preparation

- Add a valid Terracotta license key to the top level of your Terracotta installation directory. If you don't have one yet, you can register for a license at terracotta.org/downloads/bigmemorymax
- Add the following jar in the bigmemory-max-4.x.x download kit to your classpath:
    - apis/toolkit/lib/terracotta-toolkit-runtime-ee-4.x.x.jar
- Add the terracotta configuration to the configuration file (see below)
- Make sure data classes serializable
- Start a Terracotta server instance

Create Configuration File

Create a new configuration file called "ehcache-server-array.xml" in your classpath and add a new cache called "server-array" (you can name it anything you want, as long as you use the correct name in your code).

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
     name="SampleConfig" maxBytesLocalHeap="64M">
  <cache name="hello-world"/>
  <cache name="crud"/>
  <cache name="search">
    <searchable>
        <searchAttribute name="height"/>
        <searchAttribute name="weight"/>
        <searchAttribute name="bodyMassIndex" />
    </searchable>
  </cache>
  <cache name="sort">
    <searchable>
        <searchAttribute name="height"/>
        <searchAttribute name="weight"/>
    </searchable>
  </cache>
  <cache name="group">
    <searchable>
        <searchAttribute name="height"/>
        <searchAttribute name="gender"/>
        <searchAttribute name="bmi" expression="value.getBodyMassIndex()"/>
    </searchable>
  </cache>
  <cache name="server-array">
    <searchable>
        <searchAttribute name="height"/>
        <searchAttribute name="gender"/>
        <searchAttribute name="bmi" expression="value.getBodyMassIndex()"/>
    </searchable>
      <!-- Add the terracotta element to this cache. This causes this data
```

```
      set to be managed by the Terracotta server array. For the purposes of
      demonstration, we set the consistency to "strong" to ensure that data
      is always consistent across the entire distributed system. There are
      other consistency settings that may be more suitable for different
      data sets and applications. -->
    <terracotta consistency="strong"/>
  </cache>
  <!-- Add the terracottaConfig element to specify where to find the
  configuration specific to the server array.  In this case, the configuration
  is retrieved from the server array itself. -->
  <terracottaConfig url="localhost:9510"/>
</ehcache>
```

Create ServerArrayTest.java

Create and compile a Java class called ServerArrayTest:

```
import java.io.Serializable;
import java.util.List;

import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Element;
import net.sf.ehcache.search.Attribute;
import net.sf.ehcache.search.Direction;
import net.sf.ehcache.search.Query;
import net.sf.ehcache.search.Result;
import net.sf.ehcache.search.Results;
import net.sf.ehcache.search.aggregator.Aggregators;

public class ServerArrayTest {

  public static void main(final String[] args) {

    // Create a cache manager using the factory method...
    final CacheManager cacheManager = CacheManager.newInstance(Group.class
        .getResource("/ehcache-server-array.xml"));

    // Retrieve the "sort" data set...
    final Cache dataSet = cacheManager.getCache("server-array");

    // Check to see if there's any data in it yet...
    Element person1 = dataSet.get(1L);
    if (person1 == null) {
      System.out.println("We didn't find any data in the server array."
          + " This must be the first execution.");
      // The data isn't in the server array yet (this must be the first time we
      // ran the program), so let's create some objects...
      final Person janeAusten = new Person(1, "Jane", "Austen", "Female",
          5 * 12 + 7, 130);
      final Person charlesDickens = new Person(2, "Charles", "Dickens", "Male",
          5 * 12 + 9, 160);
      final Person janeDoe = new Person(3, "Jane", "Doe", "Female", 5 * 12 + 5,
          145);
      final Person alexSmith = new Person(4, "Alex", "Smith", "Other",
          5 * 12 + 5, 160);

      // Put the objects into the data set...
      dataSet.put(new Element(janeAusten.getId(), janeAusten));
      dataSet.put(new Element(charlesDickens.getId(), charlesDickens));
```

```
      dataSet.put(new Element(janeDoe.getId(), janeDoe));
      dataSet.put(new Element(alexSmith.getId(), alexSmith));
    } else {

      System.out.println("We found data in the server array from a previous"
          + " execution. No need to create new data.");

    }

    // Fetch the search attributes that we'll use in our query and when fetching
    // our results...
    Attribute<Integer> height = dataSet.getSearchAttribute("height");
    Attribute<String> gender = dataSet.getSearchAttribute("gender");
    Attribute<Float> bmi = dataSet.getSearchAttribute("bmi");

    // Create a query object. (This query is designed to select the entire data
    // set.)
    final Query query = dataSet.createQuery().addCriteria(height.gt(5 * 12));

    // Group by the gender attribute so we can perform aggregations on each
    // gender...
    query.addGroupBy(gender);
    query.includeAttribute(gender);

    // Include an aggregation of the average height and bmi of each gender in
    // the results...
    query.includeAggregator(Aggregators.average(height));
    query.includeAggregator(Aggregators.average(bmi));

    // Include the count of the members of each gender
    query.includeAggregator(Aggregators.count());

    // Make the results come back sorted by gender in alphabetical order
    query.addOrderBy(gender, Direction.ASCENDING);

    // Execute the query...
    final Results results = query.execute();

    // Print the results...
    for (Result result : results.all()) {
      String theGender = result.getAttribute(gender);
      List<Object> aggregatorResults = result.getAggregatorResults();
      Float avgHeight = (Float) aggregatorResults.get(0);
      Float avgBMI = (Float) aggregatorResults.get(1);
      Integer count = (Integer) aggregatorResults.get(2);
      System.out.println("Gender: " + theGender + "; count: " + count
          + "; average height: " + avgHeight + "; average BMI: " + avgBMI);
    }
  }

  public static class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private final String firstName;
    private final String lastName;
    private final long id;
    private final int height;
    private final int weight;
    private String gender;

    public Person(final long id, final String firstName, final String lastName,
        final String gender, final int height, final int weight) {
      this.id = id;
```

```
      this.firstName = firstName;
      this.lastName = lastName;
      this.gender = gender;
      this.height = height;
      this.weight = weight;
    }

    public String getFirstName() {
      return firstName;
    }

    public String getLastName() {
      return lastName;
    }

    public String getGender() {
      return gender;
    }

    public int getHeight() {
      return height;
    }

    public int getWeight() {
      return weight;
    }

    public float getBodyMassIndex() {
      return ((float) weight / ((float) height * (float) height)) * 703;
    }

    public long getId() {
      return id;
    }

    public String toString() {
      return "[id=" + id + ", firstName=" + firstName + ", lastName="
          + lastName + ", height=" + height + " in" + ", weight=" + weight
          + " lbs" + ", bmi=" + getBodyMassIndex() + "]";
    }
  }
}
```

Start a Terracotta Server Instance

In a terminal, go to the "server" directory in your BigMemory Max distribution. Then run the start-tc-server command:

```
%> cd /path/to/bigmemory-max-4.x.x/server
%> ./bin/start-tc-server.sh
```

Execute

The first time you run the ServerArrayTest program in a terminal, there is no data in the server array, so you should see output like this:

```
We didn't find any data in the server array. This must be the first execution.
```

Execute

```
Gender: Female; count: 2; average height: 66.0; average BMI: 22.242641

Gender: Male; count: 1; average height: 69.0; average BMI: 23.625288

Gender: Other; count: 1; average height: 65.0; average BMI: 26.622484
```

If you run the ServerArrayTest program again (without restarting the Terracotta server instance that is already running), the data will be automatically retrieved from the server array, so you should see output like this:

```
We found data in the server array from a previous execution. No need to create new data.

Gender: Female; count: 2; average height: 66.0; average BMI: 22.242641

Gender: Male; count: 1; average height: 69.0; average BMI: 23.625288

Gender: Other; count: 1; average height: 65.0; average BMI: 26.622484
```

In our default configuration, the server keeps all data in memory only (though, you can add easily add data persistence with a fast restartable store configuration). To clear all data, restart the server before you next run the ServerArrayTest program.

Next Steps

This is just a short example of using BigMemory Max with the full power of the server array to help get you started. Keep in mind that the settings we used here were chosen to make it as easy as possible to get started. We encourage you to read the rest of the documentation and experiment with the different configuration options, features and deployment topologies available to best meet your application's performance and scale needs.

BigMemory Max Quick Start

Install BigMemory Max

Installing BigMemory Max is as easy as downloading the kit and ensuring that the correct files are on your application's classpath. The only platform requirement is using JDK 1.6 or higher.

1. If you do not have a BigMemory Max kit, download it from here.

   The kit is packaged as a tar.gz file. Unpack it on the command line or with the appropriate decompression application.
2. Add the following JARs from in the kit to your application's classpath:

   - `common/lib/bigmemory-<version>.jar` – This is the main JAR to enable BigMemory.
   - `apis/ehcache/lib/ehcache-ee-<version>.jar` – This file contains the API to BigMemory Max.
   - `apis/ehcache/lib/slf4j-api-<version>.jar` – This file is the bridge, or logging facade, to the BigMemory Max logging framework.
   - `apis/ehcache/lib/slf4j-jdk14-<version>.jar` – This is a binding JAR for the provided SLF4J logging framework, java.util.logging. Binding JARs for other frameworks are available from the SLF4J website.
   - `apis/toolkit/lib/terracotta-toolkit-runtime-ee-<version>.jar` – This JAR contains the libraries for the Terracotta Server Array.
3. Save the BigMemory Max license-key file to the BigMemory Max home directory. This file, called `terracotta-license.key`, was attached to an email you received after registering for the BigMemory Max download.

   Alternatively, you can add the license-key file to your application's classpath, or specify it with the following Java system property:

   ```
   -Dcom.tc.productkey.path=/path/to/terracotta-license.key
   ```
4. BigMemory Max uses Ehcache as its user-facing interface. To configure BigMemory Max, create an `ehcache.xml` configuration file, or update the one that is provided in the config-samples/ directory of the BigMemory Max kit. For example:

   ```
   <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
        name="myBigMemoryMaxConfig">

     <!-- Tell BigMemory where to write its data to disk. -->
     <diskStore path="/path/to/my/disk/store/directory"/>

     <!-- Set 'maxBytesLocalOffHeap' to the amount of off-heap in-memory
     storage you want to use. This memory is invisible to the Java garbage
     collector, providing for gigabytes to terabytes of in-memory data without
     garbage collection pauses. -->
     <cache name="myBigMemoryMaxStore"
          maxBytesLocalHeap="512M"
          maxBytesLocalOffHeap="8G">

       <!-- Tell BigMemory to use the "localRestartable" persistence
       strategy for fast restart (optional). -->
   ```

```
    <persistence strategy="localRestartable"/>

    <!-- Include the terracotta element so that the data set will be
    managed as a client of the Terracotta server array.  -->
    <terracotta/>
  </cache>

  <!-- Specify where to find the server array configuration. In this
  case, the configuration is retrieved from the local server. -->
  <terracottaConfig url="localhost:9510" />

</ehcache>
```

Place your `ehcache.xml` file in the top-level of your classpath.

For more information on configuration options, refer to the configuration documentation and to the reference ehcache.xml configuration file in the `config-samples` directory of the BigMemory Max kit.

5. Use the `-XX:MaxDirectMemorySize` Java option to allocate enough direct memory in the JVM to accomodate the off-heap storage specified in your configuration, plus at least 250MB to allow for other direct memory usage that might occur in your application. For example:

   ```
   -XX:MaxDirectMemorySize=9G
   ```

   Set `MaxDirectMemorySize` to the amount of BigMemory you have. For more information about this step, refer to Allocating Direct Memory in the JVM.

   Also, allocate at least enough heap using the `-Xmx` Java option to accomodate the on-heap storage specified in your configuration, plus enough extra heap to run the rest of your application. For example:

   ```
   -Xmx1g
   ```

   Finally, if necessary, define the JAVA_HOME environment variable.
6. Learn BigMemory basics starting with Hello, World!, or look through the code samples for examples of how to employ the various features and capabilities of BigMemory Max.

Start the Terracotta Server and Management Console

Large data sets in BigMemory Max can be distributed across the Terracotta Server Array (TSA) and managed with the Terracotta Management Console (TMC).

1. To configure the Terracotta server, create a `tc-config.xml` configuration file, or update the one that is provided in the config-samples/ directory of the BigMemory Max kit. For example:

   ```
   <?xml version="1.0" encoding="UTF-8" ?>
   <tc:tc-config xmlns:tc="http://www.terracotta.org/config"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-8.xsd">
     <servers>
       <server host="localhost" name="My Server Name">
         <!-- Specify the path where the server should store its data. -->
         <data>/local/disk/path/to/terracotta/server1-data</data>
          <!-- Specify the port where the server should listen for client
          traffic. -->
   ```

```
        <tsa-port>9510</tsa-port>
        <jmx-port>9520</jmx-port>
        <tsa-group-port>9530</tsa-group-port>
        <!-- Enable BigMemory on the server. -->
        <offheap>
          <enabled>true</enabled>
          <maxDataSize>4g</maxDataSize>
        </offheap>
      </server>
      <!-- Add the restartable element for Fast Restartability (optional). -->
      <restartable enabled="true"/>
    </servers>
    <clients>
      <logs>logs-%i</logs>
    </clients>
</tc:tc-config>
```

Place your `tc-config.xml` file in the Terracotta server/ directory.

For more information about configuration options, refer to the TSA configuration documentation.

2. In a terminal, change to your Terracotta server/ directory. Then execute the start-tc-server command:

```
%> cd /path/to/bigmemory-max-<version>/server
%> ./bin/start-tc-server.sh
```

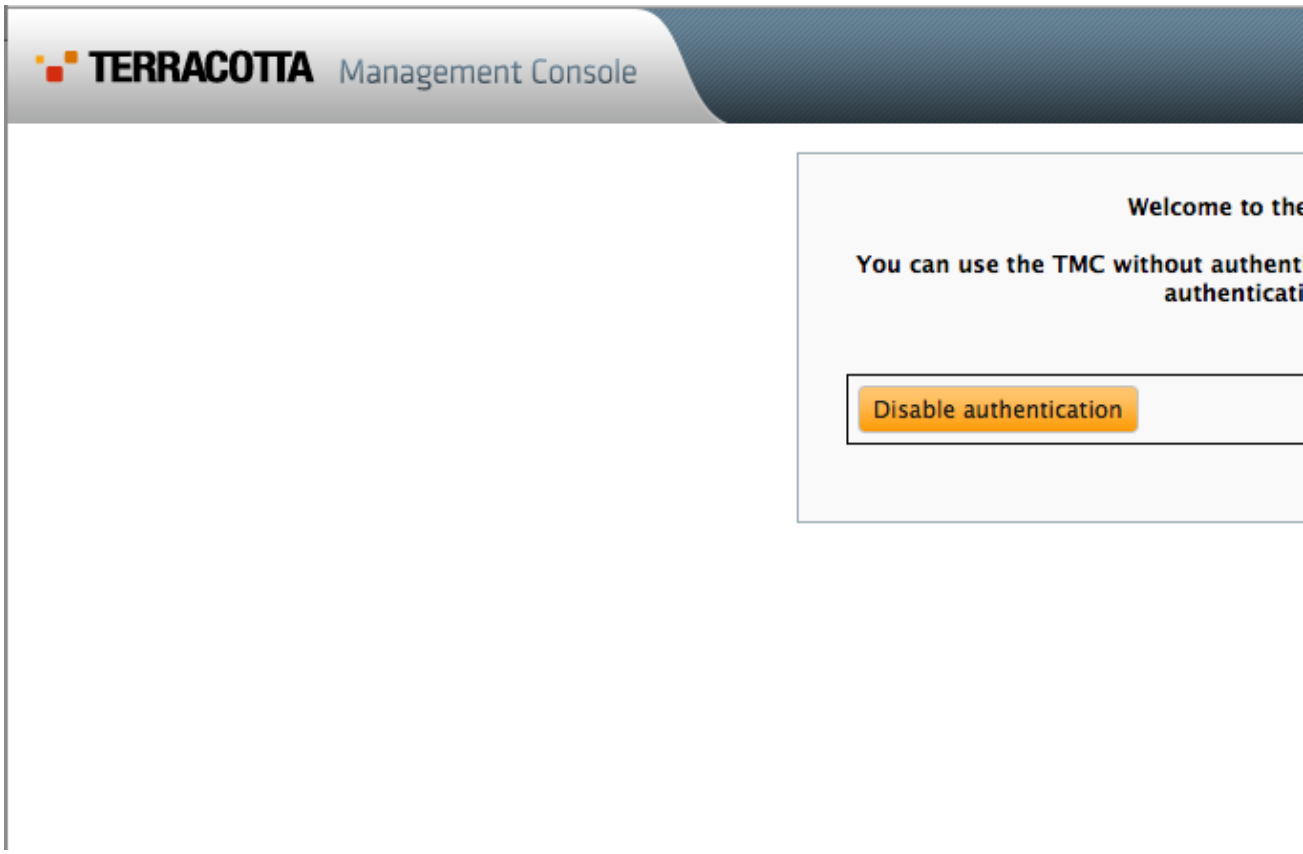
You should see confirmation in the terminal that the server started.

Note: For Microsoft Windows installations, use the BAT scripts, and where forward slashes ("/") are given in directory paths, substitute back slashes ("\").

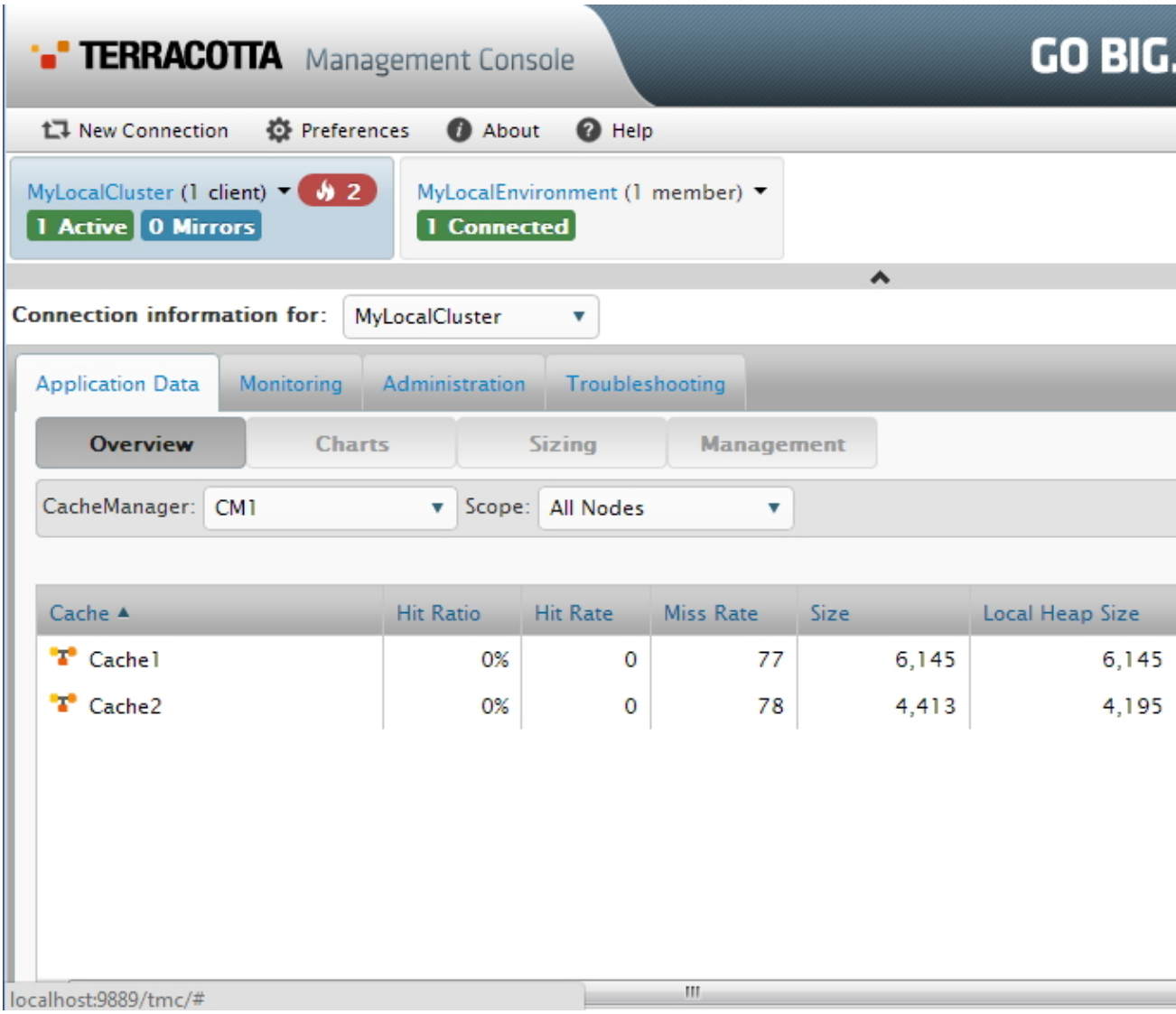
3. In a terminal, change to your Terracotta tools/management-console/ directory. Then execute the start-tmc command:

```
%> cd /path/to/bigmemory-max-<version>/tools/management-console
%> ./bin/start-tmc.sh
```

4. In a browser, enter the URL `http://localhost:9889/tmc`. When you first connect to the TMC, the authentication setup page appears, where you can choose to run the TMC with authentication or without.



5. Use the TMC to manage all of the clients and servers in your deployment. For more information about the TMC, refer to the TMC documentation.



Additional Configuration Topics

For a general overview to configuring BigMemory Max, see this introductory page. Specific configuration topics are introduced below.

Automatic Resource Control

Automatic Resource Control (ARC) gives you fine-grained controls for tuning performance and enabling trade-offs between throughput, latency and data access. Independently adjustable configuration parameters include differentiated tier-based sizing and pinning hot or eternal data in the most effective tier.

Dynamically Sizing Stores

Tuning often involves sizing stores appropriately. There are a number of ways to size the different BigMemory Max data tiers using simple configuration sizing attributes. The sizing page explains how to tune tier sizing by configuring dynamic allocation of memory and automatic balancing.

Pinning Data

One of the most important aspects of running an in-memory data store involves managing the life of the data in each BigMemory Max tier. See the data-life page for more information on the pinning, expiration, and eviction of data.

Fast Restartability

BigMemory Max has full fault tolerance, allowing for continuous access to in-memory data after a planned or unplanned shutdown, with the option to store a fully consistent record of the in-memory data on the local disk at all times. The fast-restart page covers data persistence, fast restartability, and using the local disk as a storage tier for in-memory data (both heap and off-heap stores).

Using the BigMemory Max API

BigMemory Max provides a full-featured API. See the code-samples page for a beginner's view of using the API. Selected advanced API features are introduced below.

Search

Search billions of entries—gigabytes, even terabytes of data—with results returned in less than a second. Data is indexed without significant overhead, and features like "GroupBy' are included.

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

Transactional Caching

Transactional modes are a powerful extension for performing atomic operations on data stores, keeping your data in sync with your database.

The transactions page covers the background and configuration information for BigMemory Max transactional modes. Explicit Locking is another API that can be used as a custom alternative to XA Transactions or Local transactions.

Administration and Monitoring

The Terracotta Management Console (TMC) is a web-based monitoring and administration application for tuning cache usage, detecting errors, and providing an easy-to-use access point to integrate with production management systems.

As an alternative to the TMC, standard JMX-based administration and monitoring is available.

For logging, BigMemory Max uses the flexible SLF4J logging framework.

Scale Up and Scale Out

- See the Distributed Configuration page to learn more about configuration for large amounts of

in-memory data.

- See the Terracotta Server Array pages to learn how to use the potential of BigMemory Max.

Code Samples

This page is a companion to the code samples provided in the BigMemory Max kit. For BigMemory tutorials, visit Hello, World!

Introduction

The following code samples illustrate various features of BigMemory Max. They are also available in the BigMemory Max kit in the /code-samples directory.

Title	Description
example01-config-file	BigMemory may be configured declaratively, using an XML configuration file, or programmatically via the fluent configuration API. This sample shows how to configure a basic instance of BigMemory Max declaratively with the XML configuration file.
example02-config-programmatic	Configure a basic instance of BigMemory Max programmatically with the fluent configuration API.
example03-crud	Basic create, retrieve, update and delete (CRUD) operations available in BigMemory Max.
example04-search	Basic in-memory search features of BigMemory Max.
example05-nonstop	The nonstop feature allows operations to proceed on clients that have become disconnected from the cluster. The rejoin feature then allows clients to identify a source of Terracotta configuration and rejoin a cluster. This sample demonstrates the nonstop and rejoin features of BigMemory Max.
example06-arc	Automatic Resource Control (ARC) is a powerful capability of BigMemory Max that gives users the ability to control how much data is stored in heap memory and off-heap memory. This sample shows the basic configuration options for data tier sizing using ARC.
example07-cache	BigMemory Max is a powerful in-memory data management solution. Among its many applications, BigMemory Max may be used as a cache to speed up access to data from slow or expensive databases and other remote data sources. This example shows how to enable and configure the caching features available in BigMemory Max.

To run the code samples with Maven, you will need to add the Terracotta Maven repositories to your Maven settings.xml file. Add the following repository information to your settings.xml file:

```
<repository>
    <id>terracotta-repository</id>
    <url>http://www.terracotta.org/download/reflector/releases</url>
    <releases>
        <enabled>true</enabled>
    </releases>
</repository>
```

For further information, refer to Working with Apache Maven.

Example 1: Declarative Configuration via XML

To configure BigMemory declaratively with an XML file, create a CacheManager instance, passing the a file name or an URL object to the constructor.

The following example shows how to create a CacheManager with an URL of the XML file in the classpath at `/xml/ehcache.xml`.

```
CacheManager manager = CacheManager.newInstance(
                          getClass().getResource("/xml/ehcache.xml"));
try {
  Cache bigMemory = manager.getCache("BigMemory");
  // now do stuff with it...

} finally {
  if (manager != null) manager.shutdown();
}
```

Here are the contents of the XML configuration file used by this sample:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
         name="config">
  <cache name="BigMemory"
         maxBytesLocalHeap="512M"
         maxBytesLocalOffHeap="32G"
         copyOnRead="true"
         statistics="true"
         eternal="true">
  </cache>
</ehcache>
```

The configuration element `maxBytesLocalOffHeap` lets you set how much off-heap memory to use. BigMemory's unique off-heap memory storage lets you use all of the memory available on a server in a single JVM—from gigabytes to multiple terabytes—without causing garbage collection pauses.

Example 2: Programmatic Configuration

To configure BigMemory Max programmatically, use the Ehcache fluent configuration API.

```
Configuration managerConfiguration = new Configuration()
    .name("bigmemory-config")
    .cache(new CacheConfiguration()
        .name("BigMemory")
        .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
        .maxBytesLocalOffHeap(1, MemoryUnit.GIGABYTES)
        .copyOnRead(true)
        .statistics(true)
        .eternal(true)
    );

CacheManager manager = CacheManager.create(managerConfiguration);
try {
  Cache bigMemory = manager.getCache("BigMemory");
  // now do stuff with it...
```

```
} finally {
  if (manager != null) manager.shutdown();
}
```

Example 3: Create, Read, Update and Delete (CRUD)

The CRUD sample demonstrates basic create, read, update and delete operations.

First, we create a BigMemory data store configured to use 128 MB of heap memory:

```
Configuration managerConfiguration = new Configuration();
managerConfiguration.name("config")
    .terracotta(new TerracottaClientConfiguration().url("localhost:9510"))
    .cache(new CacheConfiguration()
        .name("bigMemory-crud")
        .maxBytesLocalHeap(128, MemoryUnit.MEGABYTES)
        .terracotta(new TerracottaConfiguration())
    );

CacheManager manager = CacheManager.create(managerConfiguration);
Cache bigMemory = manager.getCache("bigMemory-crud");
```

Now that we have a BigMemory instance configured and available, we can start creating data in it:

```
final Person timDoe = new Person("Tim Doe", 35, Person.Gender.MALE,
    "eck street", "San Mateo", "CA");
bigMemory.put(new Element("1", timDoe));
```

Then, we can read data from it:

```
final Element element = bigMemory.get("1");
System.out.println("The value for key 1 is  " + element.getObjectValue());
```

And update it:

```
final Person pamelaJones = new Person("Pamela Jones", 23, Person.Gender.FEMALE,
    "berry st", "Parsippany", "LA");
bigMemory.put(new Element("1", pamelaJones));
final Element updated = bigMemory.get("1");
System.out.println("The value for key 1 is now " + updated.getObjectValue() +
                     ". key 1 has been updated.");
```

And delete it:

```
bigMemory.remove("1");
System.out.println("Try to retrieve key 1.");
final Element removed = bigMemory.get("1");
System.out.println("Value for key 1 is " + removed +
                   ". Key 1 has been deleted.");
```

You can also create or update multiple entries at once:

```
Collection<Element> elements = new ArrayList<Element>();
elements.add(new Element("1", new Person("Jane Doe", 35,
    Person.Gender.FEMALE, "eck street", "San Mateo", "CA")));
elements.add(new Element("2", new Person("Marie Antoinette", 23,
    Person.Gender.FEMALE, "berry st", "Parsippany", "LA")));
```

```
elements.add(new Element("3", new Person("John Smith", 25,
    Person.Gender.MALE, "big wig", "Beverly Hills", "NJ")));
elements.add(new Element("4", new Person("Paul Dupont", 25,
    Person.Gender.MALE, "big wig", "Beverly Hills", "NJ")));
elements.add(new Element("5", new Person("Juliet Capulet", 25,
    Person.Gender.FEMALE, "big wig", "Beverly Hills", "NJ")));

bigMemory.putAll(elements);
```

And read multiple entries at once:

```
final Map<Object, Element> elementsMap = bigMemory.getAll(
    Arrays.asList("1", "2", "3"));
```

And delete multiple entries at once:

```
bigMemory.removeAll(Arrays.asList("1", "2", "3"));
```

And delete everything at once:

```
bigMemory.removeAll();
```

Example 4: Search

BigMemory Max comes with powerful in-memory search capabilities. This sample shows how to perform basic search operations on your in-memory data.

First, create an instance of BigMemory Max with searchable attributes:

```
Configuration managerConfig = new Configuration()
    .cache(new CacheConfiguration().name("MySearchableDataStore")
        .eternal(true)
        .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
        .maxBytesLocalOffHeap(32, MemoryUnit.GIGABYTES)
        .searchable(new Searchable()
            .searchAttribute(new SearchAttribute().name("age"))
            .searchAttribute(new SearchAttribute().name("gender")
                .expression("value.getGender()"))
            .searchAttribute(new SearchAttribute().name("state")
                .expression("value.getAddress().getState()"))
            .searchAttribute(new SearchAttribute().name("name")
                .className(NameAttributeExtractor.class.getName()))
        )
    );

CacheManager manager = CacheManager.create(managerConfig);
Ehcache bigMemory = manager.getEhcache("MySearchableDataStore");
```

Now, let's put a bunch of stuff into it:

```
bigMemory.put(new Element(1, new Person("Jane Doe", 35, Gender.FEMALE,
    "eck street", "San Mateo", "CA")));
bigMemory.put(new Element(2, new Person("Marie Antoinette", 23, Gender.FEMALE,
    "berry st", "Parsippany", "LA")));
bigMemory.put(new Element(3, new Person("John Smith", 25, Gender.MALE,
    "big wig", "Beverly Hills", "NJ")));
bigMemory.put(new Element(4, new Person("Paul Dupont", 45, Gender.MALE,
```

```
    "cool agent", "Madison", "WI")));
bigMemory.put(new Element(5, new Person("Juliet Capulet", 30, Gender.FEMALE,
    "dah man", "Bangladesh", "MN")));
for (int i = 6; i < 1000; i++) {
  bigMemory.put(new Element(i, new Person("Juliet Capulet" + i, 30,
      Person.Gender.MALE, "dah man", "Bangladesh", "NJ")));
}
```

Next, create some search attributes and construct a query:

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

Query query = bigMemory.createQuery();
query.includeKeys();
query.includeValues();
query.addCriteria(name.ilike("Jul*").and(gender.eq(Gender.FEMALE)))
    .addOrderBy(age, Direction.ASCENDING).maxResults(10);
```

Then, execute the query and look at the results:

```
Results results = query.execute();
System.out.println(" Size: " + results.size());
System.out.println("----Results-----\n");
for (Result result : results.all()) {
  System.out.println("Maxt: Key[" + result.getKey()
                     + "] Value class [" + result.getValue().getClass()
                     + "] Value [" + result.getValue() + "]");
}
```

We can also use Aggregators to perform computations across query results. Here's an example that computes the average age of all people in the data set:

```
Query averageAgeQuery = bigMemory.createQuery();
averageAgeQuery.includeAggregator(Aggregators.average(age));
System.out.println("Average age: "
                   + averageAgeQuery.execute().all().iterator().next()
                       .getAggregatorResults());
```

We can also restrict the calculation to a subset based on search attributes. Here's an example that computes the average age of all people in the data set between the ages of 30 and 40:

```
Query agesBetween = bigMemory.createQuery();
agesBetween.addCriteria(age.between(30, 40));
agesBetween.includeAggregator(Aggregators.average(age));
System.out.println("Average age between 30 and 40: "
                   + agesBetween.execute().all().iterator().next()
                       .getAggregatorResults());
```

Using Aggregators, we can also find number of entries that match our search criteria. Here's an example that finds the number of people in the data set who live in New Jersey:

```
Query newJerseyCountQuery = bigMemory.createQuery().addCriteria(
                                state.eq("NJ"));
newJerseyCountQuery.includeAggregator(Aggregators.count());
System.out.println("Count of people from NJ: "
```

```
                + newJerseyCountQuery.execute().all().iterator().next()
                    .getAggregatorResults());
```

Example 5: Nonstop/Rejoin

The nonstop feature allows certain operations to proceed on clients that have become disconnected from the cluster, and it allows operations to proceed even if they cannot complete by the nonstop timeout value. The rejoin feature then allows clients to identify a source of Terracotta configuration, so that clients can rejoin a cluster after having been either disconnected from that cluster or timed out by a Terracotta server.

This example demonstrates the nonstop and rejoin features of BigMemory Max. After loading the configuration below, use the scripts provided to start and run the server. Then stop the server and verify that the nonstop feature is working. Start the server again and watch the rejoin feature in action.

```
Configuration managerConfig = new Configuration()
    .terracotta(new TerracottaClientConfiguration().url("localhost:9510").rejoin(true))
    .cache(new CacheConfiguration().name("nonstop-sample")
        .persistence(new PersistenceConfiguration().strategy(DISTRIBUTED))
        .maxBytesLocalHeap(128, MemoryUnit.MEGABYTES)
        .maxBytesLocalOffHeap(1, MemoryUnit.GIGABYTES)
        .terracotta(new TerracottaConfiguration()
        .nonstop(new NonstopConfiguration()
        .immediateTimeout(false)
            .timeoutMillis(10000).enabled(true)
        )
        ));

CacheManager manager = CacheManager.create(managerConfig);
Ehcache bigMemory = manager.getEhcache("nonstop-sample");

try {
  System.out.println("**** Put key 1 / value timDoe. ****");
  final Person timDoe = new Person("Tim Doe", 35, Person.Gender.MALE,
      "eck street", "San Mateo", "CA");
  bigMemory.put(new Element("1", timDoe));

  System.out.println("**** Get key 1 / value timDoe. ****");
  System.out.println(bigMemory.get("1"));
  waitForInput();

  System.out
      .println("**** Now you have to kill the server using the
       stop-sample-server.bat on Windows or stop-sample-server.sh otherwise ****");
  try {
    while (true) {
      bigMemory.get("1");
    }
  } catch (NonStopCacheException e) {
    System.out
        .println("**** Server is unreachable - NonStopException received when trying
        to do a get on the server. NonStop is working ****");
  }

  System.out
      .println("**** Now you have to restart the server using the start-sample-server.bat
      on Windows or start-sample-server.sh otherwise ****");

  boolean serverStart = false;
  while (serverStart  == false) {
```

```
    try {
      bigMemory.get("1");
      //if server is unreachable, exception is thrown when doing the get
      serverStart = true;
    } catch (NonStopCacheException e) {
    }
  }
  System.out
      .println("**** Server is reachable - No More NonStopException received when
      trying to do a get on the server. Rejoin is working ****");
} finally

{
  if (manager != null) manager.shutdown();
}
```

Example 6: Automatic Resource Control (ARC)

Automatic Resource Control (ARC) is a powerful capability of BigMemory Max that gives users the ability to control how much data is stored in heap memory and off-heap memory.

The following XML configuration instructs ARC to allocate a maximum of 512 M of heap memory and 8 G of off-heap memory. In this example, when 512 M of heap memory is used, ARC will automatically move data into off-heap memory up to a maximum of 8 G. The amount of off-heap memory you can use is limited only by the amount of physical RAM you have available.

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
     updateCheck="false" monitoring="autodetect"
     dynamicConfig="true"
     name="MyManager" maxBytesLocalHeap="512M"
     maxBytesLocalOffHeap="8G">

  <defaultCache>
  </defaultCache>

  <cache name="BigMemory1">
  </cache>

  <cache name="BigMemory2">
  </cache>

</ehcache>
```

This instructs the ARC capability of BigMemory to keep a maximum of 512 MB of its data in heap for nanosecond to microsecond access. In this example, as the 512 MB of heap memory fills up, ARC will automatically move data to the 8 GB off-heap store where it is available at microsecond speed. This configuration keeps heap sizes small to avoid garbage collection pauses and tuning, but still uses large amounts of in-process memory for ultra-fast access to data.

Important: *BigMemory Max is capable of addressing gigabytes to terabytes of in-memory data in a single JVM and it's free to use up to 8 GB. However, to avoid swapping take care not to configure BigMemory Max to use more memory than is physically available on your hardware.*

It's also possible to allocate resources on a per-data set basis. Here's an example of allocating 4 G of off-heap memory to the BigMemory1 data set and 4 G of off-heap memory to the BigMemory2 data set:

```
<ehcache xmlns
 ...
     name="MyManager"
     maxBytesLocalHeap="512M"
     maxBytesLocalOffHeap="32G"
     maxBytesLocalDisk="128G">

  <cache name="BigMemory1"
         maxBytesLocalOffHeap="4G">
  </cache>

  <cache name="BigMemory2"
         maxBytesLocalOffHeap="4G">
  </cache>

</ehcache>
```

Example 7: Using BigMemory As a Cache

BigMemory Max is a powerful in-memory data management solution. Among its many applications, BigMemory Max may be used as a cache to speed up access to data from slow or expensive databases and other remote data sources. This example shows how to enable and configure the caching features available in BigMemory Max.

The following programmatic configuration snippet shows how to set time-to-live (TTL) and time-to-idle (TTI) policies on a data set:

```
Configuration managerConfiguration = new Configuration();
managerConfiguration.updateCheck(true)
    .monitoring(Configuration.Monitoring.AUTODETECT)
    .name("cacheManagerCompleteExample")
    .addCache(
        new CacheConfiguration()
            .name("sample-cache")
            .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
            .maxBytesLocalOffHeap(1, MemoryUnit.GIGABYTES)
            .timeToLiveSeconds(4)
            .timeToIdleSeconds(2)

    );

CacheManager manager = CacheManager.create(managerConfiguration);
```

The `timeToLiveSeconds` directive sets the maximum age of an element in the data set. Elements older than the maximum TTL will not be returned from the data store. This is useful when BigMemory is used as a cache of external data and you want to ensure the freshness of the cache.

The `timeToIdleSeconds` directive sets the maximum time since last access of an element. Elements that have been idle longer than the maximum TTI will not be returned from the data store. This is useful when BigMemory is being used as a cache of external data and you want to bias the eviction algorithm towards removing idle entries.

If neither TTL nor TTI are set (or set to zero), data will stay in BigMemory until it is explicitly removed.

Configuring BigMemory Max

Introduction

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

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

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

This documentation focuses on XML declarative configuration. Programmatic configuration is explored in certain examples and is documented in Javadocs.

XML Configuration

BigMemory Max uses Ehcache as its user-facing interface and is configured using the Ehcache configuration system. By default, Ehcache looks for an ASCII or UTF8 encoded XML configuration file called `ehcache.xml` at the top level of the Java classpath. You may specify alternate paths and filenames for the XML configuration file by using the various CacheManager constructors.

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

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

ehcache.xsd

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

Each BigMemory Max distribution also contains a copy of `ehcache.xsd`.

ehcache-failsafe.xml

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

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

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

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

About Default Cache

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

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

Quick Start for BigMemory Max

Configuration involves adding BigMemory Max to your application's installation and setting up the Terracotta server.

Install and Configure BigMemory Max

1. If you do not have a BigMemory Max kit, download it from here.

   The kit is packaged as a tar.gz file. Unpack it on the command line or with the appropriate decompression application.
2. Add the following JARs from in the kit to your application's classpath:

   - `common/lib/bigmemory-<version>.jar` – This is the main JAR to enable BigMemory.
   - `apis/ehcache/lib/ehcache-ee-<version>.jar` – This file contains the API to BigMemory Max.
   - `apis/ehcache/lib/slf4j-api-<version>.jar` – This file is the bridge, or logging facade, to the BigMemory Max logging framework.
   - `apis/ehcache/lib/slf4j-jdk14-<version>.jar` – This is a binding JAR for the provided SLF4J logging framework, java.util.logging. Binding JARs for other frameworks are available from the SLF4J website.

- `apis/toolkit/lib/terracotta-toolkit-runtime-ee-<version>.jar` – This JAR contains the libraries for the Terracotta Server Array.

3. Save the BigMemory Max license-key file to the BigMemory Max home directory. This file, called `terracotta-license.key`, was attached to an email you received after registering for the BigMemory Max download.

   Alternatively, you can add the license-key file to your application's classpath, or specify it with the following Java system property:

   ```
   -Dcom.tc.productkey.path=/path/to/terracotta-license.key
   ```

4. BigMemory Max uses Ehcache as its user-facing interface. To configure BigMemory Max, create an `ehcache.xml` configuration file, or update the one that is provided in the config-samples/ directory of the BigMemory Max kit. For example:

   ```
   <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
        name="myBigMemoryMaxConfig">

     <!-- Tell BigMemory where to write its data to disk. -->
     <diskStore path="/path/to/my/disk/store/directory"/>

     <!-- Set 'maxBytesLocalOffHeap' to the amount of off-heap in-memory
     storage you want to use. This memory is invisible to the Java garbage
     collector, providing for gigabytes to terabytes of in-memory data without
     garbage collection pauses. -->
     <cache name="myBigMemoryMaxStore"
           maxBytesLocalHeap="512M"
           maxBytesLocalOffHeap="8G">

       <!-- Tell BigMemory to use the "localRestartable" persistence
       strategy for fast restart (optional). -->
       <persistence strategy="localRestartable"/>

       <!-- Include the terracotta element so that the data set will be
       managed as a client of the Terracotta server array.  -->
       <terracotta/>

     <!-- Specify where to find the server array configuration. In this
     case, the configuration is retrieved from the local server. -->
     <terracottaConfig url="localhost:9510" />

     </cache>
   </ehcache>
   ```

   Place your `ehcache.xml` file in the top-level of your classpath.

5. Use the `-XX:MaxDirectMemorySize` Java option to allocate enough direct memory in the JVM to accomodate the off-heap storage specified in your configuration, plus at least 250MB to allow for other direct memory usage that might occur in your application. For example:

   ```
   -XX:MaxDirectMemorySize=9G
   ```

   Set `MaxDirectMemorySize` to the amount of BigMemory you have. For more information about this step, refer to Allocating Direct Memory in the JVM.

   Also, allocate at least enough heap using the `-Xmx` Java option to accomodate the on-heap storage specified in your configuration, plus enough extra heap to run the rest of your application. For

example:

```
-Xmx1g
```

Finally, if necessary, define the JAVA_HOME environment variable.

Start the Terracotta Server and Management Console

Large data sets in BigMemory Max can be distributed across the Terracotta Server Array (TSA) and managed with the Terracotta Management Console (TMC).

1. To configure the Terracotta server, create a `tc-config.xml` configuration file, or update the one that is provided in the config-samples/ directory of the BigMemory Max kit. For example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-8.xsd">
  <servers>
    <server host="localhost" name="My Server Name">
      <!-- Specify the path where the server should store its data. -->
      <data>/local/disk/path/to/terracotta/server1-data</data>
       <!-- Specify the port where the server should listen for client
       traffic. -->
       <tsa-port>9510</tsa-port>
       <jmx-port>9520</jmx-port>
       <tsa-group-port>9530</tsa-group-port>
       <!-- Enable BigMemory on the server. -->
       <offheap>
         <enabled>true</enabled>
         <maxDataSize>4g</maxDataSize>
       </offheap>
     </server>
    <!-- Add the restartable element for Fast Restartability (optional). -->
    <restartable enabled="true"/>
  </servers>
  <clients>
    <logs>logs-%i</logs>
  </clients>
</tc:tc-config>
```

   Place your `tc-config.xml` file in the Terracotta server/ directory.

   For more information about configuration options, refer to the TSA configuration documentation.
2. In a terminal, change to your Terracotta server/ directory. Then execute the start-tc-server command:

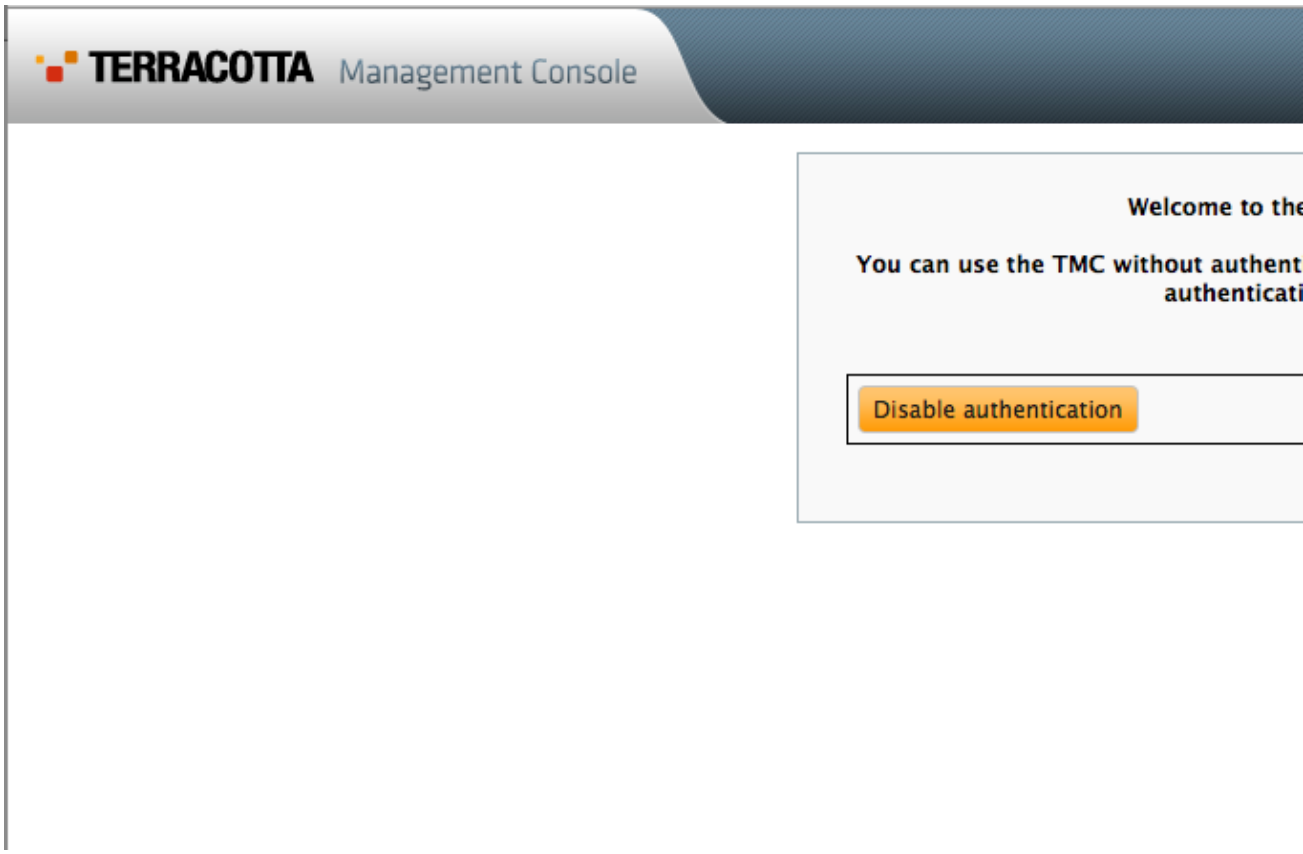
```
%> cd /path/to/bigmemory-max-<version>/server
%> ./bin/start-tc-server.sh
```

   You should see confirmation in the terminal that the server started.

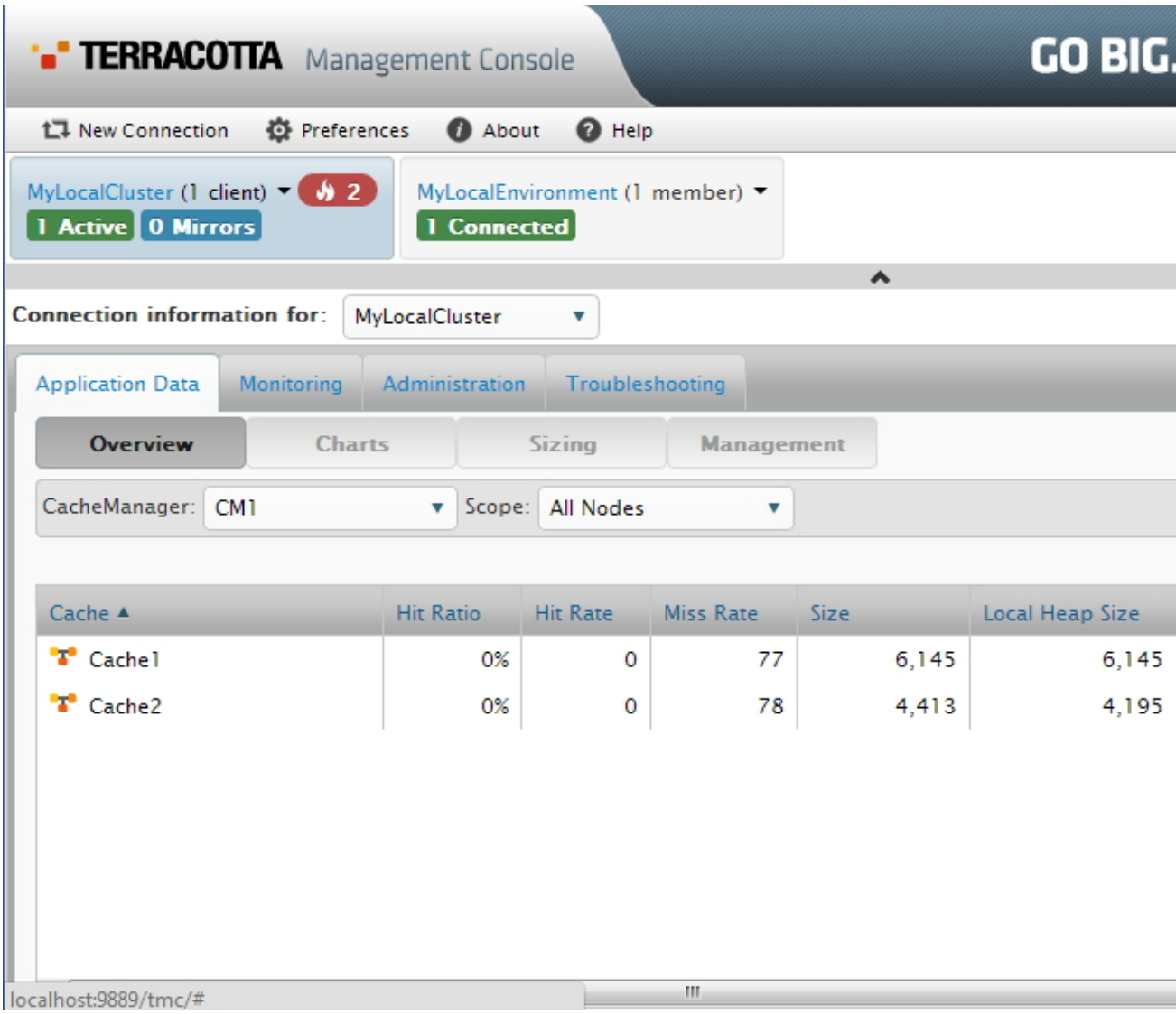
   Note: For Microsoft Windows installations, use the BAT scripts, and where forward slashes ("/") are given in directory paths, substitute back slashes ("\").
3. In a terminal, change to your Terracotta tools/management-console/ directory. Then execute the start-tmc command:

```
%> cd /path/to/bigmemory-max-<version>/tools/management-console
%> ./bin/start-tmc.sh
```

4. In a browser, enter the URL `http://localhost:9889/tmc`. When you first connect to the TMC, the authentication setup page appears, where you can choose to run the TMC with authentication or without.



5. Use the TMC to manage all of the clients and servers in your deployment.



For more information about the TMC, refer to the TMC documentation.

More Information on Configuration

Topic	Description
Terracotta Clustering	BigMemory Max can manage in-memory data in a single, standalone installation or with a Terracotta server. This page provides configuration essentials for distributing BigMemory across a Terracotta cluster or server array.
Storage Tiers	BigMemory Max includes storage options for your in-memory data. This page discusses the storage tier options and shows how to configure them in a standalone installation. Refer to Terracotta Server Array Architecture for distributed configuration information.
Sizing Storage Tiers	Tuning BigMemory Max often involves sizing data storage tiers appropriately. BigMemory Max provides a number of ways to size tiers using simple cache-configuration sizing attributes. This page explains tuning of tier size by configuring dynamic allocation of memory and automatic load balancing.
Expiration,	One of the most important aspects of managing in-memory data involves managing the

Pinning, and Eviction	life of the data in each tier. This page covers managing data life in BigMemory Max and the Terracotta Server Array, including the pinning features of Automatic Resource Control (ARC).
Fast Restartability	This page covers persistence, fast restartability, and using the local disk as a storage tier. The Fast Restart feature provides enterprise-ready crash resilience, which can serve as a fast recovery system after failures, a hot mirror of the data set on the disk at the application node, and an operational store with in-memory speed for reads and writes.

Storage Tiers

Introduction

BigMemory Max has three storage tiers, summarized here:

- Memory store – Heap memory that holds a copy of the hottest subset of data from the off-heap store. Subject to Java GC.
- Off-heap store – Limited in size only by available RAM. Not subject to Java GC. Can store serialized data only. Provides overflow capacity to the memory store.
- Disk store – Backs up in-memory data and provides overflow capacity to the other tiers. Can store serialized data only. **Note**: The disk tier is available for local (standalone) instances of BigMemory Max. For distributed BigMemory Max, a Terracotta server or server array supplies the third tier. (For more information, refer to the Distributed BigMemory Max Configuration Guide.)

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

Before running in production, it is strongly recommended that you test the BigMemory Max tiers with the actual amount of data you expect to use in production. For information about sizing the tiers, refer to Configuration Sizing Attributes.

Memory Store

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

The memory store has the following characteristics:

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

Configuring the Memory Store

The memory store is the top tier and is automatically used by BigMemory Max to store the data hotset because it is the fastest store. It requires no special configuration to enable, and its overall size is taken from the Java heap size. Since it exists in the heap, it is limited by Java GC constraints.

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

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

When an element is added to a cache and it goes beyond its maximum memory size, an existing element is either deleted, if overflow is not enabled, or evaluated for spooling to another tier, if overflow is enabled. The overflow options are `overflowToOffHeap` and `<persistence>` (disk store).

If overflow is enabled, a check for expiry is carried out. If it is expired it is deleted; if not it is spooled. The eviction of an item from the memory store is based on the optional `MemoryStoreEvictionPolicy` attribute specified in the configuration file. Legal values are LRU (default), LFU and FIFO:

- **Least Recently Used (LRU)**—LRU is the default setting. The last-used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.
- **Least Frequently Used (LFU)**—For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element with least number of hits, the Less Frequently Used element, is evicted.
- **First In First Out (FIFO)**—Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

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

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

TIP: Calculating the Size of a Cache

calculateInMemorySize() is a convenient method which can provide an estimate of the size (in bytes) of the memory store. It returns the serialized size of the cache, providing a rough estimate. Do not use this method in production as it is has a negative effect on performance.
An alternative would be to have an expiry thread. This is a trade-off between lower memory use and short locking periods and CPU utilization. The design is in favour of the latter. For those concerned with memory use, simply reduce the tier size. For more information, refer to Sizing Storage Tiers.

Off-Heap Store

The off-heap store extends the in-memory store to memory outside the of the object heap. This store, which is not subject to Java GC, is limited only by the amount of RAM available.

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

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

Allocating Direct Memory in the JVM

The off-heap store uses the direct-memory portion of the JVM. You must allocate sufficient direct memory for the off-heap store by using the JVM property `MaxDirectMemorySize`.

For example, to allocate 2GB of direct memory in the JVM:

```
java -XX:MaxDirectMemorySize=2G ...
```

Since direct memory may be shared with other processes, allocate at least 256MB (or preferably 1GB) more to direct memory than will be allocated to the off-heap store.

Note the following about allocating direct memory:

- If you configure off-heap memory but do not allocate direct memory with -XX:MaxDirectMemorySize, the default value for direct memory depends on your version of your JVM. Oracle HotSpot has a default equal to maximum heap size (-Xmx value), although some early versions may default to a particular value.
- MaxDirectMemorySize must be added to the local node's startup environment.
- Direct memory, which is part of the Java process heap, is separate from the object heap allocated by -Xmx. The value allocated by MaxDirectMemorySize must not exceed physical RAM, and is likely to be less than total available RAM due to other memory requirements.
- The amount of direct memory allocated must be within the constraints of available system memory and configured off-heap memory.
- The maximum amount of direct memory space you can use depends on the process data model (32-bit or 64-bit) and the associated operating system limitations, the amount of virtual memory available on the system, and the amount of physical memory available on the system.

Using Off-Heap Store with 32-bit JVMs

The amount of heap-offload you can achieve is limited by addressable memory. 64-bit systems can allow as much memory as the hardware and operating system can handle, while 32-bit systems have strict limitations on the amount of memory that can be effectively managed.

For a 32-bit process model, the maximum virtual address size of the process is typically 4GB, though most 32-bit operating systems have a 2GB limit. The maximum heap size available to Java is lower still due to particular operating-system (OS) limitations, other operations that may run on the machine (such as mmap operations used by certain APIs), and various JVM requirements for loading shared libraries and other code. A useful rule to observe is to allocate no more to off-heap memory than what is left over after -Xmx is set. For example, if you set -Xmx3G, then off-heap should be no more than 1GB. Breaking this rule may not cause an OutOfMemoryError on startup, but one is likely to occur at some point during the JVM's life.

If Java GC issues are afflicting a 32-bit JVM, then off-heap store can help. However, note the following:

- Everything has to fit in 4GB of addressable space. If 2GB of heap is allocated (with `-Xmx2g`) then at most are are 2GB left for off-heap data.
- The JVM process requires some of the 4GB of addressable space for its code and shared libraries plus any extra Operating System overhead.
- Allocating a 3GB heap with -Xmx, as well as 2047MB of off-heap memory, will not cause an error at startup, but when it's time to grow the heap an OutOfMemoryError is likely.
- If both -Xms3G and -Xmx3G are used with 2047MB of off-heap memory, the virtual machine will start but then complain as soon as the off-heap store tries to allocate the off-heap buffers.
- Some APIs, such as `java.util.zip.ZipFile` on a 1.5 JVM, may <mmap> files in memory. This will also use up process space and may trigger an OutOfMemoryError.

Configuring the Off-Heap Store

If an off-heap store is configured, the corresponding memory store overflows to that off-heap store. Configuring an off-heap store can be done either through XML or programmatically. With either method, the

off-heap store is configured on a per-cache basis.

Declarative Configuration

The following XML configuration creates an off-heap cache with an in-heap store (maxEntriesLocalHeap) of 10,000 elements which overflow to a 1-gigabyte off-heap area.

```
<ehcache updateCheck="false" monitoring="off" dynamicConfig="false">
  <defaultCache maxEntriesLocalHeap="10000"
              eternal="true"
              memoryStoreEvictionPolicy="LRU"
              statistics="false" />

  <cache name="sample-offheap-cache"
       maxEntriesLocalHeap="10000"
       eternal="true"
       memoryStoreEvictionPolicy="LRU"
       overflowToOffHeap="true"
       maxBytesLocalOffHeap="1G"/>
</ehcache>
```

The configuration options are:

overflowToOffHeap

Values may be true or false. When set to true, enables the cache to utilize off-heap memory storage to improve performance. Off-heap memory is not subject to Java GC cycles and has a size limit set by the Java property MaxDirectMemorySize. The default value is false.

maxBytesLocalOffHeap

Sets the amount of off-heap memory available to the cache and is in effect only if `overflowToOffHeap` is tr ue. The minimum amount that can be allocated is 1MB. There is no maximum.

For more information on sizing data stores, refer to this page.

NOTE: Heap Configuration When Using an Off-heap Store

You should set maxEntriesLocalHeap to at least 100 elements when using an off-heap store to avoid performance degradation. Lower values for maxEntriesLocalHeap trigger a warning to be logged.

Programmatic Configuration

The equivalent cache can be created using the following programmatic configuration:

```
public Cache createOffHeapCache()
{
  CacheConfiguration config = new CacheConfiguration("sample-offheap-cache", 10000)
  .overflowToOffHeap(true).maxBytesLocalOffHeap("1G");
  Cache cache = new Cache(config);
  manager.addCache(cache);
  return cache;
}
```

Disk Store

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

Note: The disk tier is available for local (standalone) instances of BigMemory Max. For distributed BigMemory Max, a Terracotta server or server array is used instead of a disk tier. (For more information, refer to the Distributed BigMemory Max Configuration Guide.)

This section covers local disk usage. Additional information may also be found on the Fast Restartability page.

Serialization

Only data that is `Serializable` can be placed in the disk store. Writes to and from the disk use `ObjectInputStream` and the Java serialization mechanism. Any non-serializable data overflowing to the disk store is removed and a NotSerializableException is thrown.

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

- The serialization time for a Java object consisting of a large Map of String arrays was 126ms, where the serialized size was 349,225 bytes.
- The serialization time for a byte[] was 7ms, where the serialized size was 310,232 bytes.

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

Configuring the Disk Store

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

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

Disk Storage Options

Two disk store options are available:

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

See the following sections for more information on these options.

localTempSwap

The "localTempSwap" persistence strategy allows local disk usage during BigMemory operation, providing an extra tier for storage. This disk storage is temporary and is cleared after a restart.

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

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

localRestartable

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

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

Disk Store Configuration Element

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

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

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

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

The following system properties are also legal, in which case they are translated:

- user.home - User's home directory
- user.dir - User's current working directory
- java.io.tmpdir - Default temp file path
- ehcache.disk.store.dir - A system property you would normally specify on the command line—for example, java -Dehcache.disk.store.dir=/u01/myapp/diskdir.

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

```
user.dir/one
```

To programmatically set a disk store path:

```
DiskStoreConfiguration diskStoreConfiguration = new DiskStoreConfiguration();

diskStoreConfiguration.setPath("/my/path/dir");

// Already created a configuration object ...
configuration.addDiskStore(diskStoreConfiguration);

CacheManager mgr = new CacheManager(configuration);
```

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

Disk Store Expiry and Eviction

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

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

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

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

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

Turning off Disk Stores

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

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

Configuration Examples

These examples show how to allocate 8GB of machine memory to different stores. It assumes a data set of 7GB - say for a cache of 7M items (each 1kb in size).

Those who want minimal application response-time variance (or minimizing GC pause times), will likely want all the cache to be off-heap. Assuming that 1GB of heap is needed for the rest of the app, they will set their Java config as follows:

```
java -Xms1G -Xmx1G -XX:maxDirectMemorySize=7G
```

And their Ehcache config as:

```
<cache
 maxEntriesLocalHeap=100
 overflowToOffHeap="true"
 maxBytesLocalOffHeap="6G"
... />
```

NOTE: Direct Memory and Off-heap Memory Allocations

To accommodate server communications layer requirements, the value of maxDirectMemorySize must be greater than the value of maxBytesLocalOffHeap. The exact amount greater depends upon the size of maxBytesLocalOffHeap. The minimum is 256MB, but if you allocate 1GB more to the maxDirectMemorySize, it will certainly be sufficient. The server will only use what it needs and the rest will remain available.

Configuration Examples

Those who want best possible performance for a hot set of data, while still reducing overall application repsonse time variance, will likely want a combination of on-heap and off-heap. The heap will be used for the hot set, the offheap for the rest. So, for example if the hot set is 1M items (or 1GB) of the 7GB data. They will set their Java config as follows

```
java -Xms2G -Xmx2G -XX:maxDirectMemorySize=6G
```

And their Ehcache config as:

```
<cache
  maxEntriesLocalHeap=1M
  overflowToOffHeap="true"
  maxBytesLocalOffHeap="5G"
  ...>
```

This configuration will compare VERY favorably against the alternative of keeping the less-hot set in a database (100x slower) or caching on local disk (20x slower).

Where the data set is small and pauses are not a problem, the whole data set can be kept on heap:

```
<cache
  maxEntriesLocalHeap=1M
  overflowToOffHeap="false"
  ...>
```

Where latency isn't an issue, the disk can be used:

```
<cache
  maxEntriesLocalHeap=1M
  <persistence strategy="localTempSwap"/>
  ...>
```

Sizing Storage Tiers

Introduction

Tuning BigMemory Max often involves sizing the data storage tiers appropriately. BigMemory Max provides a number of ways to size the different data tiers using simple sizing attributes. These sizing attributes affect memory and disk resources.

Sizing Attributes

The following table summarizes the sizing attributes in BigMemory Max's Ehcache API.

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

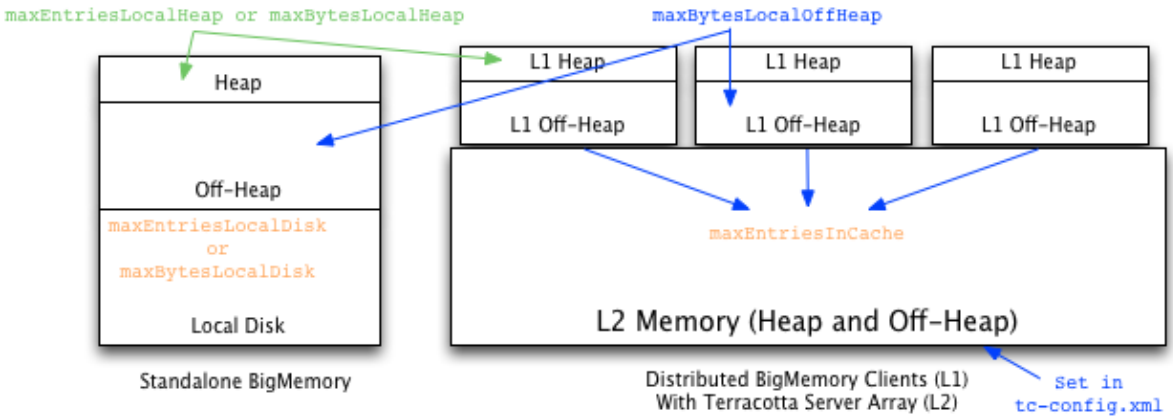
The following table summarizes sizing attributes for distributed BigMemory Max (using a Terracotta server or server array).

Tier	Attribute	Pooling available at CacheManager Level?	Description
Heap	`maxEntriesLocalHeap` `maxBytesLocalHeap`	`maxBytesLocalHeap` only	The maximum number of cache entries or bytes a cache can use in local heap memory, or, when set at the CacheManager level (`maxBytesLocalHeap` only), a local pool available to all caches under that CacheManager. This setting is required for every cache or at the CacheManager

			level.
Off-heap	`maxBytesLocalOffHeap`	Yes	The maximum number of bytes a cache can use in off-heap memory, or, when set at the CacheManager level, as a pool available to all caches under that CacheManager. This setting requires BigMemory.
Local disk	N/A	N/A	Distributed caches cannot use the local disk.
Terracotta Server Array	`maxEntriesInCache`	No	The number of cache elements that the Terracotta Server Array will store for a distributed cache. This value must be greater than or equal to the number of entries in all of the clients of the server array. Set on individual distributed caches only, this setting is unlimited by default.

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

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



Pooling Resources Versus Sizing Individual Data Sets

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

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

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

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

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

Memory Store (Heap)

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

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

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

Off-Heap Store

Off-heap sizing can be configured in bytes only, never by entries.

If a CacheManager has a pool configured for off-heap, your application cannot add caches dynamically that have off-heap configuration — doing so generates an error. In addition, if any caches that used the pool are removed programmatically or through the Terracotta Management Console (TMC), other caches in the pool cannot claim the unused portion. To allot the entire off-heap pool to the remaining caches, remove the unwanted cache from the Ehcache configuration and then reload the configuration.

To use off-heap as a data tier, a cache must have `overflowToOffHeap` set to "true". If a CacheManager has a pool configured for using off-heap, the `overflowToOffHeap` attribute is automatically set to "true" for all caches. In this case, you can prevent a specific cache from overflowing to off-heap by explicitly setting its `overflowToOffHeap` attribute to "false".

Note that an exception is thrown if any cache using an off-heap store attempts to put an element that will cause the off-heap store to exceed its allotted size. The exception will contain a message similar to the following:

```
The element '[ key = 25274, value=[B@3ebb2a91, version=1, hitCount=0,
CreationTime = 1367966069431, LastAccessTime = 1367966069431 ]'
is too large to be stored in this offheap store.
```

Local Disk Store

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

To use the disk as a temporary tier during BigMemory operation, set the `persistenceStrategy` to "localTempSwap" (refer to Temporary Disk Storage), and use the `maxBytesLocalDisk` setting to configure the size of this tier.

To use the disk for data persistence, refer to Data Persistence Implementation.

Note that BigMemory Max distributed across a Terracotta Server Arrary cannot use the local disk. For more information, refer to the Distributed BigMemory Max Configuration Guide.

Sizing Examples

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

Pooled Resources

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

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="100M"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="50G">
...

<cache name="Cache1" ... </cache>
<cache name="Cache2" ... </cache>
<cache name="Cache3" ... </cache>

</ehcache>
```

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

Explicitly Sizing Data Sets

You can explicitly allocate resources to specific caches:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="100M"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="60G">
...

<cache name="Cache1" ...
        maxBytesLocalHeap="50M"
        ...
  </cache>
```

```
<cache name="Cache2" ...
       maxBytesLocalOffHeap="5G"
       ...
  </cache>
<cache name="Cache3" ... </cache>

</ehcache>
```

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

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

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

Mixed Sizing Configurations

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

```
<ehcache xmlns...
        Name="CM2"
        maxBytesLocalHeap="100M">
...

<cache name="Cache4" ...
        maxBytesLocalHeap="50M"
        maxEntriesLocalDisk="100000"
        ...
  </cache>

<cache name="Cache5" ...
       maxBytesLocalOffHeap="10G"
       ...
  </cache>
<cache name="Cache6" ... </cache>

</ehcache>
```

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

Using Percents

The following configuration sets pools for each tier:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="1G"
```

```
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="50G">
...

<!-- Cache1 gets 400Mb of heap, 2.5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache1" ...
maxBytesLocalHeap="40%">
</cache>

<!-- Cache2 gets 300Mb of heap, 5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache2" ...
maxBytesLocalOffHeap="50%">
</cache>

<!-- Cache2 gets 300Mb of heap, 2.5Gb of off-heap, and 40Gb of disk. -->
<cache name="Cache3" ...
maxBytesLocalDisk="80%">
</cache>
</ehcache>
```

NOTE: Configuring Cache Sizes with Percentages

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

Sizing Data Sets Without a Pool

The CacheManager in this example does not pool any resources.

```
<ehcache xmlns...
        Name="CM3"
       ... >
...

<cache name="Cache7" ...
        maxBytesLocalHeap="50M"
        maxEntriesLocalDisk="100000"
        ...
  </cache>

<cache name="Cache8" ...
       maxEntriesLocalHeap="1000"
       maxBytesLocalOffHeap="10G"
       ...
  </cache>
<cache name="Cache9" ...
       maxBytesLocalHeap="50M"
...
</cache>

</ehcache>
```

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

Sizing Distributed Caches

Terracotta distributed caches can be sized as noted above, except that they do not use the local disk and therefore cannot be configured with *LocalDisk sizing attributes. Distributed caches use the storage resources (BigMemory Max and Fast Restart store) available on the Terracotta Server Array. For more information, refer to TSA Fast Restartability.

Cache-configuration sizing attributes behave as local configuration, which means that every node can load its own sizing attributes for the same caches. That is, while some elements and attributes are fixed by the first Ehcache configuration loaded in the cluster, cache-configuration sizing attributes can vary across nodes for the same cache.

For example, a cache may have the following configuration on one node:

```
<cache name="myCache"
     maxEntriesOnHeap="10000"
     maxBytesLocalOffHeap="8g"
     eternal="false"
     timeToIdleSeconds="3600"
     timeToLiveSeconds="1800">
    <persistence strategy="distributed"/>
    <terracotta/>
</cache>
```

The same cache may have the following size configuration on another node:

```
<cache name="myCache"
     maxEntriesOnHeap="10000"
     maxBytesLocalOffHeap="10g"
     eternal="false"
     timeToIdleSeconds="3600"
     timeToLiveSeconds="1800">
    <persistence strategy="distributed"/>
    <terracotta/>
</cache>
```

If the cache exceeds its size constraints on a node, then with this configuration the Terracotta Server Array provides myCache with an unlimited amount of space for spillover and backup. To impose a limit, you must set `maxEntriesInCache` to a positive non-zero value:

```
<cache name="myCache"
     maxEntriesOnHeap="10000"
     maxBytesLocalOffHeap="10g"
     eternal="false"
     timeToIdleSeconds="3600"
     timeToLiveSeconds="1800"
     maxEntriesInCache="1000000">
    <persistence strategy="distributed"/>
    <terracotta/>
</cache>
```

The Terracotta Server Array will now evict myCache entries to stay within the limit set by `maxEntriesInCache`. However, for any particular cache, eviction on the Terracotta Server Array is based on the largest size configured for that cache. In addition, the Terracotta Server Array will *not* evict any cache entries that exist on at least one client node, regardless of the limit imposed by `maxEntriesInCache`.

Note: If `maxEntriesInCache` is not set, the default value 0 is used, which means that the cache is unbounded and will not undergo capacity eviction (but periodic and resource evictions are still allowed). For more information, refer to Eviction.

For more information about TSA data management, refer to Automatic Resource Management.

Sizing the Terracotta Server Array

Since `maxEntriesInCache` is based on entries, you must size the Terracotta Server Array based on the expected average size of an entry. One way to discover this value is by using the Terracotta Management Console (TMC). Set up a test cluster with the expected data set, and connect it to the TMC. Then navigate to Application Data > Sizing Panel, and review the Relative Cache Sizes by Tier section.

Note that the average cache-entry size reported in the TMC is an estimate. For more information, refer to Using the TMC.

Overriding Size Limitations

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

Built-In Sizing Computation and Enforcement

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

Sizing of Data Set Entries

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

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

Ignoring for Size Calculations

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

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

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

```
@IgnoreSizeOf
public class Dog {
```

```
  private Gender gender;
  private String name;
}
```

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

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

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

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

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

```
# That field references a common graph between all cached entries
com.pany.domain.cache.MyCacheEntry.sharedInstance

# This will ignore all instances of that type
com.pany.domain.SharedState

# This ignores a package
com.pany.example
```

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

Configuration for Limiting the Traversed Object Graph

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

Size-Of Limitation at the CacheManager Level

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

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

This element has the following attributes

- `maxDepth` – Controls how many linked objects can be visited before the size-of engine takes any action. This attribute is required.
- `maxDepthExceededBehavior` – Specifies what happens when the max depth is exceeded while sizing an object graph:
    - "continue" – (DEFAULT) Forces the size-of engine to log a warning and continue the sizing operation. If this attribute is not specified, "continue" is the behavior used.

- "abort" – Forces the SizeOf engine to abort the sizing, log a warning, and mark the cache as not correctly tracking memory usage. With this setting, `Ehcache.hasAbortedSizeOf()` returns true.

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

Size-Of Limitation at the Cache level

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

Debugging of Size-Of Related Errors

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

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

Eviction When Using CacheManager-Level Storage

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

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

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

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

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

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

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

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

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

Simplifying:

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

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

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

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

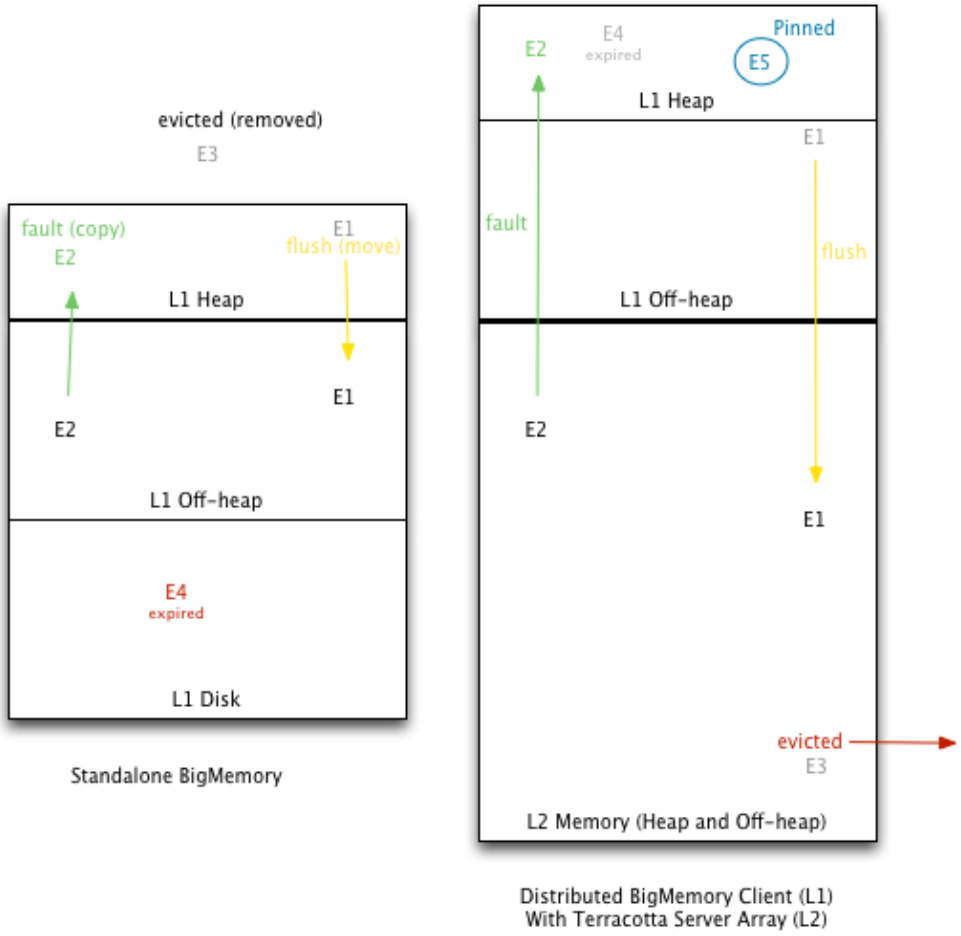
Pinning, Expiration, and Eviction

Introduction

This page covers managing the life of the data in each of BigMemory's data storage tiers, including the pinning features of Automatic Resource Control (ARC).

The following are options for data life within the BigMemory tiers:

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



Setting Expiration

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

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

- `timeToIdleSeconds` – The maximum number of seconds an element can exist in the BigMemory data store without being accessed. The element expires at this limit and will no longer be returned from BigMemory. The default value is 0, which means no TTI eviction takes place (infinite lifetime).
- `timeToLiveSeconds` – The maximum number of seconds an element can exist in the the BigMemory data store regardless of use. The element expires at this limit and will no longer be returned from BigMemory. The default value is 0, which means no TTL eviction takes place (infinite lifetime).
- `maxEntriesInCache` – The maximum sum total number of elements (cache entries) allowed for a distributed cache in all Terracotta clients. If this target is exceeded, eviction occurs to bring the count within the allowed target. The default value is 0, which means that the cache will not undergo capacity eviction (but periodic and resource evictions are still allowed). Note that `maxEntriesInCache` reflects storage allocated on the TSA.
- `eternal` – If the cache's `eternal` flag is set, it overrides any finite TTI/TTL values that have been set. Individual cache elements may also be set to eternal. Eternal elements and caches do not expire, however they may be evicted.

See How Configuration Affects Element Eviction for more information on how configuration can impact eviction.

Pinning Data

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

Data that should remain in memory can be pinned. You cannot pin individual entries, only an entire cache. There are two types of pinning, depending upon whether the pinning configuration should take precedence over resource constraints or the other way around. See the next sections for details.

Configuring Pinning

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

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

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

faulted in from the server, a behavior you can override by setting the Terracotta property `l1.servermapmanager.faultInvalidatedPinnedEntries` to false.

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

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

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

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

Pinning and Cache Sizing

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

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

Use caution with `inCache` pinning

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

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

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

Scope of Pinning

Pinning as a setting exists in the local Ehcache client (L1) memory. It is never distributed in the cluster. In case the pinned cache is bulk-loaded, the pinned data is removed from the L1 and will be faulted back in from the TSA.

Pinning achieved programmatically will not be persisted — after a restart the pinned entries are no longer pinned. Pinning is also lost when an L1 rejoins a cluster. Cache pinning in configuration is reinstated with the configuration file.

Explicitly Removing Data

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

How Configuration Affects Element Flushing and Eviction

The following example shows a cache with certain expiration settings:

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

Note the following about the myCache configuration:

- If a client accesses an entry in myCache that has been idle for more than an hour (`timeToIdleSeconds`), that element is evicted.
- If an entry expires but is not accessed, and no resource constraints force eviction, then the expired entry remains in place until a periodic evictor removes it.
- Entries in myCache can live forever if accessed at least once per 60 minutes (`timeToLiveSeconds`). However, unexpired entries may still be flushed based on other limitations (see Sizing BigMemory Tiers).

Data Freshness and Expiration

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

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

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

Data Freshness and Expiration

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

If you choose the Data Expiration method, you can read more about the cache configuration settings at cache eviction algorithms, and review the timeToIdle and timeToLive configuration settings. The most important consideration when using the expiration method is balancing data freshness with database load. The shorter you make the expiration settings - meaning the more "fresh" you try to make the data - the more load you will incur on the database.

Try out some numbers and see what kind of load your application generates. Even modestly short values such as 5 or 10 minutes will afford significant load reductions.

Fast Restartability

Introduction

BigMemory's Fast Restart feature provides enterprise-ready crash resilience by keeping a fully consistent, real-time record of your in-memory data. After any kind of shutdown — planned or unplanned — the next time your application starts up, all of the data that was in BigMemory is still available and very quickly accessible.

The advantages of the Fast Restart feature include:

- In-memory data survives crashes and enables fast restarts. Because your in-memory data does not need to be reloaded from a remote data source, applications can resume at full speed after a restart.
- A real-time record of your in-memory data provides true fault tolerance. Even with BigMemory, where terabytes of data can be held in memory, the Fast Restart feature provides the equivalent of a local "hot mirror," which guarantees full data consistency.
- A consistent record of your in-memory data opens many possibilities for business innovation, such as arranging data sets according to time-based needs or moving data sets around to different locations. The uses of the Fast Restart store can range from a simple key-value persistence mechanism with fast read performance, to an operational store with in-memory speeds during operation for both reads and writes.

Data Persistence Implementation

The BigMemory Fast Restart feature works by creating a real-time record of the in-memory data, which it persists in a Fast Restart store on the local disk. After any restart, the data that was last in memory (both heap and off-heap stores) automatically loads from the Fast Restart store back into memory.

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

```
<cache>
   <persistence strategy="localRestartable|localTempSwap|none|distributed"
                synchronousWrites="false|true"/>
</cache>
```

Strategy Options

The options for the `strategy` attribute are:

- **"localRestartable"** — Enables the Fast Restart feature which automatically logs all BigMemory data. This option provides fast restartability with fault tolerant data persistence.
- **"localTempSwap"** — Enables temporary local disk usage. This option provides an extra tier for data storage during operation, but this store is not persisted. After a restart, the disk is cleared of any BigMemory data.
- **"none"** — Does not offload data to disk. With this option, all of the working data is kept in memory only. This is the default mode.
- **"distributed"** — Defers to the `<terracotta>` configuration for persistence settings. For more information, refer to Terracotta Clustering Configuration Elements.

Synchronous Writes Options

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

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

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

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

Note: The `synchronousWrites` attribute is also available in the `<terracotta>` sub-element. If configured in both places, it must have the same value.

DiskStore Path

The path to the directory where any required disk files will be created is configured with the `<diskStore>` sub-element of the Ehcache configuration.

- For "localRestartable", a unique and explicitly specified path is required.
- For "localTempSwap", if the DiskStore path is not specified, a default path is used for the disk tier, and the default path will be auto-resolved in the case of a conflict with another CacheManager.

  Note: The Fast Restart feature does not use the disk tier in the same way that conventional disk persistence does. Therefore, when configured for "localRestartable", diskStore size measures such as Cache.getDiskStoreSize() or Cache.calculateOnDiskSize() are not applicable and will return zero. On the other hand, when configured for "localTempSwap", these measures will return size values.

Configuration Examples

This section presents possible disk usage configurations for standalone Ehcache 2.6.

Options for Crash Resilience

The following configuration provides fast restartability with fully consistent data persistence:

```
<ehcache>
  <diskStore path="/path/to/store/data"/>
  <cache>
    <persistence strategy="localRestartable" synchronousWrites="true"/>
```

```
  </cache>
</ehcache>
```

The following configuration provides fast restartability with eventually consistent data persistence:

```
<ehcache>
  <diskStore path="/path/to/store/data"/>
  <cache>
    <persistence strategy="localRestartable" synchronousWrites="false"/>
  </cache>
</ehcache>
```

Clustered Caches

For distributing BigMemory Max across a Terracotta Server Array (TSA), the persistence strategy in the ehcache.xml should be set to "distributed", and the `<terracotta>` sub-element must be present in the configuration.

```
<cache>
  maxEntriesInCache="100000">
  <persistence strategy="distributed"/>
  <terracotta clustered="true" consistency="eventual" synchronousWrites="false"/>
</cache>
```

The attribute `maxEntriesInCache` configures the maximum number of entries in a distributed cache. (`maxEntriesInCache` is not required, but if it is not set, the default is unlimited.)

Note: Restartability must be enabled in the TSA in order for clients to be restartable.

Temporary Disk Storage

The "localTempSwap" persistence strategy create a local disk tier for in-memory data during BigMemory operation. The disk storage is temporary and is cleared after a restart.

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

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

In-memory Only Cache

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

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

Programmatic Configuration Example

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

```
Configuration cacheManagerConfig = new Configuration()
    .diskStore(new DiskStoreConfiguration()
    .path("/path/to/store/data"));
CacheConfiguration cacheConfig = new CacheConfiguration()
    .name("my-cache")
    .maxBytesLocalHeap(16, MemoryUnit.MEGABYTES)
    .maxBytesLocalOffHeap(256, MemoryUnit.MEGABYTES)
    .persistence(new PersistenceConfiguration().strategy(Strategy.LOCALRESTARTABLE));

cacheManagerConfig.addCache(cacheConfig);

CacheManager cacheManager = new CacheManager(cacheManagerConfig);
Ehcache myCache = cacheManager.getEhcache("my-cache");
```

Fast Restart Performance

When configured for fast restartability ("localRestartable" persistence strategy), BigMemory becomes active on restart after all of the in-memory data is loaded. The amount of time until BigMemory is restarted is proportionate to the amount of in-memory data and the speed of the underlying infrastructure. Generally, recovery can occur as fast as the disk speed. With an SSD, for example, if you have a read throughput of 1 GB per second, you will see a similar loading speed during recovery.

Fast Restart Limitations

The following recommendations should be observed when configuring BigMemory for fast restartability:

- The size of on-heap or off-heap stores should not be changed during a shutdown. If the amount of memory allocated is reduced, elements will be evicted upon restart.
- Restartable caches should not be removed from the CacheManager during a shutdown.
- If a restartable cache is disposed, the reference to the cache is deleted, but the cache contents remain in memory and on disk. After a restart, the cache contents are once again recovered into memory and on disk. The way to safely dispose of an unused restartable cache is to call clear on the cache and then dispose, so it doesn't take any space in disk or memory.

Distributed BigMemory Max Configuration Guide

Introduction

The BigMemory Max configuration file (`ehcache.xml` by default) contains the configuration for one instance of a CacheManager (the Ehcache class managing a set of defined caches). This configuration file must be in your application's classpath to be found. When using a WAR file, `ehcache.xml` should be copied to `WEB-INF/classes`.

Note the following about `ehcache.xml` in a Terracotta cluster:

- The copy on disk is loaded into memory from the first Terracotta client (also called application server or node) to join the cluster.
- Once loaded, the configuration is persisted in memory by the Terracotta servers in the cluster and survives client restarts.
- In-memory configuration can be edited in the Terracotta Management Console (TMC). Changes take effect immediately but are *not* written to the original on-disk copy of `ehcache.xml`.
- The in-memory cache configuration is removed with server restarts if the servers are not in persistent mode (`<restartable enabled="false">`), which is the default. The original (on-disk) `ehcache.xml` is loaded.
- The in-memory cache configuration survives server restarts if the servers are in persistent mode (`<restartable enabled="true">`). If you are using the Terracotta servers with persistence of shared data, and you want the cluster to load the original (on-disk) `ehcache.xml`, the servers' database must be wiped by removing the data files from the servers' `server-data` directory. This directory is specified in the Terracotta configuration file in effect (`tc-config.xml` by default). Wiping the database causes *all persisted shared data to be lost*.

A minimal distributed-cache configuration should have the following configured:

- A CacheManager
- A Clustering element in individual cache configurations
- A source for Terracotta client configuration

CacheManager Configuration

CacheManager configuration elements and attributes are fully described in the `ehcache.xml` reference file available in the kit.

Via ehcache.xml

The attributes of `<ehcache>` are:

- name – an optional name for the CacheManager. The name is optional and primarily used for documentation or to distinguish Terracotta clustered cache state. With Terracotta clustered caches, a combination of CacheManager name and cache name uniquely identify a particular cache store in the Terracotta clustered memory. The name will show up in the TMC.

TIP: Naming the CacheManager

Via ehcache.xml

If you employ multiple Ehcache configuration files, use the `name` attribute in the <ehcache> element to identify specific CacheManagers in the cluster. The TMC provides a menu listing these names, allowing you to choose the CacheManager you want to view.

- updateCheck – an optional boolean flag specifying whether this CacheManager should check for new versions of Ehcache over the Internet. If not specified, updateCheck="true".
- monitoring – an optional setting that determines whether the CacheManager should automatically register the SampledCacheMBean with the system MBean server. Currently, this monitoring is only useful when using Terracotta clustering. The "autodetect" value detects the presence of Terracotta clustering and registers the MBean. Other allowed values are "on" and "off". The default is "autodetect".

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

Programmatic Configuration

CacheManagers can be configured programmatically with a fluent API. The example below creates a CacheManager with a Terracotta configuration specified in an URL, and creates a defaultCache and a cache named "example".

```
Configuration configuration = new Configuration()
.terracotta(new TerracottaClientConfiguration().url("localhost:9510"))
.defaultCache(new CacheConfiguration("defaultCache", 100))
.cache(new CacheConfiguration("example", 100)
.timeToIdleSeconds(5)
.timeToLiveSeconds(120)
.terracotta(new TerracottaConfiguration()));
CacheManager manager = new CacheManager(configuration);
```

Terracotta Clustering Configuration Elements

Certain elements in the Ehcache configuration control the clustering of caches with Terracotta.

terracotta

The `<terracotta>` element is an optional sub-element of `<cache>`. It can be set differently for each `<cache>` defined in `ehcache.xml`.

`<terracotta>` has one subelement, `<nonstop>` (see Non-Blocking Disconnected (Nonstop) Operation for more information).

The following `<terracotta>` attribute allows you to control the type of data consistency for the distributed cache:

- consistency – Uses no cache-level locks for better performance ("eventual" DEFAULT) or uses cache-level locks for immediate cache consistency ("strong"). When set to "eventual", allows reads without locks, which means the cache may temporarily return stale data in exchange for substantially improved performance. When set to "strong", guarantees that after any update is completed no local read can return a stale value, but at a potentially high cost to performance, because a large number of

locks may need to be stored in client and server heaps.

Once set, this consistency mode cannot be changed except by reconfiguring the cache using a configuration file and reloading the file. *This setting cannot be changed programmatically.* See Understanding Performance and Cache Consistency for more information.

Except for special cases, the following `<terracotta>` attributes should operate with their default values:

- clustered – Enables ("true" DEFAULT) or disables ("false") Terracotta clustering of a specific cache. Clustering is enabled if this attribute is not specified. Disabling clustering also disables the effects of all of the other attributes.
- localCacheEnabled – Enables ("true" DEFAULT) or disables ("false") local caching of distributed cache data, forcing all of that cached data to reside on the Terracotta Server Array. Disabling local caching may improve performance for distributed caches in write-heavy use cases.
- synchronousWrites – Enables ("true") or disables ("false" DEFAULT) synchronous writes from Terracotta clients (application servers) to Terracotta servers. Asynchronous writes (synchronousWrites="false") maximize performance by allowing clients to proceed without waiting for a "transaction received" acknowledgement from the server. This acknowledgement is unnecessary in most use cases. Synchronous writes (synchronousWrites="true") provide extreme data safety but at a performance cost by requiring that a client receive server acknowledgement of a transaction before that client can proceed. If the cache's consistency mode is eventual, or while it is set to bulk load using the bulk-load API, only asynchronous writes can occur (synchronousWrites="true" is ignored).
- concurrency – Sets the number of segments for the map backing the underlying server store managed by the Terracotta Server Array. If concurrency is not explicitly set (or set to "0"), the system selects an optimized value. See Tuning Concurrency for more information on how to tune this value.

The following attributes are used with Hibernate:

- localKeyCache – Enables ("true") or disables ("false" DEFAULT) a local key cache. BigMemory Max can cache a "hotset" of keys on clients to add locality-of-reference, a feature suitable for read-only cases. Note that the set of keys must be small enough for available memory.
- localKeyCacheSize – Defines the size of the local key cache in number (positive integer) of elements. In effect if localKeyCache is enabled. The default value, 300000, should be tuned to meet application requirements and environmental limitations.
- orphanEviction – Enables ("true" DEFAULT) or disables ("false") orphan eviction. *Orphans* are cache elements that are not resident in any Hibernate second-level cache but still present on the cluster's Terracotta server instances.
- orphanEvictionPeriod – The number of local eviction cycles (that occur on Hibernate) that must be completed before an orphan eviction can take place. The default number of cycles is 4. Raise this value for less aggressive orphan eviction that can reduce faulting on the Terracotta server, or lower it if garbage on the Terracotta server is a concern.

Default Behavior

By default, adding `<terracotta/>` to a `<cache>` block is equivalent to adding the following:

```
<cache name="sampleTerracottaCache"
    maxEntriesLocalHeap="1000"
    eternal="false"
    timeToIdleSeconds="3600"
    timeToLiveSeconds="1800"">
  <persistence strategy="distributed"/>
```

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

terracottaConfig

The `<terracottaConfig>` element enables the distributed-cache client to identify a source of Terracotta configuration. It also allows a client to rejoin a cluster after disconnecting from that cluster and being timed out by a Terracotta server. For more information on the rejoin feature, see Using Rejoin to Automatically Reconnect Terracotta Clients.

The client must load the configuration from a file or a Terracotta server. The value of the `url` attribute should contain a path to the file, a system property, or the address and TSA port (9510 by default) of a server.

TIP: Terracotta Clients and Servers

In a Terracotta cluster, the application server is also known as the client.
For more information on client configuration, see the *Clients Configuration Section* in the Terracotta Configuration Reference.

Adding an URL Attribute

Add the `url` attribute to the `<terracottaConfig>` element as follows:

```
<terracottaConfig url="<source>" />
```

where `<source>` must be one of the following:

- A path (for example, `url="/path/to/tc-config.xml"`)
- An URL (for example, `url="http://www.mydomain.com/path/to/tc-config.xml`)
- A system property (for example, `url="${terracotta.config.location}"`), where the system property is defined like this:

```
System.setProperty("terracotta.config.location","10.x.x.x:9510"");
```

- A Terracotta host address in the form `<host>:<tsa-port>` (for example, `url="host1:9510"`). Note the following about using server addresses in the form `<host>:<tsa-port>`:
  - The default TSA port is 9510.
  - In a multi-server cluster, you can specify a comma-delimited list (for example, `url="host1:9510,host2:9510,host3:9510"`).
  - If the Terracotta configuration source changes at a later time, it must be updated in configuration.

Embedding Terracotta Configuration

You can embed the contents of a Terracotta configuration file in `ehcache.xml` as follows:

```
<terracottaConfig>
   <tc-config>
       <servers>
           <server host="server1" name="s1"/>
           <server host="server2" name="s2"/>
```

```
        </servers>
        <clients>
            <logs>app/logs-%i</logs>
        </clients>
    </tc-config>
</terracottaConfig>
```

Controlling Cache Size

Certain Ehcache cache configuration attributes affect caches clustered with Terracotta.

See How Configuration Affects Element Eviction for more information on how configuration affects eviction.

To learn about eviction and controlling the size of the cache, see the data life and sizing storage tiers pages.

Setting Cache Eviction

Cache eviction removes elements from the cache based on parameters with configurable values. Having an optimal eviction configuration is critical to maintaining cache performance.

To learn about eviction and controlling the size of the cache, see the data life and sizing storage tiers pages.

Ensure that the edited `ehcache.xml` is in your application's classpath. If you are using a WAR file, `ehcache.xml` should be in `WEB-INF/classes`.

See How Configuration Affects Element Eviction for more information on how configuration can impact eviction. See Terracotta Clustering Configuration Elements for definitions of other available configuration properties.

Cache-Configuration File Properties

See Terracotta Clustering Configuration Elements for more information.

Cache Events Configuration

The `<cache>` subelement <cacheEventListenerFactory>, which registers listeners for cache events such as puts and updates, has a notification scope controlled by the attribute `listenFor`. This attribute can have one of the following values:

- local – Listen for events on the local node. No remote events are detected.
- remote – Listen for events on other nodes. No local events are detected.
- all – (DEFAULT) Listen for events on both the local node and on remote nodes.

In order for cache events to be detected by remote nodes in a Terracotta cluster, event listeners must have a scope that includes remote events. For example, the following configuration allows listeners of type MyCacheListener to detect both local and remote events:

```
<cache name="myCache" ... >
 <!-- Not defining the listenFor attribute for <cacheEventListenerFactory> is by default
      equivalent to listenFor="all". -->
```

```
 <cacheEventListenerFactory
    class="net.sf.ehcache.event.TerracottaCacheEventReplicationFactory" />
 <terracotta />
</cache>
```

You must use `net.sf.ehcache.event.TerracottaCacheEventReplicationFactory` as the factory class to enable cluster-wide cache-event broadcasts in a Terracotta cluster.

See Cache Events in a Terracotta Cluster for more information on cache events in a Terracotta cluster.

Copy On Read

The `copyOnRead` setting is most easily explained by first examining what it does when not enabled and exploring the potential problems that can arise. For a cache in which `copyOnRead` is NOT enabled, the following reference comparison will always be true:

```
Object obj1 = c.get("key").getValue();
// Assume no other thread changes the cache mapping between these get() operations ....
Object obj2 = c.get("key").getValue();
if (obj1 == obj2) {
 System.err.println("Same value objects!");
}
```

The fact that the same object reference is returned across multiple `get()` operations implies that the cache is storing a direct reference to cache value. This default behavior (copyOnRead=false) is usually desired, although there are at least two scenarios in which it is problematic:

(1) Caches shared between classloaders

and

(2) Mutable value objects

Imagine two web applications that both have access to the same Cache instance (this implies that the core Ehcache classes are in a common classloader). Imagine further that the classes for value types in the cache are duplicated in the web application (so they are not present in the common loader). In this scenario you would get ClassCastExceptions when one web application accessed a value previously read by the other application.

One obvious solution to this problem is to move the value types to the common loader, but another is to enable `copyOnRead`. When copyOnRead is in effect, the object references are unique with every `get()`. Having unique object references means that the thread context loader of the caller will be used to materialize the cache values on each `get()`. This feature has utility in OSGi environments as well where a common cache service might be shared between bundles.

Another subtle issue concerns mutable value objects in a distributed cache. Consider this simple code with a Cache containing a mutable value type (Foo):

```
class Foo {
 int field;
}
Foo foo = (Foo) c.get("key").getValue();
foo.field++;
// foo instance is never re-put() to the cache
```

```
// ...
```

If the Foo instance is never put back into the Cache your local cache is no longer consistent with the cluster (it is locally modified only). Enabling `copyOnRead` eliminates this possibility since the only way to affect cache values is to call mutator methods on the Cache.

It is worth noting that there is a performance penalty to copyOnRead since values are deserialized on every `get()`.

Configuring Robust Distributed In-memory Data Sets

Making the BigMemory Max in-memory data system robust is typically a combination of Ehcache configuration and Terracotta configuration and architecture. For more information, see the following documentation:

- Nonstop caches – Configure caches to take a specified action after an Ehcache node appears to be disconnected from the cluster.
- Rejoin the cluster – Allow Ehcache nodes to rejoin the cluster as new clients after being disconnected from the cluster.
- High Availability in a Terracotta cluster – Configure nodes to ride out network interruptions and long Java GC cycles, connect to a backup Terracotta server, and more.
- Architecture – Design a cluster that provides failover.

Incompatible Configuration

For any clustered cache, you must delete, disable, or edit configuration elements in `ehcache.xml` that are incompatible when clustering with Terracotta. Clustered caches have a `<terracotta/>` or `<terracotta clustered="true"/>` element.

The following Ehcache configuration attributes or elements should be deleted or disabled:

- Replication-related attributes such as `replicateAsynchronously` and `replicatePuts`.
- The attribute `MemoryStoreEvictionPolicy` is ignored (a clock eviction policy is used instead), however, if allowed to remain in a clustered cache configuration, the `MemoryStoreEvictionPolicy` may cause an exception.

Exporting Configuration from the Terracotta Management Console

To create or edit a cache configuration in a live cluster, see Editing Cache Configuration.

To persist custom cache configuration values, create a cache configuration file by exporting customized configuration from the TMC or create a file that conforms to the required format. This file must take the place of any configuration file used when the cluster was last started.

Nonstop (Non-Blocking) Operation

Introduction

The nonstop feature allows certain operations to proceed on Terracotta clients that have become disconnected from the cluster, or if an operation cannot complete by the nonstop timeout value. This is useful in meeting service-level agreement (SLA) requirements, responding to node failures, or building a more robust High Availability cluster.

One way BigMemory Max can go into nonstop mode is when a client receives a "cluster offline" event. Note that a nonstop instance can go into nonstop mode even if the client is not disconnected, such as when an operation is unable to complete within the timeout allotted by the nonstop configuration. In addition, nonstop instances running in a client that is unable to locate the Terracotta Server Array (TSA) at startup will initiate nonstop behavior as if the client had disconnected.

Nonstop can be used in conjunction with rejoin.

Use cases include:

- Setting timeouts on operations.

  For example, say you use BigMemory Max rather than a mainframe. The SLA calls for 3 seconds. There is a temporary network interruption that delays the response to a cache request. With the timeout you can return after 3 seconds. The lookup is then done against the mainframe. This could also be useful for write-through, writes to disk, or synchronous writes.
- Automatically responding to cluster topology events to take a pre-configured action.
- Allowing Availability over Consistency within the CAP theorem when a network partition occurs.
- Providing graceful degradation to user applications when distributed BigMemory Max becomes unavailable.

Configuring Nonstop Operation

Nonstop is configured in a `<cache>` block under the `<terracotta>` subelement. In the following example, myCache has nonstop configuration:

```
<cache name="myCache" maxEntriesLocalHeap="10000" eternal="false">
 <terracotta>
   <nonstop immediateTimeout="false" timeoutMillis="30000">
     <timeoutBehavior type="noop" />
   </nonstop>
 </terracotta>
</cache>
```

Nonstop is enabled by default or if `<nonstop>` appears in a cache's `<terracotta>` block.

Nonstop Timeouts and Behaviors

Nonstop caches can be configured with the following attributes:

- `enabled` – Enables ("true" DEFAULT) or disables ("false") the ability of a cache to execute certain actions after a Terracotta client disconnects. This attribute is optional for enabling nonstop.
- `immediateTimeout` – Enables ("true") or disables ("false" DEFAULT) an immediate timeout response if the Terracotta client detects a network interruption (the node is disconnected from the cluster). If enabled, this parameter overrides `timeoutMillis`, so that the option set in `timeoutBehavior` is in effect immediately.
- `timeoutMillis` – Specifies the number of milliseconds an application waits for any cache operation to return before timing out. The default value is 30000 (thirty seconds). The behavior after the timeout occurs is determined by `timeoutBehavior`.
- `searchTimeoutMillis` – Specifies the number of milliseconds an application waits for search operations to return before timing out. The default value is 30000 (thirty seconds). The behavior after the timeout occurs is determined by `timeoutBehavior`.

`<nonstop>` has one self-closing subelement, <timeoutBehavior>. This subelement determines the response after a timeout occurs (`timeoutMillis` expires or an immediate timeout occurs). The response can be set by the <timeoutBehavior> attribute `type`. This attribute can have one of the values listed in the following table:

Value	Behavior
`exception`	(DEFAULT) Throw NonStopCacheException. See When is NonStopCacheException Thrown? for more information on this exception.
`noop`	Return null for gets. Ignore all other cache operations. Hibernate users may want to use this option to allow their application to continue with an alternative data source.
`localReads`	For caches with Terracotta clustering, allow inconsistent reads of cache data. Ignore all other cache operations. For caches without Terracotta clustering, throw an exception.
`localReadsAndExceptionOnWrite`	For caches with Terracotta clustering, allow inconsistent reads of cache data, and throw a NonStopCacheException for write operations. For caches without Terracotta clustering, throw an exception.

Tuning Nonstop Timeouts and Behaviors

You can tune the default timeout values and behaviors of nonstop caches to fit your environment.

Network Interruptions

For example, in an environment with regular network interruptions, consider disabling `immediateTimeout` and increasing `timeoutMillis` to prevent timeouts for most of the interruptions.

For a cluster that experiences regular but short network interruptions, and in which caches clustered with Terracotta carry read-mostly data or there is tolerance of potentially stale data, you may want to set `timeoutBehavior` to `localReads`.

Slow Cache Operations

In an environment where cache operations can be slow to return and data is required to always be in sync, increase `timeoutMillis` to prevent frequent timeouts. Set `timeoutBehavior` to `noop` to force the

application to get data from another source or `exception` if the application should stop.

For example, a `cache.acquireWriteLockOnKey(key)` operation may exceed the nonstop timeout while waiting for a lock. This would trigger nonstop mode only because the lock couldn't be acquired in time. Using `cache.tryWriteLockOnKey(key, timeout)`, with the method's timeout set to less than the nonstop timeout, avoids this problem.

Bulk Loading

If a nonstop cache is bulk-loaded using the Bulk-Load API, a multiplier is applied to the configured nonstop timeout whenever the method `net.sf.ehcache.Ehcache.setNodeBulkLoadEnabled(boolean)` is used. The default value of the multiplier is 10. You can tune the multiplier using the `bulkOpsTimeoutMultiplyFactor` system property:

```
-Dnet.sf.ehcache.nonstop.bulkOpsTimeoutMultiplyFactor=10
```

Note that when nonstop is enabled, the cache size displayed in the TMC is subject to the `bulkOpsTimeoutMultiplyFactor`. Increasing this multiplier on the clients can facilitate more accurate size reporting.

This multiplier also affects the methods `net.sf.ehcache.Ehcache.getAll()`, `net.sf.ehcache.Ehcache.removeAll()`, and `net.sf.ehcache.Ehcache.removeAll(boolean)`.

Printing Stack Traces on Exceptions

You can turn on additional logging (at the INFO level) for nonstop timeouts using:

```
-Dnet.sf.ehcache.nonstop.printStackTraceOnException=true
```

This is a dynamic property that can also be managed programmatically:

```
System.setProperty(PRINT_STACK_TRACE_ON_EXCEPTION_PROPERTY, "true")
```

This property logs the stack trace for the nonstop thread. Note that, with this property set, log files can become very large in environments in which a large number of timeouts occur.

Nonstop Exceptions

Typically, application code may access the cache frequently and at various points. Therefore, with a nonstop cache, where your application could encounter NonStopCacheExceptions is difficult to predict. The following sections provide guidance on when to expect NonStopCacheExceptions and how to handle them.

When is NonStopCacheException Thrown?

NonStopCacheException is usually thrown when it is the configured behavior for a nonstop cache in a client that disconnects from the cluster. In the following example, the exception would be thrown 30 seconds after the disconnection (or the "cluster offline" event is received):

```
<nonstop immediateTimeout="false" timeoutMillis="30000">
```

```
<timeoutBehavior type="exception" />
</nonstop>
```

However, under certain circumstances the NonStopCache exception can be thrown even if a nonstop cache's timeout behavior is *not* set to throw the exception. This can happen when the cache goes into nonstop mode during an attempt to acquire or release a lock. These lock operations are associated with certain lock APIs and special cache types such as Explicit Locking, BlockingCache, SelfPopulatingCache, and UpdatingSelfPopulatingCache.

A NonStopCacheException can also be thrown if the cache must fault in an element to satisfy a `get()` operation. If the Terracotta Server Array cannot respond within the configured nonstop timeout, the exception is thrown.

A related exception, InvalidLockAfterRejoinException, can be thrown during or after client rejoin (see Using Rejoin to Automatically Reconnect Terracotta Clients). This exception occurs when an unlock operation takes place on a lock obtained *before* the rejoin attempt completed.

TIP: Use try-finally Blocks

To ensure that locks are released properly, application code using Ehcache lock APIs should encapsulate lock-unlock operations with try-finally blocks:

```
myLock.acquireLock();
try {
  // Do some work.
} finally {
  myLock.unlock();
}
```

Handling Nonstop Exceptions

Your application can handle nonstop exceptions in the same way it handles other exceptions. For nonstop caches, an unhandled-exceptions handler could, for example, refer to a separate thread any cleanup needed to manage the problem effectively.

Another way to handle nonstop exceptions is by using a dedicated Ehcache decorator that manages the exception outside of the application framework. The following is an example of how the decorator might operate:

```
try { cache.put(element); }

catch(NonStopCacheException e) {

  handler.handleException(cache, element, e);
}
```

Default Settings for Terracotta Distributed BigMemory

Introduction

A number of properties control the way the Terracotta Server Array and its clients perform in a Terracotta cluster. Some of these settings are found in the Terracotta configuration file (`tc-config.xml`), while others are found in the BigMemory configuration file (`ehcache.xml`). A few must be set programmatically.

The following sections detail the most important of these properties and shows their default values. To confirm the latest default values for your version of Terracotta software, see the XSD included with your Terracotta kit.

Terracotta Server Array

A Terracotta cluster is composed of clients and servers. Terracotta properties often use a shorthand notation where a client is referred to as "l1" and a server as "l2".

These properties are set at the top of `tc-config.xml` using a configuration block similar to the following:

```
<tc-properties>
   <property name="l2.nha.tcgroupcomm.reconnect.enabled" value="true" />
<!-- More properties here. -->
</tc-properties>
```

See the Terracotta Server Array documentation for more information on the Terracotta Server Array.

Reconnection and Logging Properties

The following reconnection properties are shown with default values. These properties can be set to custom values using Terracotta configuration properties (`<tc-properties>`/`<property>` elements in `tc-config.xml`).

Property	Default Value	Notes
l2.nha.tcgroupcomm.reconnect.enabled	true	Enables server-to-server reconnections.
l2.nha.tcgroupcomm.reconnect.timeout	5000ms	l2-l2 reconnection timeout.
l2.l1reconnect.enabled	true	Enables an l1 to reconnect to servers.
l2.l1reconnect.timeout.millis	5000ms	The reconnection time out, after which an l1 disconnects.
l1.max.connect.retries	-1	The number of allowed reconnection tries from an l1 to an l2. Affects both initial and subsequent reconnection attempts. -1 allows infinite attempts.
tc.config.getFromSource.timeout	30000ms	Timeout for getting configuration from a source. For example, this controls how long a client can try to access configuration from a server. If the client fails to do so, it will fail to connect to the cluster.
logging.maxBackups	20	

		Upper limit for number of backups of Terracotta log files.
logging.maxLogFileSize	512MB	Maximum size of Terracotta log files before rolling logging starts.

HealthChecker Tolerances

The following properties control disconnection tolerances between Terracotta servers (l2 l2), Terracotta servers and Terracotta clients (l2 -> l1), and Terracotta clients and Terracotta servers (l1 -> l2).

l2l2 GC tolerance : 40 secs, cable pull/network down tolerance : 10secs

```
l2.healthcheck.l2.ping.enabled = true
l2.healthcheck.l2.ping.idletime = 5000
l2.healthcheck.l2.ping.interval = 1000
l2.healthcheck.l2.ping.probes = 3
l2.healthcheck.l2.socketConnect = true
l2.healthcheck.l2.socketConnectTimeout = 5
l2.healthcheck.l2.socketConnectCount = 10
```

l2->l1 GC tolerance : 40 secs, cable pull/network down tolerance : 10secs

```
l2.healthcheck.l1.ping.enabled = true
l2.healthcheck.l1.ping.idletime = 5000
l2.healthcheck.l1.ping.interval = 1000
l2.healthcheck.l1.ping.probes = 3
l2.healthcheck.l1.socketConnect = true
l2.healthcheck.l1.socketConnectTimeout = 5
l2.healthcheck.l1.socketConnectCount = 10
```

l1->l1 GC tolerance : 50 secs, cable pull/network down tolerance : 10secs

```
l1.healthcheck.l2.ping.enabled = true
l1.healthcheck.l2.ping.idletime = 5000
l1.healthcheck.l2.ping.interval = 1000
l1.healthcheck.l2.ping.probes = 3
l1.healthcheck.l2.socketConnect = true
l1.healthcheck.l2.socketConnectTimeout = 5
l1.healthcheck.l2.socketConnectCount = 13
```

Terracotta Clients

Client configuration properties typically address the behavior, size, and functionality of in-memory data stores. Others affect certain types of cache-related bulk operations.

Properties are set in `ehcache.xml` except as noted.

General Settings

The following default settings affect in-memory data. For more information on these settings, see the BigMemory Max documentation.

Property	Default Value	Notes
value mode	SERIALIZATION	

consistency	EVENTUAL	
XA	false	
orphan eviction	true	
local key cache	false	
synchronous writes	false	
ttl	0	0 means never expire.
tti	0	0 means never expire.
transactional mode	off	
persistence strategy	none	
maxEntriesInCache	0	0 means that the cache will not undergo capacity eviction (but periodic and resource evictions are still allowed)
maxBytesLocalHeap	0	
maxBytesLocalOffHeap	0	
maxEntriesLocalHeap	0	0 means infinite.

NonStop Cache

The following default settings affect the behavior of the cache when while the client is disconnected from the cluster. For more information on these settings, see the nonstop-cache documentation.

Property	Default Value	Notes
enable	false	
timeout behavior	exception	
timeout	30000ms	
`net.sf.ehcache.nonstop.bulkOpsTimeoutMultiplyFactor`	10	This value is a timeout multiplication factor affecting bulk operations such as `removeAll()` and `getAll()`. Since the default nonstop timeout is 30 seconds, it sets a timeout of 300 seconds for those operations. The default can be changed programmatically: `cache.getTerracottaConfiguration().getNonstopConfiguration().setBulkOpsTimeoutMultiplyFactor(10)`

Bulk Operations

The following properties are shown with default values. These properties can be set to custom values using Terracotta configuration properties.

Increasing batch sizes may improve throughput, but could raise latency due to the load on resources from processing larger blocks of data.

Property	Default Value	Notes
ehcache.bulkOps.maxKBSize	1MB	Batch size for bulk operations such as putAll and removeAll.
ehcache.getAll.batchSize	1000	

		The number of elements per batch in a getAll operation.
ehcache.incoherent.putsBatchByteSize	5MB	For bulk-loading mode. The minimum size of a batch in a bulk-load operation. Increasing batch sizes may improve throughput, but could raise latency due to the load on resources from processing larger blocks of data.
ehcache.incoherent.putsBatchTimeInMillis	600 ms	For bulk-loading mode. The maximum time the bulk-load operation takes to batch puts before flushing to the Terracotta Server Array.

BigMemory Max Configuration Reference

BigMemory Max uses the standard Ehcache configuration file to set clustering and consistency behavior, optimize cached data, support for JTA and OSGi, and more.

Dynamically Changing Cache Configuration

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

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

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

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

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

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

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

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

Dynamic Configuration Changes for Distributed BigMemory Max

Just as for standalone BigMemory, mutating the configuration of distributed BigMemory requires access to the set methods of `cache.getCacheConfiguration()`.

Dynamic Configuration Changes for Distributed BigMemory Max

The following table provides information for dynamically changing common configuration options in a Terracotta cluster. The table's Scope column, which specifies where the configuration is in effect, can have one of the following values:

- Client – The Terracotta client where the CacheManager runs.
- TSA – The Terracotta Server Array for the cluster.
- BOTH – Both the client and the TSA.

Note that configuration options whose scope covers "BOTH" are distributed and therefore affect a cache on all clients.

Configuration Option	Dynamic	Scope	Notes
Cache name	NO	TSA	
Nonstop	NO	Client	Enable High Availability
Timeout	YES	Client	For nonstop.
Timeout Behavior	YES	Client	For nonstop.
Immediate Timeout When Disconnected	YES	Client	For nonstop.
Time to Idle	YES	BOTH	
Time to Live	YES	BOTH	
Maximum Entries or Bytes in Local Stores	YES	Client	This and certain other sizing attributes may be pooled by the CacheManager, creating limitations on how they can be changed.
Maximum Entries in Cache	YES	TSA	
Persistence Strategy	N/A	N/A	
Disk Expiry Thread Interval	N/A	N/A	
Disk Spool Buffer Size	N/A	N/A	
Maximum Off-heap	N/A	N/A	Maximum off-heap memory allotted to the TSA.
Eternal	YES	BOTH	
Pinning	NO	BOTH	See Pinning, Expiration, and Eviction.
Clear on Flush	NO	Client	
Copy on Read	NO	Client	
Copy on Write	NO	Client	
Statistics	YES	Client	Cache statistics. Change dynamically with `cache.setStatistics(boolean)` method.
Logging	NO	Client	Ehcache and Terracotta logging is specified in configuration. However, cluster events can be set dynamically.
Consistency	NO	Client	It is possible to switch to and from bulk mode.
Synchronous Writes	NO	Client	

To apply non-dynamic L1 changes, remove the existing cache and then add (to the same CacheManager) a new cache with the same name as the removed cache, and which has the new configuration. Restarting the CacheManager with an updated configuration, where all cache names are the same as in the previous configuration, will also apply non-dynamic L1 changes.

Passing Copies Instead of References

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

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

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

The default configuration is "false" for both options.

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

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

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

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

```
CacheConfiguration cacheConfiguration = new CacheConfiguration("copyCache", 10);

CopyStrategyConfiguration copyStrategyConfiguration = new CopyStrategyConfiguration();
copyStrategyConfiguration.setClass("com.company.ehcache.MyCopyStrategy");

cacheConfiguration.addCopyStrategy(copyStrategyConfiguration);
```

Special System Properties

net.sf.ehcache.disabled

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

net.sf.ehcache.use.classic.lru

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

Non-Blocking Disconnected (Nonstop) Cache

A nonstop cache allows certain cache operations to proceed on clients that have become disconnected from the cluster or if a cache operation cannot complete by the nonstop timeout value. One way clients go into nonstop mode is when they receive a "cluster offline" event. Note that a nonstop cache can go into nonstop mode even if the node is not disconnected, such as when a cache operation is unable to complete within the timeout allotted by the nonstop configuration.

Configuring Nonstop

Nonstop is configured in a `<cache>` block under the `<terracotta>` subelement. In the following example, myCache has nonstop configuration:

```
<cache name="myCache" maxEntriesLocalHeap="10000" eternal="false">
   <persistence strategy="distributed"/>
   <terracotta>
     <nonstop immediateTimeout="false" timeoutMillis="30000">
       <timeoutBehavior type="noop" />
     </nonstop>
   </terracotta>
</cache>
```

Nonstop is enabled by default or if `<nonstop>` appears in a cache's `<terracotta>` block.

Nonstop Timeouts and Behaviors

Nonstop caches can be configured with the following attributes:

- `enabled` – Enables ("true" DEFAULT) or disables ("false") the ability of a cache to execute certain actions after a Terracotta client disconnects. This attribute is optional for enabling nonstop.
- `immediateTimeout` – Enables ("true") or disables ("false" DEFAULT) an immediate timeout response if the Terracotta client detects a network interruption (the node is disconnected from the cluster). If enabled, the first request made by a client can take up to the time specified by `timeoutMillis` and subsequent requests timeout immediately.
- `timeoutMillis` – Specifies the number of milliseconds an application waits for any cache operation to return before timing out. The default value is 30000 (thirty seconds). The behavior after the timeout occurs is determined by `timeoutBehavior`.

`<nonstop>` has one self-closing subelement, <timeoutBehavior>. This subelement determines the response after a timeout occurs (`timeoutMillis` expires or an immediate timeout occurs). The response can be set by the <timeoutBehavior> attribute `type`. This attribute can have one of the values listed in the following table:

Value	Behavior

`exception`	(DEFAULT) Throw `NonStopCacheException`. See When is NonStopCacheException Thrown? for more information on this exception.
`noop`	Return null for gets. Ignore all other cache operations. Hibernate users may want to use this option to allow their application to continue with an alternative data source.
`localReads`	For caches with Terracotta clustering, allow inconsistent reads of cache data. Ignore all other cache operations. For caches without Terracotta clustering, throw an exception.

Tuning Nonstop Timeouts and Behaviors

You can tune the default timeout values and behaviors of nonstop caches to fit your environment.

Network Interruptions

For example, in an environment with regular network interruptions, consider disabling `immediateTimeout` and increasing `timeoutMillis` to prevent timeouts for most of the interruptions.

For a cluster that experiences regular but short network interruptions, and in which caches clustered with Terracotta carry read-mostly data or there is tolerance of potentially stale data, you may want to set `timeoutBehavior` to `localReads`.

Slow Cache Operations

In an environment where cache operations can be slow to return and data is required to always be in sync, increase `timeoutMillis` to prevent frequent timeouts. Set `timeoutBehavior` to `noop` to force the application to get data from another source or `exception` if the application should stop.

For example, a `cache.acquireWriteLockOnKey(key)` operation may exceed the nonstop timeout while waiting for a lock. This would trigger nonstop mode only because the lock couldn't be acquired in time. Using `cache.tryWriteLockOnKey(key, timeout)`, with the method's timeout set to less than the nonstop timeout, avoids this problem.

Bulk Loading

If a nonstop cache is bulk-loaded using the Bulk-Load API, a multiplier is applied to the configured nonstop timeout whenever the method `net.sf.ehcache.Ehcache.setNodeBulkLoadEnabled(boolean)` is used. The default value of the multiplier is 10. You can tune the multiplier using the `bulkOpsTimeoutMultiplyFactor` system property:

```
-Dnet.sf.ehcache.nonstop.bulkOpsTimeoutMultiplyFactor=10
```

Note that when nonstop is enabled, the cache size displayed in the TMC is subject to the `bulkOpsTimeoutMultiplyFactor`. Increasing this multiplier on the clients can facilitate more accurate size reporting.

This multiplier also affects the methods `net.sf.ehcache.Ehcache.getAll()`, `net.sf.ehcache.Ehcache.removeAll()`, and `net.sf.ehcache.Ehcache.removeAll(boolean)`.

When is NonStopCacheException Thrown?

NonStopCacheException is usually thrown when it is the configured behavior for a nonstop cache in a client that disconnects from the cluster. In the following example, the exception would be thrown 30 seconds after the disconnection (or the "cluster offline" event is received):

```
<nonstop immediateTimeout="false" timeoutMillis="30000">
<timeoutBehavior type="exception" />
</nonstop>
```

However, under certain circumstances the NonStopCache exception can be thrown even if a nonstop cache's timeout behavior is *not* set to throw the exception. This can happen when the cache goes into nonstop mode during an attempt to acquire or release a lock. These lock operations are associated with certain lock APIs and special cache types such as Explicit Locking, BlockingCache, SelfPopulatingCache, and UpdatingSelfPopulatingCache.

A NonStopCacheException can also be thrown if the cache must fault in an element to satisfy a `get()` operation. If the Terracotta Server Array cannot respond within the configured nonstop timeout, the exception is thrown.

A related exception, InvalidLockAfterRejoinException, can be thrown during or after client rejoin (see Using Rejoin to Automatically Reconnect Terracotta Clients). This exception occurs when an unlock operation takes place on a lock obtained *before* the rejoin attempt completed.

TIP: Use try-finally Blocks

To ensure that locks are released properly, application code using Ehcache lock APIs should encapsulate lock-unlock operations with try-finally blocks: myLock.acquireLock(); try { // Do some work. } finally { myLock.unlock(); }

How Configuration Affects Element Eviction

Element eviction is a crucial part of keeping cluster resources operating efficiently. Element eviction and expiration are related, but an expired element is not necessarily evicted immediately and an evicted element is not necessarily an expired element. Cache elements may be evicted due to resource and configuration constraints, while expired elements are evicted from the Terracotta client when a *get* or *put* operation occurs on that element (sometimes called *inline* eviction).

The Terracotta server array contains the full key set (as well as all values), while clients contain a subset of keys and values based on elements they've faulted in from the server array.

Typically, an expired cache element is evicted, or more accurately flushed, from a client tier to a lower tier when a `get()` or `put()` operation occurs on that element. However, a client may also flush expired, and then unexpired elements, whenever a cache's sizing limit for a specific tier is reached or it is under memory pressure. This type of eviction is intended to meet configured and real memory constraints.

To learn about eviction and controlling the size of the cache, see data life and sizing tiers.

Flushing from clients does not mean eviction from the server array. Servers will evict expired elements and elements can become candidates for eviction from the server array when servers run low on allocated BigMemory. Unexpired elements can also be evicted if they meet the following criteria:

- They are in a cache with infinite TTI/TTL (Time To Idle and Time To Live), or no explicit settings for TTI/TTL. Enabling a cache's `eternal` flag overrides any finite TTI/TTL values that have been set.
- They are not resident on any Terracotta client. These elements can be said to have been "orphaned". Once evicted, they will have to be faulted back in from a system of record if requested by a client.

For more information about Terracotta Server Array data eviction, refer to Automatic Resource Management.

Understanding Performance and Cache Consistency

Cache consistency modes are configuration settings and API methods that control the behavior of clustered caches with respect to balancing data consistency and application performance. A cache can be in one of the following consistency modes:

- **Eventual** – This mode guarantees that data in the cache will eventually be consistent. Read/write performance is substantially boosted at the cost of potentially having an inconsistent cache for brief periods of time. This mode is set using the Ehcache configuration file and cannot be changed programmatically (see the attribute "consistency" in `<terracotta>`).
- **Strong** – This mode ensures that data in the cache remains consistent across the cluster at all times. It guarantees that a read gets an updated value only after all write operations to that value are completed, and that each put operation is in a separate transaction. The use of locking and transaction acknowledgments maximizes consistency at **a potentially substantial cost in performance**. This mode is set using the Ehcache configuration file and cannot be changed programmatically (see the attribute "consistency" in `<terracotta>`).
- **Bulk Load** – This mode is optimized for bulk-loading data into the cache without the slowness introduced by locks or regular eviction. It is similar to the eventual mode, but has batching, higher write speeds, and weaker consistency guarantees. This mode is set using the bulk-load API only (see Bulk-Load API). When turned off, allows the configured consistency mode (either strong or eventual) to take effect again.

Use configuration to set the permanent consistency mode for a cache as required for your application, and the bulk-load mode only during the time when populating (warming) or refreshing the cache.

The following APIs and settings also affect consistency:

- **Explicit Locking** – This API provides methods for cluster-wide (application-level) locking on specific elements in a cache. There is guaranteed consistency across the cluster at all times for operations on elements covered by a lock. When used with the strong consistency mode in a cache, *each cache operation* is committed in a single transaction. When used with the eventual consistency mode in a cache, *all cache operations covered by an explicit lock* are committed in a single transaction. While explicit locking of elements provides fine-grained locking, there is still the potential for contention, blocked threads, and increased performance overhead from managing clustered locks. See Explicit Locking for more information.
- **UnlockedReadsView** – A cache decorator that allows dirty reads of the cache. This decorator can be used only with caches in the strong consistency mode. UnlockedReadsView raises performance for this mode by bypassing the requirement for a read lock. See Unlocked Reads for Consistent Caches (UnlockedReadsView) for more information.
- **Atomic methods** – To guarantee write consistency at all times and avoid potential race conditions for put operations, use the atomic methods `Cache.putIfAbsent(Element element)` and `Cache.replace(Element oldOne, Element newOne)`. These methods throw an

UnsupportedOperationException if used with eventual consistency since a race condition cannot be prevented. To guarantee the return value in eventual consistency, use the cache decorator StronglyConsistentCacheAccessor, or use locks (see Explicit Locking). StronglyConsistentCacheAccessor will use locks with its special substituted versions of the atomic methods. Note that using locks may impact performance.

- **Bulk-loading methods** – Bulk-loading Cache methods putAll(), getAll(), and removeAll() provide high-performance and eventual consistency. These can also be used with strong consistency. If you can use them, it's unnecessary to use bulk-load mode.

To optimize consistency and performance, consider using eventually consistent caches while selectively using appropriate locking in your application where cluster-wide consistency is critical at all times.

Cache Events in a Terracotta Cluster

Cache events are fired for certain cache operations:

- **Evictions** – An eviction on a client generates an eviction event on that client. An eviction on a Terracotta server fires an event on a random client.
- **Puts** – A `put()` on a client generates a put event on that client.
- **Updates** – If a cache uses fast restart, then an update on a client generates a put event on that client.
- **Orphan eviction** – An orphan is an element that exists only on the Terracotta Server Array. If an orphan is evicted, an eviction event is fired on a random client.

See Cache Events Configuration for more information on configuring the scope of cache events.

Handling Cache Update Events

Caches generate put events whenever elements are put or updated. If it is important for your application to distinguish between puts and updates, check for the existence of the element during `put()` operations:

```
if (cache.containsKey(key)) {
  cache.put(element);
  // Action in the event handler on replace.
} else {
  cache.put(element);
  // Action in the event handler on new puts.
}
```

To protect against races, wrap the if block with explicit locks (see Explicit Locking). You can also use the atomic cache methods putIfAbsent() or to check for the existence of an element:

```
// Returns null if successful or returns the existing (old) element.
if((olde = cache.putIfAbsent(element)) == null) {

  // Action in the event handler on new puts.

} else {
  cache.replace(old, newElement); // Returns true if successful.
  // Action in the event handler on replace.
}
```

If your code cannot use these approaches (or a similar workaround), you can force update events for cache updates by setting the Terracotta property `ehcache.clusteredStore.checkContainsKeyOnPut`

at the top of the Terracotta configuration file (`tc-config.xml` by default) before starting the Terracotta Server Array:

```
<tc-properties>
 <property name="ehcache.clusteredStore.checkContainsKeyOnPut" value="true" />
</tc-properties>
```

Enabling this property can substantially degrade performance.

Configuring Caches for High Availability

Enterprise Ehcache caches provide the following High Availability (HA) settings:

- **Non-blocking cache** – Also called nonstop cache. When enabled, this attribute gives the cache the ability to take a configurable action after the Terracotta client receives a cluster-offline event. See Non-Blocking Disconnected (Nonstop) Cache for more information.
- **Rejoin** – The rejoin attribute allows a Terracotta client to reconnect to the cluster after it receives a cluster-online event. See Using Rejoin to Automatically Reconnect Terracotta Clients for more information.

To learn about configuring HA in a Terracotta cluster, see Configuring Terracotta Clusters For High Availability.

Using Rejoin to Automatically Reconnect Terracotta Clients

A Terracotta client may disconnect and be timed out (ejected) from the cluster. Typically, this occurs because of network communication interruptions lasting longer than the configured HA settings for the cluster. Other causes include long GC pauses and slowdowns introduced by other processes running on the client hardware.

You can configure clients to automatically rejoin a cluster after they are ejected. If the ejected client continues to run under nonstop cache settings, and then senses that it has reconnected to the cluster (receives a clusterOnline event), it can begin the rejoin process.

Note the following about using the rejoin feature:

- Rejoin is for CacheManagers with only nonstop caches. If one or more of a CacheManager's caches is not set to be nonstop, and rejoin is enabled, an exception is thrown at initialization. An exception is also thrown in this case if a cache is created programmatically without nonstop.
- Clients rejoin as new members and will wipe all cached data to ensure that no pauses or inconsistencies are introduced into the cluster.
- Any nonstop-related operations that begin (and do not complete) before the rejoin operation completes may be unsuccessful and may generate a NonStopCacheException.
- If a client with rejoin enabled is running in a JVM with Terracotta clients that do not have rejoin, then only that client will rejoin after a disconnection. The remaining clients cannot rejoin and may cause the application to behave unpredictably.
- Once a client rejoins, the clusterRejoined event is fired on that client only.

Configuring Rejoin

The rejoin feature is disabled by default. To enable the rejoin feature in an Enterprise Ehcache client, follow these steps:

1. Ensure that all of the caches in the Ehcache configuration file where rejoin is enabled have nonstop enabled.
2. Ensure that your application does not create caches on the client without nonstop enabled.
3. Enable the rejoin attribute in the client's `<terracottaConfig>` element:

```
<terracottaConfig url="myHost:9510" rejoin="true" />
```

For more options on configuring `<terracottaConfig>`, see the configuration reference.

Working With Transactional Caches

Transactional caches add a level of safety to cached data and ensure that the cached data and external data stores are in sync. Distributed caches can participate in Java Transaction API (JTA) transactions as a fully compliant XA resource. This is useful in JTA applications requiring caching, or where cached data is critical and must be persisted and remain consistent with System of Record data.

However, transactional caches are slower than non-transactional caches due to the overhead from having to write transactionally. Transactional caches also have the following restrictions:

- Data can be accessed only transactionally, even for read-only purposes. You must encapsulate data access with `begin()` and `commit()` statements. This may not be necessary under certain circumstances (see, for example, the discussion on Spring in Transactions).
- `copyOnRead` and `copyOnWrite` must be enabled. These `<cache>` attributes are "false" by default and must set to "true".
- Caches must be strongly consistent. A transactional cache's `consistency` attribute must be set to "strong".
- Nonstop caches cannot be made transactional except in strict mode (xa_strict). Transactional caches in other modes must *not* contain the `<nonstop>` subelement.
- Decorating a transactional cache with UnlockedReadsView can return inconsistent results for data obtained through UnlockedReadsView. Puts, and gets not through UnlockedReadsView, are not affected.
- Objects stored in a transactional cache must override `equals()` and `hashCode()`. If overriding `equals()` and `hashCode()` is not possible, see Implementing an Element Comparator.
- Caches can be dynamically changed to bulk-load mode, but any attempt to perform a transaction when this is the case will throw a `CacheException`.

For more information about transactional caches, see this API page.

You can choose one of three different modes for transactional caches:

- Strict XA – Has full support for XA transactions. May not be compatible with transaction managers that do not fully support JTA.
- XA – Has support for the most common JTA components, so likely to be compatible with most transaction managers. But unlike strict XA, may fall out of sync with a database after a failure (has no recovery). Integrity of cache data, however, is preserved.

- Local – Local transactions written to a local store and likely to be faster than the other transaction modes. This mode does not require a transaction manager and does not synchronize with remote data sources. Integrity of cache data is preserved in case of failure.

NOTE: Deadlocks

Both the XA and local mode write to the underlying store synchronously and using pessimistic locking. Under certain circumstances, this can result in a deadlock, which generates a DeadLockException after a transaction times out and a commit fails. Your application should catch DeadLockException (or TransactionException) and call `rollback()`. Deadlocks can have a severe impact on performance. A high number of deadlocks indicates a need to refactor application code to prevent races between concurrent threads attempting to update the same data.
These modes are explained in the following sections.

Strict XA (Support for All JTA Components)

Note that Ehcache as an XA resource:

- Has an isolation level of ReadCommitted.
- Updates the underlying store asynchronously, potentially creating update conflicts. With this optimistic locking approach, Ehcache may force the transaction manager to roll back the entire transaction if a `commit()` generates a RollbackException (indicating a conflict).
- Can work alongside other resources such as JDBC or JMS resources.
- Guarantees that its data is always synchronized with other XA resources.
- Can be configured on a per-cache basis (transactional and non-transactional caches can exist in the same configuration).
- Automatically performs enlistment.
- Can be used standalone or integrated with frameworks such as Hibernate.
- Is tested with the most common transaction managers by Atomikos, Bitronix, JBoss, WebLogic, and others.

Configuration

To configure a cache as an XA resource able to participate in JTA transactions, the following `<cache>` attributes must be set as shown:

- transactionalMode="xa_strict"
- copyOnRead="true"
- copyOnWrite="true"

In addition, the `<cache>` subelement `<terracotta>` must not have clustering disabled.

For example, the following cache is configured for JTA transactions with strict XA:

```
<cache name="com.my.package.Foo"
     maxEntriesLocalHeap="500"
     eternal="false"
     copyOnRead="true"
     copyOnWrite="true"
     consistency="strong"
     transactionalMode="xa_strict">
   <persistence strategy="distributed"/>
```

```
   <terracotta />
</cache>
```

Any other XA resource that could be involved in the transaction, such as a database, must also be configured to be XA compliant.

Usage

Your application can directly use a transactional cache in transactions. This usage must occur after the transaction manager has been set to start a new transaction and before it has ended the transaction.

For example:

```
...
myTransactionMan.begin();
Cache fooCache = cacheManager.getCache("Foo");
fooCache.put("1", "Bar");
myTransactionMan.commit();
...
```

If more than one transaction writes to a cache, it is possible for an XA transaction to fail. See Avoiding XA Commit Failures With Atomic Methods for more information.

Setting Up Transactional Caches in Hibernate

If your application is using JTA, you can set up transactional caches in a second-level cache with Ehcache for Hibernate. To do so, ensure the following:

Ehcache

- You are using Ehcache 2.1.0 or higher.
- The attribute `transactionalMode` is set to "xa" or "xa-strict".
- The cache is clustered (the <cache> element has the subelement <terracotta clustered="true">). For example, the following cache is configured to be transactional:
- The cache UpdateTimestampsCache is not configured to be transactional. Hibernate updates to `org.hibernate.cache.UpdateTimestampsCache` prevent it from being able to participate in XA transactions.

Hibernate

- You are using Hibernate 3.3.
- The factory class used for the second-level cache is `net.sf.ehcache.hibernate.EhCacheRegionFactory`.
- Query cache is turned off.
- The value of `current_session_context_class` is `jta`.
- The value of `transaction.manager_lookup_class` is the name of a TransactionManagerLookup class (see your Transaction Manager).
- The value of `transaction.factory_class` is the name of a TransactionFactory class to use with the Hibernate Transaction API.
- The cache concurrency strategy is set to TRANSACTIONAL. For example, to set the cache concurrency strategy for `com.my.package.Foo` in `hibernate.cfg.xml`:

Or in a Hibernate mapping file (hbm file):

```
<cache usage="transactional"/>
```

Or using annotations:

```
@Cache(usage=CacheConcurrencyStrategy.TRANSACTIONAL)
public class Foo {...}
```

WARNING: Use the TRANSACTIONAL concurrency strategy with transactional caches only. Using with other types of caches will cause errors.

XA (Basic JTA Support)

Transactional caches set to "xa" provide support for basic JTA operations. Configuring and using XA does not differ from using local transactions (see Local Transactions), except that "xa" mode requires a transaction manager and allows the cache to participate in JTA transactions.

NOTE: Atomikos Transaction Manager

When using XA with an Atomikos transaction Manager, be sure to set `com.atomikos.icatch.threaded_2pc=false` in the Atomikos configuration. This helps prevent unintended rollbacks due to a bug in the way Atomikos behaves under certain conditions.

For example, the following cache is configured for JTA transactions with XA:

```
<cache name="com.my.package.Foo"
     maxEntriesLocalHeap="500"
     eternal="false"
     copyOnRead="true"
     copyOnWrite="true"
     consistency="strong"
     transactionalMode="xa">
   <persistence strategy="distributed"/>
   <terracotta />
</cache>
```

Any other XA resource that could be involved in the transaction, such as a database, must also be configured to be XA compliant.

Local Transactions

Local transactional caches (with the `transactionalMode` attribute set to "local") write to a local store using an API that is part of the Ehcache core API. Local transactions have the following characteristics:

- Recovery occurs at the time an element is accessed.
- Updates are written to the underlying store immediately.
- Get operations on the underlying store may block during commit operations.

To use local transactions, instantiate a TransactionController instance instead of a transaction manager instance:

```
TransactionController txCtrl = cacheManager.getTransactionController();
...
txCtrl.begin();
Cache fooCache = cacheManager.getCache("Foo");
```

```
fooCache.put("1", "Bar");
txCtrl.commit();
...
```

You can use `rollback()` to roll back the transaction bound to the current thread.

TIP: Finding the Status of a Transaction on the Current Thread

You can find out if a transaction is in process on the current thread by calling `TransactionController.getCurrentTransactionContext()` and checking its return value. If the value isn't null, a transaction has started on the current thread.

Commit Failures and Timeouts

Commit operations can fail if the transaction times out. If the default timeout requires tuning, you can get and set its current value:

```
int currentDefaultTransactionTimeout = txCtrl.getDefaultTransactionTimeout();
...
txCtrl.setDefaultTransactionTimeout(30); // in seconds -- must be greater than zero.
```

You can also bypass the commit timeout using the following version of `commit()`:

```
txCtrl.commit(true); // "true" forces the commit to ignore the timeout.
```

Avoiding XA Commit Failures With Atomic Methods

If more than one transaction writes to a cache, it is possible for an XA transaction to fail. In the following example, if a second transaction writes to the same key ("1") and completes its commit first, the commit in the example may fail:

```
...
myTransactionMan.begin();
Cache fooCache = cacheManager.getCache("Foo");
fooCache.put("1", "Bar");
myTransactionMan.commit();
...
```

One approach to prevent this type of commit failure is to use one of the atomic put methods, such as `Cache.replace()`:

```
myTransactionMan.begin();
int val = cache.get(key).getValue();  // "cache" is configured to be transactional.
Element olde = new Element (key, val);

// True only if the element was successfully replaced.
if (cache.replace(olde, new Element(key, val + 1)) {
 myTransactionMan.commit();
}
else { myTransactionMan.rollback(); }
```

Another useful atomic put method is `Cache.putIfAbsent(Element element)`, which returns null on success (no previous element exists with the new element's key) or returns the existing element (the put is not executed). Atomic methods cannot be used with null elements, or elements with null keys.

Implementing an Element Comparator

For all transactional caches, the atomic methods `Cache.removeElement(Element element)` and `Cache.replace(Element old, Element element)` must compare elements for the atomic operation to complete. This requires all objects stored in the cache to override `equals()` and `hashCode()`.

If overriding these methods is not desirable for your application, a default comparator is used (`net.sf.echache.store.DefaultElementValueComparator`). You can also implement a custom comparator and specify it in the cache configuration with <elementValueComparator>:

```
<cache name="com.my.package.Foo"
     maxEntriesLocalHeap="500"
     eternal="false"
     copyOnRead="true"
     copyOnWrite="true"
     consistency="strong"
     transactionalMode="xa">
   <elementValueComparator class="com.company.xyz.MyElementComparator" />
   <persistence strategy="distributed"/>
   <terracotta />
</cache>
```

Custom comparators must implement `net.sf.ehcache.store.ElementValueComparator`.

A comparator can also be specified programmatically.

Working With OSGi

To allow Enterprise Ehcache to behave as an OSGi component, the following attributes should be set as shown:

```
<cache ... copyOnRead="true" ... >
...
  <terracotta ... clustered="true" ... />
...
</cache>
```

Your OSGi bundle will require the following JAR files (showing versions from a BigMemory Max 4.0.0):

- `ehcache-ee-2.7.0.jar`
- `terracotta-toolkit-runtime-ee-4.0.0.jar`
- `slf4j-api-1.6.6.jar`
- `slf4j-nop-1.6.1.jar`

  Or use another appropriate logger binding.

Use the following directory structure:

```
-- net.sf.ehcache
          |
          | - ehcache.xml
          |- ehcache-ee-2.7.0.jar
      |
```

```
      |- terracotta-toolkit-runtime-ee-4.0.0.jar
          |
          | - slf4j-api-1.6.6.jar
          |
          | - slf4j-nop-1.6.6.jar
          |
          | - META-INF/
              | - MANIFEST.MF
```

The following is an example manifest file:

```
Manifest-Version: 1.0
 Export-Package: net.sf.ehcache;version="2.7.0"
 Bundle-Vendor: Terracotta
 Bundle-ClassPath: .,ehcache-ee-2.7.0.jar,terracotta-toolkit-runtime-ee-4.0.0.jar
     ,slf4j-api-1.6.6.jar,slf4j-nop-1.6.6.jar
 Bundle-Version: 2.7.0
 Bundle-Name: EHCache bundle
 Created-By: 1.6.0_15 (Apple Inc.)
 Bundle-ManifestVersion: 2
 Import-Package: org.osgi.framework;version="1.3.0"
 Bundle-SymbolicName: net.sf.ehcache
 Bundle-RequiredExecutionEnvironment: J2SE-1.5
```

Use versions appropriate to your setup.

To create the bundle, execute the following command in the `net.sf.ehcache` directory:

```
jar cvfm net.sf.ehcache.jar MANIFEST.MF *
```

Key Classes and Methods of the BigMemory API

Introduction

BigMemory currently uses Ehcache as its user-facing data access API. See the Ehcache API documentation for details.

The key Ehcache classes used are:

- CacheManager
- Cache
- Element

These classes form the core of the BigMemory API. This document introduces these classes, along with other important components of the BigMemory API.

When applications use BigMemory through the Ehcache API, a CacheManager is instantiated to manage logical data sets (represented as Cache objects in the Ehcache API—though, they may be used to store any kind of data, not just cache data). These data sets contain name-value pairs called Elements.

The logical representations of these key components are actualized mostly through the classes discussed below. These classes' methods provide the main programmatic access to working with Ehcache.

CacheManager

The class `CacheManager` 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 may 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, BigMemory Max 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.

If a CacheManager is part of a cluster, there will also be listener ports which must be unique.

See the API documentation for more information on these methods, including options for passing in configuration. For examples, see Code Samples.

Cache

This 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.

Caches are instantiated from configuration or programmatically using one of the `Cache()` constructors. Certain cache characteristics, such as 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()`, `isSearchable()`, etc.), or perform certain cache-wide operations (for example, flush, load, initialize, dispose, etc.).

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

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.

Element

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()`.

BigMemory Max Search API

Introduction

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

Searchable attributes may 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();
```

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 which 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 ehcache.xml.

```
<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 with:

```
<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 (described in the Defining Attributes section below):

By Configuration

```
<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 out of them. The following example shows this more typical case. Attribute Extractors are explained in more detail in the following section.

```
<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 this Javadoc.

Disk usage with the Terracotta Server Array

Search indexes are stored on Terracotta server disks. The default path is the same directory as the configured `<data>` location. You can customize the path using the `<index>` element in the server's `tc-config.xml` configuration file.

The Terracotta Server Array can be configured to be restartable in addition to including searchable caches, but both of these features require disk storage. When both are enabled, be sure that enough disk space is available. Depending upon the number of searchable attributes, the amount of disk storage required might be 3 times the amount of in-memory data.

It is highly recommended to store the search index (`<index>`) and the Fast Restart data (`<data>`) on separate disks.

Defining Attributes

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

Attributes are extracted from keys or values during search. This is done using `AttributeExtractors`. Extracted attributes must be one of the following supported types:

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

If an attribute cannot be extracted, due to not being found or being 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 later. For example, if an integer value was initially returned for attribute named "Age" by the attribute extractor, then 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 on the `Query` class. So, for example, the attribute for "key" may be referenced in a query by `Query.KEY`. For even greater readability, it is recommended to statically import so that, in this example, you would just 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 which 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 simply 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 one of either "key", "value", or "element". From the starting object a chain of either method calls or field names follows. Method calls and field names can be freely mixed in the chain. Some more examples:

```
<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 allows you to specify the element to inspect and the attribute name, and it returns the attribute value.

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

Provide your extractor class, as shown in the following example:

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

For example, a custom attribute extractor could be passed an Employee object and then extract a specific attribute:

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

If you need to pass state to your custom extractor, you may do so with properties, as shown in the following example:

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

```
</cache>
```

If properties are provided, then 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 that would 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

- In order to use the `DynamicAttributesExtractor`, the cache must be configured to be searchable and dynamically indexable (refer to Making a Cache Searchable above).
- 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, as 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, then this set of attributes will always be queried for extracted values, regardless of whether or not there is a dynamic search attribute extractor 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 will be thrown.
- Clustered BigMemory clients do not share dynamic extractor instances or implementations. In a clustered searchable deployment, the initially configured attribute extractors cannot vary from one client to another, and this is enforced by propagating them across the cluster. However, for dynamic attribute extractors, each clustered client maintains its own dynamic extractor instance, not shared with the others. Each distributed application using dynamic search must therefore maintain its own attribute extraction consistency.

Creating a Query

BigMemory Max Search uses a fluent, object-oriented Query API, following DSL principles, which should be familiar to Java programmers. Here is a simple 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, as shown in this example:

```
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 given them in the configuration. For example:

```
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 may 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 criteria 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, do the following:

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

More complex compound expressions can be further created with extra nesting. See the Expression JavaDoc for a complete list of expressions.

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)`. The full listing of operator shorthand is shown below.

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
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.
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

Making Queries Immutable

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

Obtaining and Organizing Query Results

Queries return a `Results` object which contains a list of objects of class `Result`. Each `Element` in the cache found with a query will be represented as a `Result` object. So if a query finds 350 elements, there will be 350 `Result` objects. An exception to this would be if no keys or attributes are included but aggregators are -- in this case, there will be 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, refer to the Aggregators JavaDoc.

Ordering Results

Query results may be ordered in ascending or descending order by adding an `addOrderBy` clause to the query, which 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

BigMemory Max query results may be grouped similarly to using an SQL GROUP BY statement. The BigMemory GroupBy feature provides the option to group results according to specified attributes by adding 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 like this:

```
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. This necessitates the following rules and considerations when using GroupBy:

- 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 may not be used in a GroupBy clause. This means that `includeKeys()` and `includeValues()` may not 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 will 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 `discard()` to free up resources. In the distributed implementation with Terracotta, resources may be used to hold results for paging or return.

Interrogating Results

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

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

Finding Null (or Not Null) Values

You can find null values (or not-null values) by replacing the null value with the string "NULL" in every element with a null value. A custom attribute extractor can be used to search for the string "NULL" to find null values (or use a not-equal search for not-null values).

If a string cannot be used for element values, then for each field (that may be a search target) set up a related dummy field and assign it a value of "0" (null) or "1" (not null). Queries can then check the dummy field to find nulls or not-nulls.

Sample Application

We have created a simple standalone sample application with few dependencies for you to easily get started with BigMemory Search. You can also check out the source:

```
git clone git://github.com/sharrissf/Ehcache-Search-Sample.git
```

The Ehcache Test Sources page has further examples on how to use each Search feature.

Scripting Environments

BigMemory Search is readily amenable to scripting. The following example shows how to use it with BeanShell:

```
Interpreter i = new Interpreter();
//Auto discover the search attributes and add them to the interpreter's context
Map<String, SearchAttribute> attributes =
    cache.getCacheConfiguration().getSearchAttributes();
for (Map.Entry<String, SearchAttribute> entry : attributes.entrySet()) {
i.set(entry.getKey(), cache.getSearchAttribute(entry.getKey()));
LOG.info("Setting attribute " + entry.getKey());
}
//Define the query and results. Add things which would be set in the GUI i.e.
//includeKeys and add to context
Query query = cache.createQuery().includeKeys();
Results results = null;
i.set("query", query);
i.set("results", results);
//This comes from the freeform text field
String userDefinedQuery = "age.eq(35)";
//Add on the things that we need
String fullQueryString =
    "results = query.addCriteria(" + userDefinedQuery + ").execute()";
i.eval(fullQueryString);
results = (Results) i.get("results");
assertTrue(2 == results.size());
for (Result result : results.all()) {
LOG.info("" + result.getKey());
}
```

Implementation and Performance

BigMemory Max Backed by the Terracotta Server Array

This implementation uses indexes which are maintained on each Terracotta server. With distributed BigMemory Max, the data is sharded across the number of active nodes in the cluster, and the index for each shard is maintained on that shard's server. Searches are performed using the Scatter-Gather pattern. The query executes on each node and the results are then aggregated back in the BigMemory Max that initiated the search.

Search operations perform in O(log n / number of shards) time. Performance is excellent but can be improved simply by adding more servers to the array. Also, because Search results are returned over the network, and the data returned could potentially be very large, techniques to limit return size are recommended. For more information, refer to Best Practices.

Standalone BigMemory Max

BigMemory uses a Search index that is maintained at the local node. The index is stored under a directory in the DiskStore and is available whether or not persistence is enabled. Any overflow from the on-heap tier of the cache, whether to the off-heap tier or to the disk tier, is searched using indexes.

Search operations perform in O(log(n)) time. For tips that can aid performance, refer to Best Practices.

For caches that are on-heap only, the implementation does not use indexes. Instead, it performs a fast iteration of the cache, relying on the very fast access to do the equivalent of a table scan for each query. Each element in the cache is only visited once. Attributes are not extracted ahead of time. They are done during query execution.

On-heap only search operations perform in O(n) time. Check out this Maven-based performance test showing performance of an on-heap-only search. The test shows search performance of an average of representative queries at 4.6 ms for a 10,000 entry cache, and 427 ms for a 1,000,000 entry cache. Accordingly, this implementation is suitable for development and testing.

Best Practices for Optimizing Searches

1. Construct searches wisely by including only the data that is actually required.
    - Only use `includeKeys()` and/or `includeAttribute()` if those values are actually required for your application logic.
    - If you don't 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, then don't use `includeValues()` in the original 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: `includeKeys()` and `includeValues()` have lazy deserialization, which means that keys and values are de-serialized only when `result.getKey()` or `result.getValue()` is called. This means there is a time cost only when the key is needed. However, there is still some time cost with `includeKeys()` and `includeValues()`, so consider carefully when constructing your queries.
2. Searchable keys and values are automatically indexed by default. If you will not be including them in your query, turn off automatic indexing with the following:

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

3. Limit the size of the results set with `query.maxResults(int number_of_results)`. Another recommendation for managing the size of the result set is to use a built-in Aggregator function to return a summary statistic (see the `net.sf.ehcache.search.aggregator` package in this Javadoc).
4. Make your search as specific as possible. Queries with "ILike" criteria and fuzzy (wildcard) searches may take longer than more specific queries. Also, if you are using a wildcard, try making it the trailing part of the string instead of the leading part (`"321*"` instead of `"*123"`). If you want leading wildcard searches, then you should create a `<searchAttribute>` with the string value

reversed in it, so that your query can use the trailing wildcard instead.
5. 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))`.
6. Index dates as integers. This can save time and may even be faster if you have to do a conversion later on.
7. Searches of eventually consistent BigMemory Max data sets are faster because queries are executed immediately, without waiting for pending transactions at the local node to commit. **Note**: This means that if a thread adds an element into an eventually consistent cache and immediately runs a query to fetch the element, it will not be visible in the search results until the update is published to the server.

Concurrency Notes

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

Index Updating

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

For caches with concurrency control, an index will 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

There are several ways unexpected results could present:

- A search returns an Element reference which no longer exists.
- Search criteria select an Element, but the Element has been updated and a new Search would no longer match the Element.
- Aggregators, such as `sum()`, might disagree with the same calculation done by redoing the calculation yourself by re-accessing the cache for each key and repeating the calculation.
- Because the cache is always updated before the search index, it is possible that a value reference may refer to a value that has been removed from the cache. If this happens, the value will be null but the key and attributes which were supplied by the now stale cache index will be non-null. Because values in Ehcache are also allowed to be null, you cannot tell whether your value is null because it has been removed from the cache since 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.

Bulk Loading

Introduction

BigMemory Max has a bulk-loading mode that dramatically speeds up bulk loading into caches using the Terracotta Server Array (TSA). Bulk loading is designed to be used for:

- cache warming - where caches need to be filled before bringing an application online
- periodic batch loading - say an overnight batch process that uploads data

The bulk-load API optimizes bulk-loading of data by removing the requirement for locks and adding transaction batching. The bulk-load API also allows applications to discover whether a cache is in bulk-load mode and to block based on that mode.

API

With bulk loading, the API for putting data into BigMemory Max stays the same. Just use `cache.put(...)` `cache.load(...)` or `cache.loadAll(...)`. What changes is that there is a special mode that suspends the normal distributed-cache consistency guarantees and provides optimised flushing to the TSA (the L2 cache).

NOTE: The Bulk-Load API and the Configured Consistency Mode

The initial consistency mode of a cache is set by configuration and cannot be changed programmatically (see the attribute "consistency" in `<terracotta>`. The bulk-load API should be used for temporarily suspending the configured consistency mode to allow for bulk-load operations.
The following are the bulk-load API methods that are available in `org.terracotta.modules.ehcache.Cache`.

- `public boolean isClusterBulkLoadEnabled()`

  Returns true if a cache is in bulk-load mode (is not consistent) throughout the cluster. Returns false if the cache is not in bulk-load mode (is consistent) anywhere in the cluster.
- `public boolean isNodeBulkLoadEnabled()`

  Returns true if a cache is in bulk-load mode (is not consistent) on the current node. Returns false if the cache is not in bulk-load mode (is consistent) on the current node.
- `public void setNodeBulkLoadEnabled(boolean)`

  Sets a cache's consistency mode to the configured consistency mode (false) or to bulk load (true) on the local node. There is no operation if the cache is already in the mode specified by `setNodeBulkLoadEnabled()`. When using this method on a nonstop cache, a multiple of the nonstop cache's timeout value applies. The bulk-load operation must complete within that timeout multiple to prevent the configured nonstop behavior from taking effect. For more information on tuning nonstop timeouts, see Tuning Nonstop Timeouts and Behaviors.
- `public void waitUntilBulkLoadComplete()`

  Waits until a cache is consistent before returning. Changes are automatically batched and the cache is updated throughout the cluster. Returns immediately if a cache is consistent throughout the cluster.

Note the following about using bulk-load mode:

- Consistency cannot be guaranteed because `isClusterBulkLoadEnabled()` can return false in one node just before another node calls `setNodeBulkLoadEnabled(true)` on the same cache. Understanding exactly how your application uses the bulk-load API is crucial to effectively managing the integrity of cached data.
- If a cache is not consistent, any ObjectNotFound exceptions that may occur are logged.
- `get()` methods that fail with ObjectNotFound return null.
- Eviction is independent of consistency mode. Any configured or manually executed eviction proceeds unaffected by a cache's consistency mode.

Bulk-Load API Example Code

The following example code shows how a clustered application with BigMemory Max can use the bulk-load API to optimize a bulk-load operation:

```
import net.sf.ehcache.Cache;
public class MyBulkLoader {
CacheManager cacheManager = new CacheManager();  // Assumes local ehcache.xml.
  Cache cache = cacheManager.getEhcache(\"myCache\"); // myCache defined in ehcache.xml.
  cache.setNodeBulkLoadEnabled(true); // myCache is now in bulk mode.
// Load data into myCache.

// Done, now set myCache back to its configured consistency mode.
cache.setNodeBulkLoadEnabled(false);
}
```

NOTE: Potentional Error With Non-Singleton CacheManager

Ehcache does not allow multiple CacheManagers with the same name to exist in the same JVM. `CacheManager()` constructors creating non-singleton CacheManagers can violate this rule, causing an error. If your code may 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.

On another node, application code that intends to touch myCache can run or wait, based on whether myCache is consistent or not:

```
...
if (!cache.isClusterBulkLoadEnabled()) {
// Do some work.
}
else {
 cache.waitUntilBulkLoadComplete()
 // Do the work when waitUntilBulkLoadComplete() returns.
}
...
```

Waiting may not be necessary if the code can handle potentially stale data:

```
...
if (!cache.isClusterBulkLoadEnabled()) {
// Do some work.
}
```

```
else {
// Do some work knowing that data in myCache may be stale.
}
...
```

Performance Improvement

The performance improvement is an order of magnitude faster. ehcacheperf (Spring Pet Clinic) now has a bulk load test which shows the performance improvement for using a Terracotta cluster. Consider also that multi-threading is likely to improve performance.

FAQ

How does bulk-loading affect pinned caches?

If a cache has been pinned, switching the cache into bulk-load mode removes the cached data. The data will then be faulted in from the TSA as it is needed.

Are there any alternatives to putting the cache into bulk-load mode?

Bulk-loading Cache methods putAll(), getAll(), and removeAll() provide high-performance and eventual consistency. These can also be used with strong consistency. If you can use them, it's unnecessary to use bulk-load mode. See the API documentation for details.

Why does the bulk loading mode only apply to Terracotta clusters?

Ehcache, both standalone and replicated, is already very fast and nothing needed to be added.

How does bulk load with RMI distributed caching work?

The core updates are very fast. RMI updates are batched by default once per second, so bulk loading will be efficiently replicated.

Bulk Loading and Nonstop

If a nonstop cache is bulk-loaded, a multiplier is applied to the configured nonstop timeout whenever the method `net.sf.ehcache.Ehcache.setNodeBulkLoadEnabled(boolean)` is used. The default value of the multiplier is 10. You can tune the multiplier using the `bulkOpsTimeoutMultiplyFactor` system property:

```
-Dnet.sf.ehcache.nonstop.bulkOpsTimeoutMultiplyFactor=10
```

For a bulk-loaded nonstop cache, the cache size displayed in the TMC is subject to the `bulkOpsTimeoutMultiplyFactor`. Increasing this multiplier on the clients can facilitate more accurate size reporting.

This multiplier also affects the methods `net.sf.ehcache.Ehcache.getAll()`, `net.sf.ehcache.Ehcache.removeAll()`, and `net.sf.ehcache.Ehcache.removeAll(boolean)`.

Performance Tips

Bulk Loading on Multiple Nodes

The implementation scales very well when the load is split up against multiple Ehcache CacheManagers on multiple machines. Adding nodes for bulk loading is likely to improve performance.

Why not run in bulk load mode all the time

Terracotta clustering provides consistency, scaling and durability. Some applications will require consistency, or not for some caches, such as reference data. It is possible to run a cache permanently in inconsistent mode.

Refresh Ahead

Introduction

This pattern is intended to proactively update cached data to avoid serving stale data. It is also a solution for the "thundering herd" problem of read-through caching.

Inline Refresh Ahead

Inline refresh allows caches to automatically refresh entries based on a timer. Entries whose age reaches the configured time limit, and are accessed, are reloaded by CacheLoader methods.

Configuring Inline Refresh

Inline refresh ahead is configured per cache using a cache decorator:

```
<cache ...

  <cacheLoaderFactory class="com.company.my.ConcreteCacheLoaderFactory"
      properties="some_property=1, some_other_property=2" />

  <cacheDecorator class="net.sf.ehcache.constructs.refreshahead.RefreshAheadCacheFactory"
      properties="name=myCacheRefresher,
      timeToRefreshSeconds=200,
      batchSize=10,
      numberOfThreads=4,
      maximumBacklogItems=100,
      evictOnLoadMiss=true" />
  </cacheDecorator>
  ...
</cache>
```

The cache-decorator class is required for implementing the refresh-ahead mechanism. Note that inline-refresh configuration properties are optional unless marked REQUIRED. The following table describes these properties.

Property	Definition	Default Value
name	The name used to identify the cache decorator. If left null, the cache decorator is accessed instead of the underlying cache when the cache's name is referenced.	N/A
timeToRefreshSeconds	REQUIRED. The number of seconds an entry can exist in the cache before it is refreshed (reloaded) on access. Expired items that have yet to be evicted cannot be refreshed.	N/A
maximumBacklogItems	REQUIRED. The maximum number of refresh requests allowed in the refresh work queue. Once this limit is exceeded, items are dropped from the queue to prevent potentially overtaxing resources. If refresh requests are queued faster than they are being cleared, this limit can be exceeded.	N/A
batchSize	The maximum number of refresh requests that can be batched for processing by a thread. For more frequent processing of requests—at a	100

	cost to performance—lower the value.	
numberOfThreads	The number of threads to use for background refresh operations on the decorator. If multiple cache loaders are configured, refresh operations occur in turn.	1
evictOnLoadMiss	A boolean to force an immediate eviction on a reload miss (true) or to not evict before normal eviction takes effect (false). If true, entries that do not refresh due to a failure of the reload operation (all cacheloaders fail) are evicted even if they have not expired based on time-to-live or time-to-idle settings.	false

How timeToRefreshSeconds Works With Expiration

The `timeToRefreshSeconds` value is at least how old an entry must be before a get operation triggers an automatic refresh. The refresh is a reload operation that resets an entry's time-to-live (TTL) countdown. A Time-to-idle (TTI) setting is reset upon access and so has no effect if it is greater than the refresh time limit (TTR), unless no access occurs.

For example, for a cache with a TTR of 10 seconds, a TTL of 30 seconds, and a TTI of 0 (infinite idle time), any access to an entry that occurs between 10 and 30 seconds triggers an automatic refresh of that entry. This assumes that the entry is reloaded ahead of the TTL limit so that the TTL timer is reset.

Scheduled Refresh Ahead

You can configure refresh-ahead operations to occur on a regular schedule using Quartz scheduled jobs. These jobs can asynchronously load new values using configured cache loaders on a schedule defined by a cron expression. This type of refresh operation is useful when all or a large portion of a cache's data must be updated regularly.

Note that at least one CacheLoader must be configured for caches using scheduled refresh.

Configuring Scheduled Refresh

Scheduled refresh ahead is configured using a cache extension:

```
<cache ...

   <cacheLoaderFactory class="com.company.my.ConcreteCacheLoaderFactory"
     properties="some_property=1, some_other_property=2" />

   <cacheExtensionFactory
     class="net.sf.ehcache.constructs.scheduledrefresh.ScheduledRefreshCacheExtensionFactory"
     properties="batchSize=100;
                 quartzJobCount=2;
                 cronExpression=0 0 12 * * ?"
     propertySeparator=";"/>
...
</cache>
```

Because a cron expression can contain commas (","), which is the default delimiter for properties, the `propertySeparator` attribute can be used to specify a different delimiter.

The cache-extension class is required for implementing the refresh-ahead mechanism. This cache extension is

responsible for instantiating a dedicated Quartz scheduler with its own RamJobStore for each non-clustered cache configured for scheduled refresh ahead.

Note that scheduled-refresh configuration properties are optional unless marked REQUIRED. The following table describes these properties. Classes are found in `net.sf.ehcache.constructs.scheduledrefresh`.

Property	Definition	Default Value
cronExpression	REQUIRED. This is a string expression, in cron format, dictating when the refresh job should be run. See the Quartz documentation for more information.	N/A
batchSize	(Integer) This is the number of keys which a single job will refresh at a time as jobs are scheduled.	100
keyGenerator	If null, `SimpleScheduledRefreshKeyGenerator` is used to enumerate all keys in the cache. Or, to filter the keys to be refreshed, provide a class that implements the `ScheduledRefreshKeyGenerator` interface, overring its `generateKeys()` method.	Null
useBulkLoad	If this is true, the node will be put in bulkload mode as keys are loaded into the cache.	false
quartzJobCount	Controls the number of threads in the Quartz job store created by this instance of ScheduleRefreshCacheExtension. At least 2 threads are needed, 1 for the controlling job and 1 to run refresh batch jobs. There is always an additional thread created with respect to the value set. Therefore the default value of "2" creates the controlling job (always created) and 2 worker threads to process incoming jobs.	2
scheduledRefreshName	By default, the underlying Quartz job store is named based on a combination of CacheManager and cache names. If two or more ScheduledRefreshExtensions are going to be attached to the same cache, this property provides a necessary way to name separate instances.	Null
evictOnLoadMiss	A boolean to force an immediate eviction on a reload miss (true) or to not evict before normal eviction takes effect (false). If true, entries that do not refresh due to a failure of the reload operation (all cacheloaders fail) are evicted even if they have not expired based on time-to-live or time-to-idle settings.	true
jobStoreFactory	By default, a RAMJobStore is used by the Quartz instance. To define a custom job store, implement the `ScheduledRefreshJobStorePropertiesFactory` interface.	See below.
tcConfigUrl	The URL to a Terracotta server in the form <server-address>:<tsa-port>, same as the Quartz property `org.quartz.jobStore.tcConfigUrl`. Setting this allows the Quartz instances supporting scheduled refresh to run distributed in a Terracotta cluster.	N/A
parallelJobCount	Maximum number of simultaneous refresh batch jobs that can run when Quartz instances supporting scheduled refresh run	Equals the value of

	distributed in a Terracotta cluster. Tune this value to improve performance by beginning with a value equal to the number of (identical) distributed scheduled-refresh nodes multiplied by the total number of Quartz nodes. For example, if there are 4 instances and 4 Quartz nodes, set this value to 16 and confirm that resources are not being overly utilized. If resources are still underutilized, increase this value and gauge the impact on resources and performance.	quartzJobCount.

The default jobStoreFactory is `ScheduledRefreshRAMJobStoreFactory`. However, if the property `tcConfigUrl` is specified, the jobStoreFactory used is `ScheduledRefreshTerracottaJobStoreFactory`.

Implementing the CacheLoader

Implementing CacheLoaderFactory through the BigMemory Max API is required to effect reloading of entries for refresh-ahead operations. In configuration, specify the concrete class that extends `net.sf.ehcache.loader.CacheLoaderFactory` and call `createCacheLoader(myCache, properties)` to create the cache's cacheloader. For example, if the configured concrete class is the following:

```
<cache name="myCache" ...

  <cacheLoaderFactory class="com.company.my.ConcreteCacheLoaderFactory"
      Properties="some_property=1, some_other_property=2" />

  <!-- Additional cacheLoaderFactory elements can be added.
       These form a chain so that if a CacheLoader returns null,
       the next in line is tried. -->

  ...
</cache>
```

then it can be used programmatically:

```
CacheLoader cacheLoader = ConcreteCacheLoaderFactory.createCacheLoader(myCache, properties);
// Custom properties can be passed to the implemented CacheLoader.
```

`cacheLoader` must implement the `CacheLoader.loadAll()` method to load the refreshed entries.

Transactions in Ehcache

Introduction

BigMemory Max suports Global Transactions, with "xa_strict" and "xa" modes, and Local Transactions with "local" mode.

For more discussion on these modes, and related topics, see Working With Transactional Caches.

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.

Transactional Methods

The following methods require a transactional context to run:

- put()
- get()
- getQuiet()
- remove()
- getKeys()
- getSize()
- containsKey()
- removeAll()
- putWithWriter()
- removeWithWriter()
- putIfAbsent()
- removeElement()
- replace()

This list is applies to all transactional modes.

All other methods work non-transactionally but can be called on a transactional cache, either within or outside of a transactional context.

Change Visibility

The isolation level offered in BigMemory's Ehcache API 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.

When to use Transactional Modes

Transactional modes are a powerful extension of Ehcache allowing 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 orJMS, 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

The objects you are going to 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` — Those must be overridden because the transactional stores rely on element value comparison. See: `ElementValueComparator` and the `elementValueComparator` configuration setting.

Additional restrictions:

- `transactionalMode` can only be used with `terracotta consistency="strong"`.
- Caches can be dynamically changed to bulk-load mode, but any attempt to perform a transaction when this is the case causes a `CacheException` to be thrown.
- If using "xa_strict", set synchronous writes (in `ehcache.xml`) to prevent potential loss of unwritten data on the client:

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

Configuration

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". Enabling a cache for "xa_strict" transactions is shown in the following example:

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

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. The differences are explained in the sections below.

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 TransactionManager 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", 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", the cache always responds to the TransactionManager'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 Apps

We have three sample applications showing how to use XA with a variety of technologies.

XA Sample App

This sample application uses 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);
}
```

The source code for the demo can be checked out from the Terracotta Forge. A README.txt explains how to get the sample app going.

XA Banking Application

This application shows a real world scenario. A Web app reads <account transfer> messages from a queue and tries to execute these account transfers. With JTA turned on, failures are rolled back so that the cached account balance is always the same as the true balance summed from the database. This app is a Spring-based Java web app 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 TransactionManager. 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`

The source code for the demo can be checked out from the Terracotta Forge. A README.txt explains how to get the sample app going.

Transaction Managers

Automatically Detected Transaction Managers

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

- 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`:

```
<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 TransactionManager in GlassFish v3:

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

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.

Introduction 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

This example shows a transaction which 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

Managing Contention

If two transactions, either standalone or across the cluster, 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. The normal cluster semantics are used. Because transactions only work with consistency=strong caches, the first transaction is the thread that manages to atomically place a soft lock on the Element. (This is done with the CAS based putIfAbsent and replace methods.)

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.

Accordingly, "xa_strict" should only be used where full guarantees are required, otherwise one of the other modes should be used.

FAQ

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.

Is IBM Websphere Transaction Manager supported?

Mostly. "xa_strict" is not supported due to each version of Websphere being a custom implementation, that is, there is no stable interface to implement against. However, "xa", which uses TransactionManager callbacks and "local" 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.

How do transactions interact with Write-behind and Write-through caches?

If your transactional enabled cache is being used with a writer, write operations are queued until transaction commit time. Solely a 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.

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 transactional Ehcache?

WebLogic uses an optimization that is not supported by our 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>
```

How do I make Atomikos work with the Ehcache "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 follows:

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

Explicit Locking

Introduction

BigMemory Max's 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 behaviour 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.

The 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);
```

Example

Here is a brief example:

```
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 it 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.

Write-through and Write-behind Caching with the CacheWriter

Introduction

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 network has an end-of-day procedure where data gets written up 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, and the queue of writes are whittled down 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.

Node time synchronisation

Ideally node times should be synchronised. 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 synchronise 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 and more particularly in clustered usage, 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 caches, auto-versioning will silently fail.
- Verifications with the underlying resource to check if the scheduled write-behind operation is still relevant.

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. And in the distributed caching scenario, using Terracotta clustering extends the same guarantee to the cluster. If using transactions, the cache is the XAResource, and a commit is a commit to the cache. The cache effectively becomes the System Of Record ("SOR"). Terracotta clustering provides HA and durability and can easily act as the SOR. The database then becomes a backup to the SOR. 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 How to Size Caches for more information.

Expiry

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

Introductory Video

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

Sample Application

We have created a sample web application for a raffle which fully demonstrates how to use write behind. You can also checkout the Ehcache Raffle application, that demonstrates Cache Writers and Cache Loaders from github.com.

Configuration

There are many configuration options. See the `CacheWriterConfiguration` for properties that may be set and their effect. 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:

Configuration

```
<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>
```

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's 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);
```

Configuration Attributes

The CacheWriterFactory supports the following attributes:

Configuration 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 optimisations.
- 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. For example, following is the method signature for `cache.putWithWriter(...)`.

```
/**
* 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;
```

See the Cache JavaDoc for the complete API.

SPI

The Ehcache 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 a
* 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 operations.
* <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 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.
* <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);

}
```

FAQ

Is there a way to monitor the write-behind queue size?

Use the method `net.sf.ehcache.statistics.LiveCacheStatistics#getWriterQueueLength()`. 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.

What happens if an exception occurs when the 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

Is there a way to monitor the write-behind queue size?

point.

Blocking Cache and Self-Populating Cache

Introduction

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. Let's say 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, 10 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 about Blocking Cache, refer to this Javadoc.

SelfPopulatingCache

You want to use the BlockingCache, but the requirement to always release the lock creates gnarly code. You also want to think about what you are doing without thinking about the caching. Enter the SelfPopulatingCache. The name SelfPopulatingCache is synonymous with Pull-through cache, which is a common caching term. SelfPopulatingCache though always is in addition to a BlockingCache. SelfPopulatingCache uses a `CacheEntryFactory`, that 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 about Self-populating Cache, refer to this Javadoc.

Terracotta Cluster Events

Introduction

The Terracotta distributed BigMemory Max cluster events API provides access to Terracotta cluster events and cluster topology. This event-notification mechanism reports events related to the nodes in the Terracotta cluster, not cache events.

Cluster Topology

The interface `net.sf.ehcache.cluster.CacheCluster` provides methods for obtaining topology information for a Terracotta cluster. The following methods are available:

- `String getScheme()`

  Returns a scheme name for the cluster information. Currently `TERRACOTTA` is the only scheme supported. The scheme name is used by CacheManager.getCluster() to return cluster information.
- `Collection<ClusterNode> getNodes()`

  Returns information on all the nodes in the cluster, including ID, hostname, and IP address.
- `boolean addTopologyListener(ClusterTopologyListener listener)`

  Adds a cluster-events listener. Returns true if the listener is already active.
- `boolean removeTopologyListener(ClusterTopologyListener)`

  Removes a cluster-events listener. Returns true if the listener is already inactive.

The interface `net.sf.ehcache.cluster.ClusterNode` provides methods for obtaining information on specific cluster nodes.

```
public interface ClusterNode {
/**
* Get a unique (per cluster) identifier for this node.
*
* @return Unique per cluster identifier
*/
String getId();
/**
* Get the host name of the node
*
* @return Host name of node
*/
String getHostname();
/**
* Get the IP address of the node
*
* @return IP address of node
*/
String getIp();
}
```

Listening For Cluster Events

The interface `net.sf.ehcache.cluster.ClusterTopologyListener` provides methods for detecting the following cluster events:

```
public interface ClusterTopologyListener {
/**
* A node has joined the cluster
*
* @param node The joining node
*/
void nodeJoined(ClusterNode node);
/**
* A node has left the cluster
*
* @param node The departing node
*/
void nodeLeft(ClusterNode node);
/**
* This node has established contact with the cluster and can execute clustered operations.
*
* @param node The current node
*/
void clusterOnline(ClusterNode node);
/**
* This node has lost contact (possibly temporarily) with the cluster and cannot execute
* clustered operations
*
* @param node The current node
*/
void clusterOffline(ClusterNode node);
}
/**
* This node lost contact and rejoined the cluster again.
*
```

* This event is only fired in the node which rejoined and not to all the connected nodes * @param oldNode The old node which got disconnected * @param newNode The new node after rejoin */ void clusterRejoined(ClusterNode oldNode, ClusterNode newNode);

Example Code

This example prints out the cluster nodes and then registers a `ClusterTopologyListener` which prints out events as they happen.

```
CacheManager mgr = ...
CacheCluster cluster = mgr.getCluster("TERRACOTTA");
  // Get current nodes
Collection<ClusterNode> nodes = cluster.getNodes();
for(ClusterNode node : nodes) {
  System.out.println(node.getId() + " " + node.getHostname() + " " + node.getIp());
}
  // Register listener
cluster.addTopologyListener(new ClusterTopologyListener() {
  public void nodeJoined(ClusterNode node) { System.out.println(node + " joined"); }
  public void nodeLeft(ClusterNode node) { System.out.println(node + " left"); }
  public void clusterOnline(ClusterNode node) { System.out.println(node + " enabled"); }
  public void clusterOffline(ClusterNode node) { System.out.println(node + " disabled"); }
```

```
  public void clusterRejoined(ClusterNode node, ClusterNode newNode) {
    System.out.println(node + " rejoined the cluster as " + newNode);
  }
});
```

Troubleshooting

In most cases the Terracotta cluster-events API behaves as expected. Unexpected results can occur under the circumstances described below.

getCluster Returns Null For Programmatically Created CacheManagers

If a CacheManager instance is created and configured programmatically (without an `ehcache.xml` or other external configuration resource), `getCluster("TERRACOTTA")` may return null even if a Terracotta cluster exists. To ensure that cluster information is returned in this case, get a cache that is clustered with Terracotta:

```
// mgr created and configured programmatically.
CacheManager mgr = new CacheManager();
// myCache has Terracotta clustering.
Cache cache = mgr.getEhcache("myCache");
// A Terracotta client has started, making available cluster information.
CacheCluster cluster = mgr.getCluster("TERRACOTTA");
```

nodeJoined for the Current Node

Since the current node joins the cluster before code adding the topology listener runs, the current node may never receive the nodeJoined event. You can detect if the current node is in the cluster by checking if the cluster is online:

```
cluster.addTopologyListener(cacheListener);
if(cluster.isClusterOnline()) {
  cacheListener.clusterOnline(cluster.getCurrentNode());
}
```

Multiple NodeJoined Events in the Same JVM

Since multiple Terracotta clients can exist in the same JVM, for example when using the Terracotta Toolkit and Ehcache, multiple NodeJoined events can be generated in that JVM. Remote clients will not be able to differentiate between the clients that generated the NodeJoined events.

Cache Decorators

Introduction

BigMemory Max 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 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.

Creating a Decorator

Programmatically

Cache decorators are created as follows:

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

The class must implement Ehcache.

By Configuration

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

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

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

Adding decorated caches to the CacheManager

Having created a decorator programmatically it is generally useful to put it in a place where multiple threads may 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 overwridden Ehcache methods will take on new behaviours 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.

Using CacheManager.addDecoratedCache()

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 {

Built-in Decorators

BlockingCache

A blocking decorator for an Ehcache, backed by a {@link Ehcache}.

It 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 is useful for constructing read-through or self-populating caches. BlockingCache is used by `CachingFilter`.

SelfPopulatingCache

A selfpopulating 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

These are decorated. See Cache Exception Handlers for full details.

Event Listeners

CacheManager Event Listeners

BigMemory Max's Ehcache implementation includes CacheManager event listeners. These listeners allow implementers to register callback methods that will be executed when a CacheManager event occurs. Cache 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.

Configuration

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.

Cache Event Listeners

BigMemory Max's Ehcache implementation also includes 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.

Configuration

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` which is used to create 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 the 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. See the page on Classloading for details on how the loading of these classes will be done.

Adding a Listener Programmatically

To add a listener programmatically, follow this example:

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

Example: Running Multiple Event Listeners on Separate Nodes

If a listener B in one node is listening for an event generated by the action of listener A on another node, it will fail to receive an event unless listener A performs the action in a different thread.

For example, if listener A detects a put into cache A and in turn puts an element into cache B, then listener B should receive an event (if it is correctly registered to cache B). However, with the following code, listener B would fail to receive the event generated by the put:

```
// This method is within listener A
public void notifyElementPut(...) {
  ...
  ...
  cache.put(...);
  ...
  ...

}
```

The following code allows listener B to receive the event:

```
// This method is within listener A
public void notifyElementPut(...) {
  executorService.execute(new Runnable() {
    public void run()
    {
     ...
     ...
     cache.put(...);
     ...
     ...
     }
  ...
  ...
  }
...
```

Cache Exception Handlers

Introduction

BigMemory Max's Ehcache implementation includes Cache 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.

CacheExceptionHandlers may be set either declaratively in the ehcache.xml configuration file, or programmatically.

Declarative Configuration

Cache event listeners are configured per cache. Each cache can have at most one exception handler. An exception handler is configured by adding a `cacheExceptionHandlerFactory` element 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. 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);
}
```

The implementations need to be placed in the classpath accessible to Ehcache. See the page on Classloading for details on how classloading of these classes will be done.

Programmatic Configuration

The following example shows how to add exception handling to a cache, and then adding 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);
```

Cache Extensions

Introduction

BigMemory Max's Ehcache implementation includes 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. See Blocking Cache 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 customization.

Declarative Configuration

Cache extensions are configured per cache. Each cache can have zero or more. A `CacheExtension` is configured 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 it's
* 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. See the page on Classloading for details on how class loading of these classes will be done.

Programmatic Configuration

Cache extensions may also be programmatically added to a Cache as shown.

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

Class Loading and Class Loaders

Introduction

Class loading, within the plethora of environments that Ehcache can be running, could be complex. But with BigMemory Max, 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).

Developing Applications With the Terracotta Toolkit

Introduction

The Terracotta Toolkit is intended for developers working on scalable applications, frameworks, and software tools. The Terracotta Toolkit provides the following features:

- **Ease-of-use** – A stable API, fully documented classes (see the Terracotta Toolkit Javadoc), and a versioning scheme that's easy to understand.
- **Guaranteed compatibility** – Verified by a Terracotta Compliance Kit that tests all classes to ensure backward compatibility.
- **Extensibility** – Includes all of the tools used to create Terracotta products, such as concurrent maps, locks, counters, queues.
- **Flexibility** – Can be used to build clustered products that communicate with multiple clusters.
- **Platform independence** – Runs on any Java 1.6 or 1.7 JVM and requires no boot-jars, agents, or container-specific code.

The Terracotta Toolkit 2.x is available with Terracotta kits version 4.0.0 and higher.

Installing the Terracotta Toolkit

The Terracotta Toolkit is contained in the following JAR file:

```
${BIGMEMORY_HOME}/apis/toolkit/lib/terracotta-toolkit-runtime-ee-<version>.jar
```

The Terracotta Toolkit JAR file should be on your application's classpath or in `WEB-INF/lib` if using a WAR file.

Maven users can add the Terracotta Toolkit as a dependency (shown for BigMemory Max 4.0.0):

```
<dependency>
   <groupId>org.terracotta</groupId>
   <artifactId>terracotta-toolkit-runtime-ee</artifactId>
   <version>4.0.0</version>
</dependency>
```

See the Terracotta kit version you plan to use for the correct API and JAR versions to specify in the dependency block.

The repository is given by the following:

```
<repository>
 <id>terracotta-repository</id>
 <url>http://www.terracotta.org/download/reflector/releases</url>
 <releases>
    <enabled>true</enabled>
 </releases>
</repository>
```

Understanding Versions

The products you create with the Terracotta Toolkit depend on the API at its heart. The Toolkit's API has a version number with a major digit and a minor digit that indicate its compatibility with other versions. The major version number indicates a breaking change, while the minor version number indicates a compatible change. For example, Terracotta Toolkit API version 1.1 is compatible with version 1.0. Version 1.2 is compatible with both versions 1.1 and 1.0. Version 2.0 is not compatible with any version 1.x, but will be forward compatible with any version 2.x.

Working With the Terracotta Toolkit

Introduction

The Terracotta Toolkit provides access to a number of useful classes, or tools, such as distributed collections. All provided Toolkit types are automatically distributed and all changes (except as noted below) are shared across all nodes by the Terracotta Server Array. This gives all nodes the same view of state.

While the Toolkit types differ in their usage and functionality based on characteristics inherited from their parent types, there are certain shared features.

This document provides information on getting started using the Terracotta Toolkit. For more detailed information on Toolkit types, see the Terracotta Toolkit Javadoc.

Initializing the Toolkit

To access the tools in the Toolkit, your application must first initialize the Terracotta Toolkit. Initializing the Terracotta Toolkit always begins with starting a Terracotta client:

```
...
// These classes must be imported:
import org.terracotta.toolkit.*;
...

// A configuration source must be specified in the argument.
// In this case the source is a server and its configured port.
Toolkit toolkit = ToolkitFactory.createToolkit("toolkit:terracotta://localhost:9510");
...
```

When a Terracotta client is started, it must load a Terracotta configuration. Programmatically, `createToolkit()` takes as argument the source for the client's configuration. In the example above, the configuration source is a Terracotta server running on the local host, with port set to 9510. In general, a filename, an URL, or a resolvable hostname or IP address and port number can be used. The specified server instance must be running and accessible before the code that starts the client executes.

The data structures and other tools provided by the Toolkit are automatically distributed (clustered) when your application is run in a Terracotta cluster. Since the Toolkit is obtained from an instantiated client, all Toolkit tools must be used clustered. Unclustered use is not supported in this version.

Adding Rejoin Behavior

Clients can be initialized with the ability to rejoin the cluster after being ejected. Clients rejoin as new clients.

Clients are initialized with rejoin using a Properties object:

```
Properties properties = new Properties();
properties.put("rejoin","true");
Toolkit toolkit = ToolkitFactory
                    .createToolkit("toolkit:terracotta://localhost:9510", properties);
```

Toolkit Data Structures

All data structures must have serializable keys and values. In addition, data structures may have other requirements on keys as noted.

ToolkitStore

The `ToolkitStore` is a key-value data store that is automatically distributed in a Terracotta cluster. The `ToolkitStore` is used similarly to the `ToolkitCache` except that it lacks eviction. It is limited to keys of type String.

To create a `ToolkitStore`, use the following:

```
// Enforce type checking -- can store only Value.class.
ToolkitStore<Value> store1 = toolkit.getStore("myStore1", Value.class);

// Do not enforce type checking -- any type of value can be stored.
ToolkitStore store2 = toolkit.getStore("myStore2",  null);

// Use the provided configuration instead of default configuration.
ToolkitStore<Value> store3 = toolkit.getStore("myStore3", configuration, Value.class);
```

Store Configuration

Configuration properties available for the toolkit are shown in the following table.

Property	Default Value	Definition	Edit Dynamically?
CONSISTENCY	EVENTUAL	EVENTUAL sets no cache-level locks for better performance and allows reads without locks, which means the cache may temporarily return stale data in exchange for substantially improved performance. STRONG uses cache-level locks for immediate cache consistency at a potentially high cost to performance, guaranteeing that after any update is completed no local read can return a stale value. If using strong consistency with off-heap, a large number of locks may need to be stored in client and server heaps. In this case, be sure to test the cluster with the expected data set to detect situations where OutOfMemory errors are likely to occur. SYNCHRONOUS_STRONG adds synch-write locks to STRONG consistency, which otherwise uses asynchronous writes. SYNCHRONOUS_STRONG provides extreme data safety at a very high	No

		performance cost by requiring that a client receive server acknowledgement of a transaction before that client can proceed.	
CONCURRENCY	256	Sets the number of segments for the map backing the underlying server store managed by the Terracotta Server Array. If concurrency is not explicitly set (or set to "0"), the system selects an optimized value.	No
MAX_COUNT_LOCAL_HEAP	0 (no limit)	The maximum number of data entries that the store can have in local heap memory.	Yes (locally)
MAX_BYTES_LOCAL_HEAP	0 (no limit)	The maximum amount of data in bytes that the store can have in local heap memory.	Yes (locally)
MAX_BYTES_LOCAL_OFF_HEAP	0 (no limit)	The maximum amount of data in bytes that the store can have in local off-heap memory.	No
OFF_HEAP_ENABLED	false	Enables ("true") or disables ("false") the store's use of off-heap memory.	No
LOCAL_CACHE_ENABLED	true	Enables ("true") or disables ("false") local caching of distributed store data, forcing all of that cached data to reside on the Terracotta Server Array. Disabling local caching may improve performance for distributed stores in write-heavy use cases.	Yes (locally)

Stores created without explicit configuration are provided a default configuration. To customize configuration, either pass configuration to `ToolkitStore` or edit the store's configuration.

You can also create a default store configuration with `ToolkitCacheConfigBuilder()`:

```
Configuration initiallyDefaultConfig = new ToolkitStoreConfigBuilder().build();
```

This configuration can be passed to a store and further edited. Store configuration is governed by the same rules as cache configuration. To learn more about how to configure a store and edit its configuration, see the cache configuration section, remembering to substitute the "store" types and methods.

ToolkitCache

The `ToolkitCache` is a fully featured cache that is automatically distributed in a Terracotta cluster. It has configurable eviction and also inherits all of the functionality of the Toolkit store. It is limited to keys of type String.

To create a `ToolkitCache`, use the following:

```
// Enforce type checking -- can store only Value.class.
ToolkitCache<Value> cache1 = toolkit.getCache("myCache1", Value.class);
```

```
// Do not enforce type checking -- any type of value can be stored.
ToolkitCache cache2 = toolkit.getCache("myCache2",  null);

// Use the provided configuration instead of default configuration.
ToolkitCache<Value> cache3 = toolkit.getCache("myCache3", configuration, Value.class);

// You can also create a default cache configuration with ToolkitCacheConfigBuilder():

Configuration initiallyDefaultConfig = new ToolkitCacheConfigBuilder().build();
```

This configuration can be passed to a cache and further edited based on certain rules.

Configuring ToolkitCache

Caches created without explicit configuration are provided a default configuration. To customize configuration, either pass configuration to `ToolkitCache` or edit the cache's configuration.

In addition to the configuration properties inherited from `ToolkitStore`, the following configuration properties are available to the `ToolkitCache`:

Property	Default Value	Definition	Edit Dynamically
MAX_TTI_SECONDS	0	The time-to-idle setting for all cache entries. Default (0) is unlimited.	Yes (cluster-wide)
MAX_TTL_SECONDS	0	The time-to-live setting for all cache entries. Default (0) is unlimited.	Yes (cluster-wide)
MAX_TOTAL_COUNT	0	The total number of entries allowed for the cache across all client nodes.	Yes (cluster-wide)
PINNING_STORE	NONE	Cache data is forced to remain in the designated storage tier, if any, and cannot be evicted. Valid values are INCACHE (any tier where the cache's data is located), LOCALHEAP, LOCALMEMORY (heap and off-heap), NONE. Issues due to lack of memory could arise with pinned caches that grow in size.	No

To create a cache with a custom configuration use the following:

```
// Create the configuration with the fluent builder interface.
Configuration configuration = new ToolkitCacheConfigBuilder()
              .maxBytesLocalHeap(2g)
              .offheapEnabled(true)
              .maxBytesLocalOffheap(10g)
              .maxTTLseconds(3600)
              .maxTTIseconds(600)
              .maxTotalCount(40000)
              .pinningStore(LOCALMEMORY)
              .build();

ToolkitCache<String> cache3 = toolkit.getCache("myCache3", configuration, String.class);

// Create the configuration using a builder instance.
    ToolkitCacheConfigBuilder builder = new ToolkitCacheConfigBuilder();
    builder.maxCountLocalHeap(10000);
    builder.offheapEnabled(true);
    builder.maxBytesLocalOffheap(10 * GIGABYTE);
```

```
    ToolkitCache cache4 = toolkit.getCache("myCache4", null);

    builder.apply(cache4);
```

To find the values of specific configuration properties, use the corresponding get methods. For example:

```
   Long maxCountLocalHeap = builder.getMaxCountLocalHeap()
```

Editing Cache Configuration

Some configuration properties can be edited dynamically. For example:

```
   // Change the time-to-live value for cache4.

    builder.maxTTLseconds(3000);

    builder.apply(cache4);
```

In the example above, `builder` reapplies its other configuration properties with the same values without effect. Applying configuration properties with values that match the cache's configuration values leaves those values unchanged.

To learn which configuration properties can be edited dynamically, see the table showing basic configuration and the table showing cache-specific configuration.

Note that default configurations have MAX_COUNT_LOCAL_HEAP in effect, and this cannot be edited:

```
// Create the configuration using a builder instance.
    ToolkitCacheConfigBuilder builder5 = new ToolkitCacheConfigBuilder();
    builder.maxBytesLocalHeap(2g);
    builder.offheapEnabled(true);
    builder.maxBytesLocalOffheap(10 * GIGABYTE);

    // cache5 will have a default configuration.
    ToolkitCache cache5 = toolkit.getCache("myCache5", null);

    // The following will throw an IllegalStateException.
    builder5.apply(cache5);
```

Clustered Collections

Clustered collections, all based on standard Java clustered collections, are found in the package `org.terracotta.collections`. Operations on these collections are locked.

ToolkitBlockingQueue

Since this is a bounded `BlockingQueue`, use `ToolkitBlockingQeue.getCapacity()` to return an integer representing the queue's maximum capacity (maximum number of entries), which is not changeable once the queue has been created. If no capacity is specified, the effective capacity is `Integer.MAX_VALUE`.

To create a `ToolkitBlockingQueue`, use the following:

```
// A blocking queue with no capacity restriction. Restricted to the type Value.class.
ToolkitBlockingQueue<Value> queue1 =
```

```
    toolkit.getBlockingQueue("myBlockingQueue1", Value.class);

// A blocking queue with no capacity or type restrictions.
ToolkitBlockingQueue queue2 = toolkit.getBlockingQueue("myBlockingQueue2", null);

// A blocking queue with specified capacity. Restricted to the type Value.class.
ToolkitBlockingQueue queue3 =
    toolkit.getBlockingQueue("myBlockingQueue3", 1024, Value.class);
```

If the blocking queue specified by `getBlockingQueue()` exists, it is returned instead (no new blocking queue is created). Note that if queue3 exists and has a capacity value different than the one specified in `getBlockingQueue()`, an IllegalArgumentException is thrown.

ToolkitMap and ToolkitSortedMap

These types behave as expected. Note that `ToolkitSortedMap` values must allow ordering (implement Comparable).

To create a `ToolkitMap`, use the following:

```
// Enforce type checking -- can store only Value.class using keys of type Key.class.
ToolkitMap<Key, Value> map = toolkit.getMap("myMap", Key.class, Value.class);

// Do not enforce type checking -- any type of key-value can be stored.
ToolkitMap<Key, Value> map = toolkit.getMap("myMap", null, null);
```

`ToolkitSortedMap` is created similarly.

ToolkitSet, ToolkitSortedSet, and ToolkitList

These types behave as expected, but note the following:

- ToolkitList does not allow null elements.
- SortedSet values must allow ordering (implement Comparable).

All are created similarly. To create a `ToolkitSet`, for example, use the following:

```
// Enforce type checking -- can store only Value.class.
ToolkitSet<Value> set1 = toolkit.getSet("mySet1", Value.class);

// Do not enforce type checking -- any type of value can be stored.
ToolkitSet set2 = toolkit.getSet("mySet2", null);
```

Cluster Information and Messaging

To implement messaging in a Terracotta cluster, you can use the built-in cluster events and custom messages with a clustered notifier.

Cluster Events

You can access cluster information for monitoring the nodes in a Terracotta cluster, as well as obtaining specific information about those nodes.

For example, you can set up a cluster listener to receive events about the status of client nodes:

```
// Start a client and access cluster events and meta data such as topology
// for the cluster that the client belongs to:

ClusterInfo clusterInfo = toolkit.getClusterInfo();

// Register a cluster listener:
clusterInfo.addClusterListener(new ClusterListener());
```

Custom Listener Classes

You can write your own listener classes that implement the event methods, and add or remove your own listeners:

```
MyClusterListener myClusterListener = new MyClusterListener();
clusterInfo.addClusterListener(new myClusterListener());

// To remove a listener:
clusterInfo.removeClusterListener(myClusterListener);
```

Handling Cluster Events

To handle events, use `onClusterEvent()`, which is called whenever a ClusterEvent occurs. See the suggested approaches below.

Custom Implemented Methods

Implement methods to handle each type of event, then use them in a test for the event type.

```
public void onClusterEvent(ClusterEvent event) {

   if (event.getType() == NODE_LEFT) {
      handleNodeLeft();
   }

   if (event.getType() == NODE_JOINED) {
      handleNodeJoined();
   }

   if (event.getType() == OPERATIONS_DISABLED) {
      handleOperationsDisabled();
   }

   if (event.getType() == OPERATIONS_ENABLED) {
      handleOperationsEnabled();
   }

}
```

Switch Case

Implement a switch-case block to handle each element type.

```
public void onClusterEvent(ClusterEvent event) {
    switch (event.getType()) {
        case NODE_JOINED:
              // Handle the fact that a node joined.
```

```
            break;
        case NODE_LEFT:
             // Handle node leaving.
            break;
        case OPERATIONS_ENABLED:
             // Handle operations disabled.
            break;
        case OPERATIONS_ENABLED:
             // Handle operations disabled.
            break;
        default:
             // handle other events generically
    }
}
```

Toolkit Notifier

`ToolkitNotifier` provides simple-to-use, clustered, and customizable messaging.

To create a `ToolkitNotifier`, use the following:

```
// Enforce type checking -- can send/receive only Value.class.
ToolkitNotifier<Value> notifier1 = toolkit.getNotifier("my-notifier1", Value.class);

// Do not enforce type checking -- any type of value can be sent/received.
ToolkitNotifier notifier2 = toolkit.getNotifier("my-notifier2", null);

// Add a notification listener.
notifier1.addNotificiationListener(new MyToolkitNotificationListener());
```

The notification listener is a custom implementation of the `ToolkitNotificationListener` interface. Specifically, implement `ToolkitNotificationListener.onNotification()`.

To send a message, use the following:

```
notifier1.notifyListeners(new Value());
```

Only remote listeners receive the message.

Locks

Clustered locks allow you to perform safe operations on clustered data. The following types of locks are available:

- **READ-WRITE** – This provides an exclusive read lock and a shared read lock.
- **WRITE** – This is an exclusive write lock that blocks all reads and writes. To improve performance, this lock flushes changes to the Terracotta Server Array asynchronously.

To get and use clustered READ-WRITE locks, use the following:

```
ToolkitReadWriteLock rwLock = toolkit.getReadWriteLock("myReadWriteLock");
rwlock.readLock().lock(); // shared read lock
try {
    // some read operation under the lock
} finally {
```

```
    rwlock.readLock().unlock();
}

rwlock.writeLock().lock(); // exclusive write lock
try {
    // some write operation under the lock
} finally {
    rwlock.writeLock().unlock();
}
```

To get and use an exclusive WRITE lock, use the following:

```
ToolkitLock wLock = toolkit.getLock("myWriteLock");
wLock.lock();
try {
    // some operation under the lock
} finally {
    wLock.unlock();
}
```

If you are using BigMemory Max, you can use explicit locking methods on specific keys. See the BigMemory Max documentation for more information.

Barriers

Coordinating independent nodes is useful in many aspects of development, from running more accurate performance and capacity tests to more effective management of workers across a cluster. A clustered barrier is a simple and effective way of coordinating client nodes.

To get a clustered barrier, use the following:

```
// Get a clustered barrier with 4 parties.
ToolkitBarrier clusteredBarrier = toolkit.getBarrier("myBarrier", 4);
```

Note the following:

- `getBarrier()` as implemented in Terracotta Toolkit returns an object that behaves like CyclicBarrier but lacks `barrierAction` support.
- The number of parties must be an integer equal to or greater than 1.
- If `getBarrier()` returns an existing barrier, and the specified number of parties does not match that of the returned barrier, an IllegalArgumentException is thrown.

Utilities

Utilities such as a clustered `AtomicLong` help track counts across a cluster. You can get (or create) a `ToolkitAtomicLong` using `toolkit.getAtomicLong(String name)`.

An events utility, available with `Toolkit.fireOperatorEvent(OperatorEventLevel level, String applicationName, String eventMessage)`, can be used to log specific application-related events.

Shared Characteristics

Certain characteristics are shared by many of the Terracotta Toolkit types.

Nonstop Behavior for Data Structures

The nonstop feature is enabled by specifying it at the time a toolkit is initialized:

```
Toolkit toolkit =
    ToolkitFactory.createToolkit("nonstop-toolkit:terracotta://localhost:9510");
```

Use the following if you are passing a properties object (to add rejoin behavior for example):

```
Toolkit toolkit = ToolkitFactory
            .createToolkit("nonstop-toolkit:terracotta://localhost:9510", properties);
```

Nonstop behavior allows the following limited functionality whenever a Terracotta client disconnects from the TSA:

- Throw exception for read and write operations
- Silently ignore read and write operations (no-op)
- Allow local reads

In addition, nonstop data structures running in a client that is unable to locate the TSA at startup will initiate nonstop behavior as if the client had disconnected.

Toolkit Types Supporting Nonstop

ToolkitCaches and ToolkitStores can be configured with the full range of nonstop behaviors. The following Toolkit types support only throwing a NonStopException: ToolkitAtomicLong, ToolkitList, ToolkitMap<K,V>, ToolkitSortedMap<K,V>, ToolkitLock, ToolkitReadWriteLock, ToolkitNotifier, ToolkitSet, ToolkitSortedSet. Other Toolkit types cannot be configured with nonstop.

Adding Nonstop Config to a Cache

You can add nonstop configuration to Toolkit caches.

```
// Get the cache:
ToolkitCache<String> cache = toolkit.getCache("myCache", configuration, String.class);

// Build the nonstop configuration:
NonStopConfiguration nsconfig = new NonStopConfigurationBuilder()
                                    .nonStopReadTimeoutBehavior(LOCAL_READS)
                                    .timeOutMillis(90000)
                                    .build();

// Register the nonstop configuration with the cache:
NonStopFeature nonStop = toolkit.getFeature(ToolkitFeatureType.NONSTOP);
NonStopConfigurationRegistry registry = nonStop.getNonStopConfigurationRegistry();
registry.registerForInstance(nsconfig, myCache, ToolkitCache)
```

Disposing of Toolkit Objects

Many types are destroyable and can be disposed of using `destroy()`. If the object is distributed, it is destroyed in all nodes if `destroy()` is called in one node. For example, `ToolkitCache.destroy()` disposes of the cache and its data in all nodes where it exists.

Calling `destroy()` on a nonexistent object has no effect (no-op). However, attempting to use a destroyed object throws an IllegalStateException.

The following Toolkit types are destroyable: ToolkitAtomicLong, ToolkitBarrier, ToolkitBlockingQueue, ToolkitCache<K,V>, ToolkitList, ToolkitMap<K,V>, ToolkitNotifier, ToolkitSet, ToolkitSortedMap<K,V>, ToolkitSortedSet, ToolkitStore<K,V>.

Finding the Assigned Name

Most Toolkit types have assigned String names. You can find an object's assigned name using its `getName()` method, even after that object has been destroyed.

The following Toolkit types support `getName()`: ToolkitAtomicLong, ToolkitBarrier, ToolkitBlockingQueue, ToolkitCache<K,V>, ToolkitList, ToolkitLock, ToolkitMap<K,V>, ToolkitNotifier, ToolkitReadWriteLock, ToolkitSet, ToolkitSortedMap<K,V>, ToolkitSortedSet, ToolkitStore<K,V>.

Returning Existing Toolkit Objects

With certain Toolkit types, the `Toolkit` get methods will either return the existing named object or create it. For example, running `Toolkit.getMap("myMap", null, null)` returns myMap if it already exists, otherwise a new map is created with the name "myMap".

The following Toolkit types support this functionality: ToolkitBlockingQueue, ToolkitCache<K,V>, ToolkitList, ToolkitMap<K,V>, ToolkitNotifier, ToolkitSet, ToolkitSortedMap<K,V>, ToolkitSortedSet, ToolkitStore<K,V>.

Names can be reused. If the specified name belonged to a destroyed object, a new object is created using that name.

Terracotta Toolkit Reference

This section describes functional aspects of the Terracotta Toolkit.

Reconnected Client Rejoin

A Terracotta client may disconnect and be timed out (ejected) from the cluster. Typically, this occurs because of network communication interruptions lasting longer than the configured HA settings for the cluster. Other causes include long GC pauses and slowdowns introduced by other processes running on the client hardware.

With the rejoin feature, clients will follow a predictable pattern during temporary disconnections:

1. The client receives a clusterOffline event and attempts to reconnect.
2. The reconnection timeout is reached and the TSA will no longer wait for the client (the client is ejected from the cluster).
3. The client no longer blocks the application, instead throwing RejoinException if any application threads attempt to use its data structures.
4. Upon receiving a clusterOnline event, the client reinitializes its state and joins the cluster as a new client.
5. If the rejoin fails, the client continues as before (throwing RejoinException as expected) until a new clusterOnline event is received.

Note the following about the rejoin process:

- Clients rejoin as new members and will wipe all local cached data to ensure that no pauses or inconsistencies are introduced into the cluster. Data that was pending on the rejoined client (data NOT yet acknowledged and persisted by the server) MAY BE LOST.
- Clients cannot rejoin a new cluster; if the TSA has been restarted and its data has not been persisted, the client can never rejoin and must be restarted.
- Any nonstop-related operations that begin (and do not complete) before the rejoin operation completes may be unsuccessful and may generate a NonStopCacheException. Other operations that begin and do not complete before the rejoin operation completes may throw other exceptions.
- Nonstop and rejoin are independent aspects of the behavior of disconnected clients, but if nonstop is in effect and set to throw an exception, only the NonStopException is thrown. RejoinException is thrown if nonstop is not in effect, or if it is set to no-op or local reads.
- If a Terracotta client with rejoin enabled is running in a JVM with clients that do not have rejoin, then only that client will rejoin after a disconnection. The remaining clients cannot rejoin and may cause the application to behave unpredictably.
- Once a client rejoins, the clusterRejoined event is fired on that client only.

Connection Issues

Client creation can block on resolving URL at this point:

```
TerracottaClient client = new TerracottaClient("myHost:9510");
```

If it is known that resolving "myHost" may take too long or hang, your application can should wrap client instantiation with code that provides a reasonable timeout.

A separate connection issue can occur after the server URL is resolved but while the client is attempting to connect to the server. The timeout for this type of connection can be set using the Terracotta property `l1.socket.connect.timeout` (see First-Time Client Connection).

Multiple Terracotta Clients in a Single JVM

When using the Terracotta Toolkit, you may notice that there are more Terracotta clients in the cluster than expected.

Multiple Clients With a Single Web Application

This situation can arise whenever multiple classloaders are involved with multiple copies of the Toolkit JAR.

For example, to run a web application in Tomcat, one copy of the Toolkit JAR may need to be in the application's `WEB-INF/lib` directory while another may need to be in Tomcat's common `lib` directory to support loading of the context-level . In this case, two Terracotta clients will be running with every Tomcat instance.

Clients Sharing a Node ID

Clients instantiated using the same constructor (a constructor with matching parameters) in the same JVM will share the same node ID. For example, the following clients will have the same node ID:

```
TerracottaClient client1 = new TerracottaClient("myHost:9510");
TerracottaClient client2 = new TerracottaClient("myHost:9511");
```

Cluster events generated from client1 and client2 will appear to come from the same node. In addition, cluster topology methods may return ambiguous or useless results.

Web applications, however, can get a unique node ID even in the same JVM as long as the Terracotta Toolkit JAR is loaded by a classloader specific to the web application instead of a common classloader.

Terracotta Tools Catalog

Introduction

A number of useful tools are available to help you get the most out of installing, testing, and maintaining BigMemory. Unless indicated otherwise, these tools are included with the BigMemory kit.

If a tool has a script associated with it, the name of the script (and its path within appears in parentheses in the title for that tool section. The script file extension is `.sh` for UNIX/Linux and `.bat` for Microsoft Windows.

Detailed guides exist for some of the tools. Check the entry for a specific tool to see if more documentation is available.

Archive Utility (archive-tool)

`archive-tool` is used to gather logs generated by a Terracotta server or client for the purpose of contacting Terracotta with a support query.

Microsoft Windows

```
[PROMPT] %BIGMEMORY_HOME%\server\bin\archive-tool.bat <args>
```

UNIX/Linux

```
[PROMPT] ${BIGMEMORY_HOME}/server/bin/archive-tool.sh <args>
```

where `<args>` are:

- [-n] (No Data - excludes data files)
- [-c] (Client - include files from the client)
- [path to terracotta config xml file (tc-config.xml) or path to logs directory
- [output filename in .zip format]

Database Backup Utility (backup-data)

The backup utility creates a backup of the data being shared by your application by taking a snapshot of the data held by the Terracotta Server Array (TSA). Unless a different directory is specified in configuration, backups are saved to the default directory `${user.dir}/terracotta/backups`.

Configuring Backup

You can override this default behavior by specifying a different backup directory in the server's configuration file using the `<data-backup>` property:

```
<servers>
   <restartable enabled="true"/>
...
   <server host="%i" name="myServer">
   ...
     <data-backup>/Users/myBackups</data-backup>
```

```
    </server>
...
</servers>
```

Note that enabling `<restartable>` mode is required for using the backup utility.

Creating a Backup

The backup utility relies on the Terracotta Management Server (TMS) to locate and execute backups. The TMS must be running and connected to the TSA for the backup to take place.

Microsoft Windows

```
[PROMPT] %BIGMEMORY_HOME%\tools\security\bin\backup-data.bat <args>
```

UNIX/Linux

```
[PROMPT] ${BIGMEMORY_HOME}/tools/security/bin/backup-data.sh <args>
```

where `<args>` are:

- [l] <tms-host:port> – The host and port used to connect to the TMS. If ommitted, `localhost:9889` is used by default.
- [u] <username> – If the TMS requires authentication, a username must be specified.
- [p] <password> – If the TMS requires authentication, a password must be specified.
- [a] <agentID> – Specify the agent ID of a TSA. The agent ID is set as a connection name when the connection to the TSA is configured on the TMS. If no agent ID is provided, the TMS returns a list of configured agent IDs.
- [k] This flag causes invalid TMS SSL certificates to be ignored.

For example, to initiate a backup on a cluster with the agent ID "someConnection":

```
${BIGMEMORY_HOME}/tools/security/bin/backup-data.sh -l my-tms-host:9889 \
    -u admin -p admin -a someConnection -k
```

If initiation is successful, the script reports that the backup process has started. Once the backup is complete, the backup data files can be used to restore data in place of the current data files.

Backup Status (backup-status)

The backup-status script is run from the `tools/security/bin` directory. This script complements the backup-data utility by checking on the status of executed backups for a specified cluster. For example, to return a list of backup operations on the agent myClusterAgent:

```
[PROMPT] ${BIGMEMORY_HOME}/tools/security/bin/backup-status -l http://myTMShost:9889 -a myCluster
```

The backup-status script takes the same arguments as backup-data.

Cluster Thread and State Dumps (debug-tool, cluster-dump)

The cluster and thread- and state-dump debug tools provide a way to easily generate debugging information that can be analyzed locally or forwarded to support personnel. These tools work against the Terracotta Management Server that is monitoring the target Terracotta cluster. All components must be running at the time a tool is used.

debug-tool generates thread dumps for all nodes in the cluster, with each node's dump saved its log file. A flag is available for saving the thread dumps to a single zip file. cluster-dump provides a similar service, but adds each nodes state, including information on locks. Note that these tools can generate a substantial amount of data.

For more information on operating these tools, run the associated script with the `-h` flag. For example:

Microsoft Windows

```
[PROMPT] %BIGMEMORY_HOME%\tools\security\bin\debug-tool.bat -h
```

UNIX/Linux

```
[PROMPT] ${BIGMEMORY_HOME}/tools/security/bin/debug-tool.sh -h
```

Distributed Garbage Collector (run-dgc)

`run-dgc` is a utility that causes the specified cluster to perform distributed garbage collection (DGC). Use run-dgc to force a periodic DGC cycle in environments where inline DGC is not in effect, such as when using Terracotta Toolkit data structures. In most situations, however, automated DGC collection is sufficient for most environments.

This utility relies on the Terracotta Management Server (TMS) to locate and execute backups. The TMS must be running and connected to the TSA for the DGC to be initiated.

Microsoft Windows

```
[PROMPT] %BIGMEMORY_HOME%\tools\security\bin\run-dgc.bat <args>
```

UNIX/Linux

```
[PROMPT] ${BIGMEMORY_HOME}/tools/security/bin/run-dgc.sh <args>
```

where `<args>` are:

- [l] <tms-host:port> – The host and port used to connect to the TMS. If ommitted, `localhost:9889` is used by default.
- [u] <username> – If the TMS requires authentication, a username must be specified.
- [p] <password> – If the TMS requires authentication, a password must be specified.
- [a] <agentID> – Specify the agent ID of a TSA. The agent ID is set as a connection name when the connection to the TSA is configured on the TMS. If no agent ID is provided, the TMS returns a list of configured agent IDs.
- [k] This flag causes invalid TMS SSL certificates to be ignored.

NOTE: Running Concurrent DGC Cycles

Two DGC cycles cannot run at the same time. Attempting to run a DGC cycle on a server while another DGC cycle is in progress generates an error.
For more information on distributed garbage collection, see the TSA architecture guide.

Start and Stop Server Scripts (start-tc-server, stop-tc-server)

Use the `start-tc-server` script to run the Terracotta Server, optionally specifying a configuration file:

Microsoft Windows

```
[PROMPT] %BIGMEMORY_HOME%\server\bin\start-tc-server.bat ^
    [-n <name of server>] [-f <config specification>]
```

UNIX/Linux

```
[PROMPT] ${BIGMEMORY_HOME}/server/bin/start-tc-server.sh \
    [-n <name of server>] [-f <config specification>]
```

`<config specification>` can be one of:

- path to configuration file
- URL to configuration file
- <server host>:<tsa-port> of another running Terracotta Server

Note the following:

- If no configuration is specified, a file named `tc-config.xml` in the current working directory will be used.
- If no configuration is specified and no file named `tc-config.xml` is found in the current working directory, a default configuration will be used.
- If no server is named, and more than one server exists in the configuration file used, an error is printed to standard out and no server is started.

Use the `stop-tc-server` script to cause the Terracotta Server to gracefully terminate:

Microsoft Windows

```
[PROMPT] %BIGMEMORY_HOME%\server\bin\stop-tc-server.bat <host-name> <jmx-port> <args>
```

UNIX/Linux

```
[PROMPT] ${BIGMEMORY_HOME}/server/bin/stop-tc-server.sh <host-name> <jmx-port> <args>
```

where `<args>` are:

- [f] `<file-or-URL>` – Specifies the configuration file to use, as a file path or URL.
- [--force] – Force shutdown of the active server.

In production mode, if the stop-tc-server script detects that the mirror server in STANDBY state isn't reachable, it issues a warning and fails to shut down the active server. If failover is not a concern, you can override this behavior with the `--force` flag.

- [n] `<server-name>` – The name of the server to shut down. Defaults to the local host.
- [s] – If the server is secured with a JMX password, then a username and password must be passed into the script.
- [u] – Specify the JMX username.
- [w] – Specify the JMX password.

Stopping an SSL-Secured Server

Stop a server in a secure Terracotta cluster using the stop-tc-server script with the following arguments:

- `-f <tc-config-file>` — A valid path to the self-signed certificate must have been specified in the server's configuration file.
- `-u <username>` — The user specified must have the "admin" role.
- `-w <password>`

For more information, refer to Setting Up Server Security.

Server Status (server-stat)

The status tool is a command-line utility for checking the current status of one or more Terracotta server instances.

Microsoft Windows

```
[PROMPT] %BIGMEMORY_HOME%\server\bin\server-stat.bat <args>
```

UNIX/Linux

```
[PROMPT] ${BIGMEMORY_HOME}/server/bin/server-stat.sh <args>
```

where `<args>` are:

- [-s] host1,host2,... – Check one or more servers using the given hostnames or IP addresses using the default JMX port (9520).
- [-s] host1:9520,host2:9521,... – Check one or more servers using the given hostnames or IP addresses with JMX port specified.
- [-f] <path>/tc-config.xml – Check the servers defined in the specified configuration file.

The status tool returns the following data on each server it queries:

- **Health** – OK (server responding normally) or FAILED (connection failed or server not responding correctly).
- **Role** – The server's position in an active-mirror group. Single servers always show ACTIVE. Backups are shown as MIRROR or PASSIVE.
- **State** – The work state that the server is in. When ready, active servers should show ACTIVE-COORDINATOR, while mirror servers should show MIRROR-STANDBY or PASSIVE-STANDBY.

- **JMX port** – The TCP port the server is using to listen for JMX events.
- **Error** – If the status tool fails, the type of error.

Example

The following example shows usage of and output from the status tool.

```
[PROMPT] server-stat.sh -s myhost:9521

 localhost.health: OK
 localhost.role: ACTIVE
 localhost.state: ACTIVE-COORDINATOR
 localhost.jmxport: 9521
```

If no server is specified, by default the tool checks the status of localhost at JMX port 9520.

Version Utility (version)

The version tool is a utility script that outputs information about the BigMemory installation, including the version, date, and version-control change number from which the installation was created. When contacting Terracotta with a support query, please include the output from the version tool to expedite the resolution of your issue.

Microsoft Windows

```
[PROMPT] %BIGMEMORY_HOME%\server\bin\version.bat
```

UNIX/Linux

```
[PROMPT] ${BIGMEMORY_HOME}/server/bin/version.sh&
```

Use the following flags to produce more information:

- [r] – Produces detailed, raw information in a "property=value" format.
- [v] – Produces more detailed information.

Terracotta Maven Plugin

The Terracotta Maven Plugin allows you to use Maven to install, integrate, update, run, and test your application with Terracotta.

The Terracotta Maven Plugin, along with more documentation, is available from the Terracotta Forge.

Management and Monitoring using JMX

Introduction

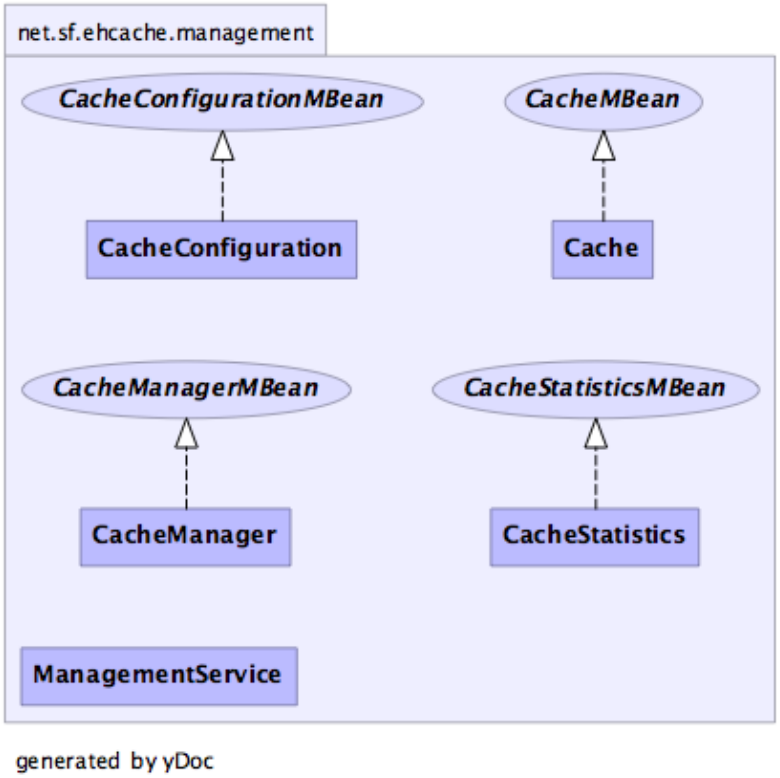
JMX creates a standard way of instrumenting classes and making them available to a management and monitoring infrastructure. This provides an alternative to the Terracotta Management Server for custom or third-party tools.

JMX Overview

The `net.sf.ehcache.management` package contains MBeans and a `ManagementService` for JMX management of BigMemory Max. It is in a separate package so that JMX libraries are only required if you want to use it - there is no leakage of JMX dependencies into the core Ehcache package.

Use `net.sf.ehcache.management.ManagementService.registerMBeans(...)` static method to register a selection of MBeans to the MBeanServer provided to the method. If you wish to monitor Ehcache but not use JMX, use the existing public methods on `Cache` and `CacheStatistics`.

The Management package is illustrated in the follwing image.



MBeans

BigMemory Max supports Standard MBeans. MBeans are available for the following:

- CacheManager
- Cache
- CacheConfiguration
- CacheStatistics

All MBean attributes are available to a local MBeanServer. The CacheManager MBean allows traversal to its collection of Cache MBeans. Each Cache MBean likewise allows traversal to its CacheConfiguration MBean and its CacheStatistics MBean.

JMX Remoting

The Remote API allows connection from a remote JMX Agent to an MBeanServer via an `MBeanServerConnection`. Only `Serializable` attributes are available remotely. The following Ehcache MBean attributes are available remotely:

- limited CacheManager attributes
- limited Cache attributes
- all CacheConfiguration attributes
- all CacheStatistics attributes

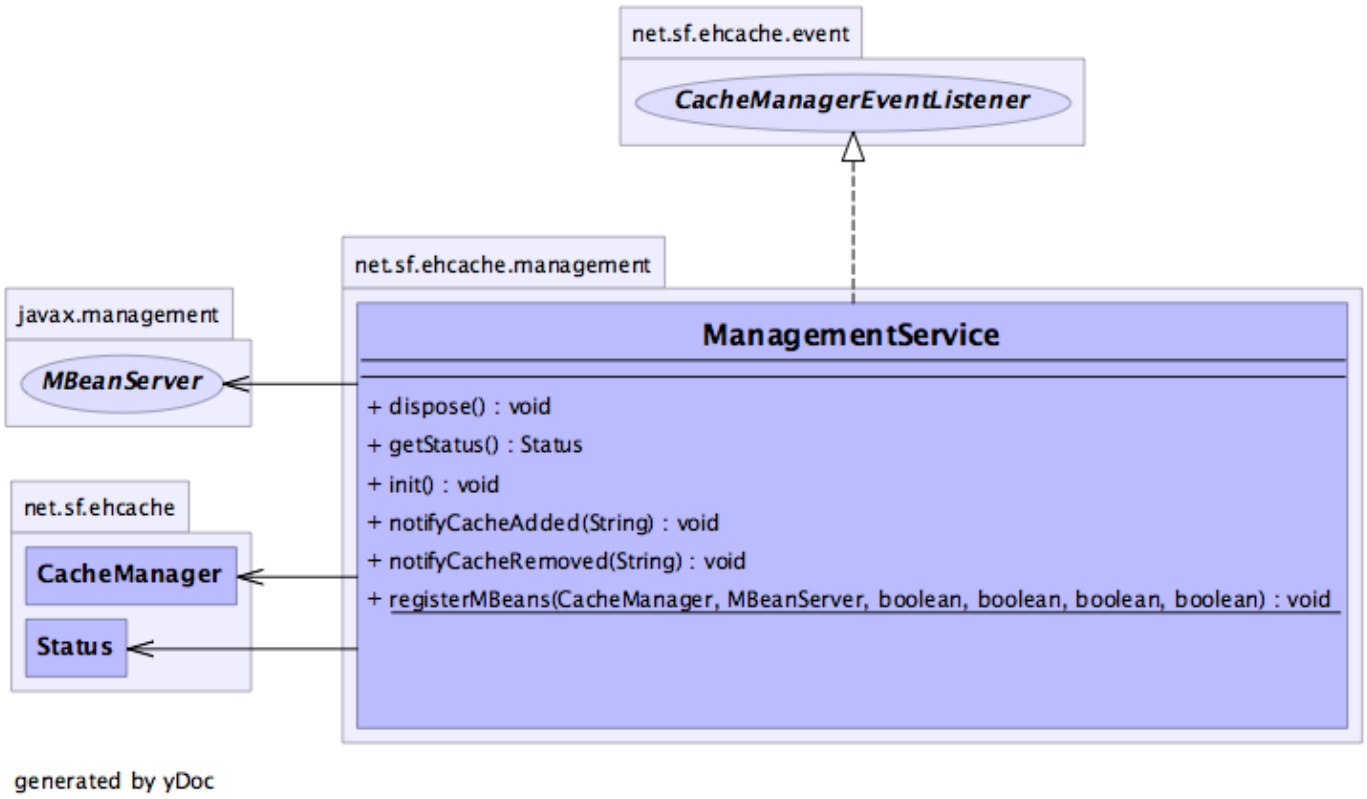
Most attributes use built-in types. To access all attributes, add ehcache.jar to the remote JMX client's classpath. For example, `jconsole -J-Djava.class.path=ehcache.jar`.

`ObjectName` naming scheme

- CacheManager - "net.sf.ehcache:type=CacheManager,name=<CacheManager>"
- Cache - "net.sf.ehcache:type=Cache,CacheManager=<cacheManagerName>,name=<cacheName>"
- CacheConfiguration
- "net.sf.ehcache:type=CacheConfiguration,CacheManager=<cacheManagerName>,name=<cacheName>"
- CacheStatistics - "net.sf.ehcache:type=CacheStatistics,CacheManager=<cacheManagerName>,name=<cacheName>"

The Management Service

The `ManagementService` class is the API entry point.



There is only one method, `ManagementService.registerMBeans` which is used to initiate JMX registration of a CacheManager's instrumented MBeans. The `ManagementService` is a `CacheManagerEventListener` and is therefore notified of any new Caches added or disposed and updates the MBeanServer appropriately. Initiated MBeans remain registered in the MBeanServer until the CacheManager shuts down, at which time the MBeans are deregistered. This ensures correct behavior in application servers where applications are deployed and undeployed.

```
/**
* This method causes the selected monitoring options to be be registered
* with the provided MBeanServer for caches in the given CacheManager.
*
* While registering the CacheManager enables traversal to all of the other
*  items,
* this requires programmatic traversal. The other options allow entry points closer
* to an item of interest and are more accessible from JMX management tools like JConsole.
* Moreover CacheManager and Cache are not serializable, so remote monitoring is not
* possible * for CacheManager or Cache, while CacheStatistics and CacheConfiguration are.
* Finally * CacheManager and Cache enable management operations to be performed.
*
* Once monitoring is enabled caches will automatically added and removed from the
* MBeanServer * as they are added and disposed of from the CacheManager. When the
* CacheManager itself * shutsdown all registered MBeans will be unregistered.
*
* @param cacheManager the CacheManager to listen to
* @param mBeanServer the MBeanServer to register MBeans to
* @param registerCacheManager Whether to register the CacheManager MBean
* @param registerCaches Whether to register the Cache MBeans
* @param registerCacheConfigurations Whether to register the CacheConfiguration MBeans
* @param registerCacheStatistics Whether to register the CacheStatistics MBeans
*/
public static void registerMBeans(
```

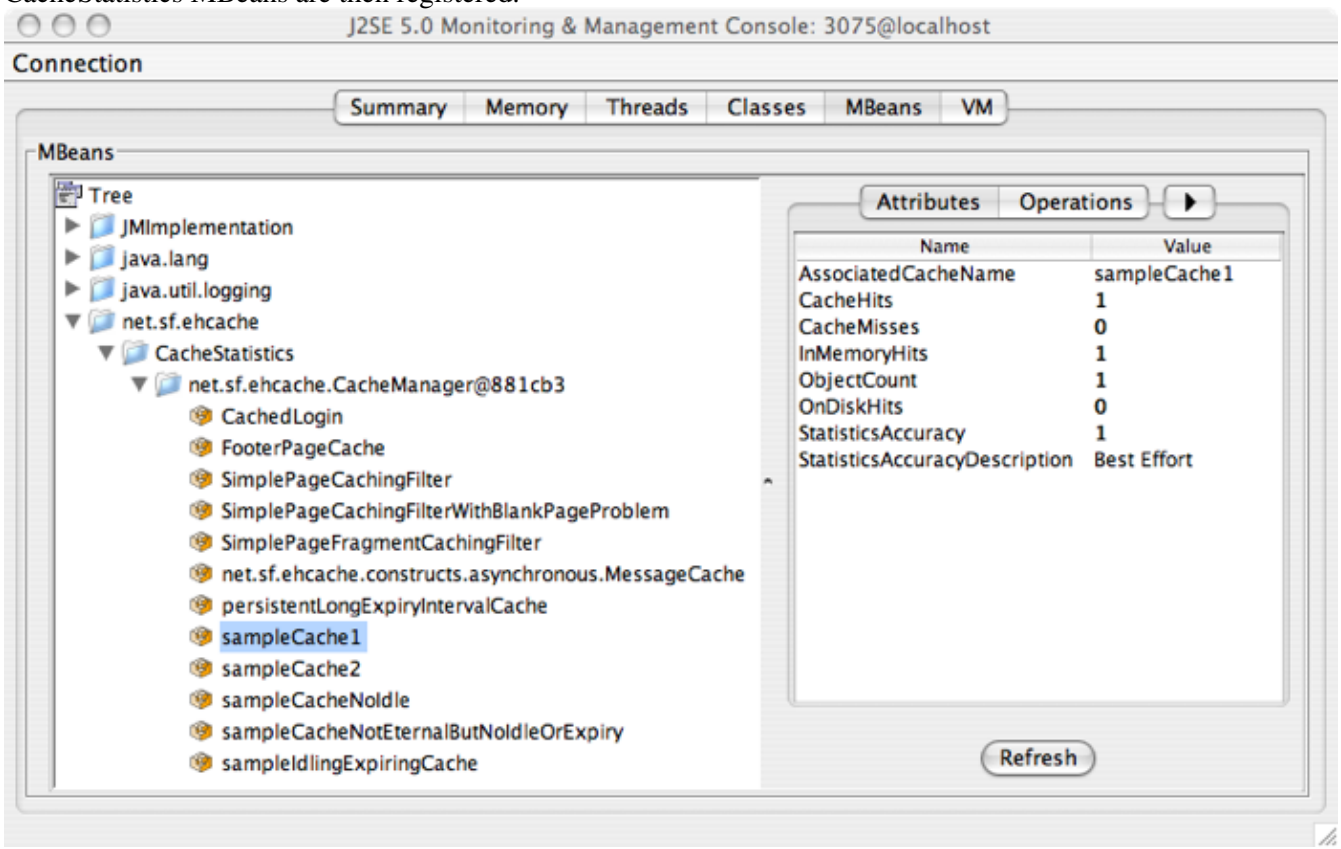
```
    net.sf.ehcache.CacheManager cacheManager,
    MBeanServer mBeanServer,
    boolean registerCacheManager,
    boolean registerCaches,
    boolean registerCacheConfigurations,
    boolean registerCacheStatistics) throws CacheException {
```

JConsole Example

This example shows how to register CacheStatistics in the JDK platform MBeanServer, which works with the JConsole management agent.

```
CacheManager manager = new CacheManager();
MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
ManagementService.registerMBeans(manager, mBeanServer, false, false, false, true);
```

CacheStatistics MBeans are then registered.



CacheStatistics MBeans in JConsole

Hibernate statistics

If you are running Terracotta clustered caches as hibernate second-level cache provider, it is possible to access the hibernate statistics and Ehcache stats via JMX. `EhcacheHibernateMBean` is the main interface that exposes all the APIs via JMX. It basically extends two interfaces -- `EhcacheStats` and `HibernateStats`. Please look into the specific interface for more details. You may also refer to this online tutorial.

Performance

Collection of cache statistics is not entirely free of overhead, however, the statistics API switches on/off automatically according to usage. If you need few statistics, you incur little overhead; on the other hand, as you use more statistics, you can incur more. Statistics are off by default.

SSL-Secured JMX Monitoring

This page documents setting up an SSL-enabled connection for remote monitoring of a Terracotta Server from a simple Java client using Java Management Extensions (JMX) technology.

Before creating this connection, be sure to complete the instructions on the Securing Terracotta Clusters with SSL page.

Compile the Client

Use the sample client code below, but adapt the host, port, username, and password variables according to your setup.

```
import java.util.HashMap;
 import java.util.Map;

 import javax.management.remote.JMXConnector;
 import javax.management.remote.JMXConnectorFactory;
 import javax.management.remote.JMXServiceURL;
 import javax.management.remote.rmi.RMIConnectorServer;
 import javax.rmi.ssl.SslRMIClientSocketFactory;
 import javax.rmi.ssl.SslRMIServerSocketFactory;

 public class Main {
   public static void main(String[] args) throws Exception {
     String host = "terracotta-server-host";
     String port = "9520";
     String username = "terracotta";
     String password = "terracotta-user-password";

     Object[] credentials = { username, password.toCharArray() };

     SslRMIClientSocketFactory csf = new SslRMIClientSocketFactory();
     SslRMIServerSocketFactory ssf = new SslRMIServerSocketFactory();

     Map<String, Object> env = new HashMap<String, Object>();
     env.put(RMIConnectorServer.RMI_CLIENT_SOCKET_FACTORY_ATTRIBUTE, csf);
     env.put(RMIConnectorServer.RMI_SERVER_SOCKET_FACTORY_ATTRIBUTE, ssf);
     env.put("com.sun.jndi.rmi.factory.socket", csf);
     env.put("jmx.remote.credentials", credentials);

     JMXServiceURL serviceURL = new JMXServiceURL("service:jmx:rmi://" + host + ":" + port +
         "/jndi/rmi://" + host + ":" + port + "/jmxrmi");
     JMXConnector jmxConnector = JMXConnectorFactory.connect(serviceURL, env);

     // do some work with the JMXConnector

     jmxConnector.close();
   }
 }
```

Run the Client

After compiling your client, configure the JVM with a truststore containing your Terracotta Server's certificate. You can simply re-use the one created for the Terracotta Server (refer to Securing Terracotta Clusters with SSL).

```
% java -Djavax.net.ssl.trustStore=/your/path/to/truststore.jks \
   -Djavax.net.ssl.trustStorePassword=your_truststore_password \
   Main
```

About the Credentials

In the above example, the client's credentials are encoded as an array of Objects. The Object array contains the username as a String in the array's first slot, and the password as a char[] in the array's second slot. The Object array is then passed to the connection as the "jmx.remote.credentials" entry. Passing the credentials in this format is necessary to avoid an authentication failure, except for the following exception. If you are using the JConsole tool, the credentials are sent as String[]{String,String} instead of String[]{String,char[]}.

Troubleshooting

Password stack trace

The stack trace below indicates that the password you specified in the `javax.net.ssl.trustStorePassword` system property is not the same as in the truststore you specified in the `javax.net.ssl.trustStore` system property.

```
Exception in thread "main" java.io.IOException: Failed to retrieve RMIServer stub: javax.namin
     java.net.SocketException: java.security.NoSuchAlgorithmException: Error constructing impl
     at javax.management.remote.rmi.RMIConnector.connect(RMIConnector.java:338)
     at javax.management.remote.JMXConnectorFactory.connect(JMXConnectorFactory.java:248)
     at Main.main(Main.java:31)
     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
     at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
     at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
     at java.lang.reflect.Method.invoke(Method.java:597)
     at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
 Caused by: javax.naming.CommunicationException [Root exception is java.rmi.ConnectIOException
     java.net.SocketException: java.security.NoSuchAlgorithmException: Error constructing impl
     at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:101)
     at com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:185)
     at javax.naming.InitialContext.lookup(InitialContext.java:392)
     at javax.management.remote.rmi.RMIConnector.findRMIServerJNDI(RMIConnector.java:1886)
     at javax.management.remote.rmi.RMIConnector.findRMIServer(RMIConnector.java:1856)
     at javax.management.remote.rmi.RMIConnector.connect(RMIConnector.java:255)
     ... 7 more
 Caused by: java.rmi.ConnectIOException: Exception creating connection to: localhost; nested e
     java.net.SocketException: java.security.NoSuchAlgorithmException: Error constructing impl
     at sun.rmi.transport.tcp.TCPEndpoint.newSocket(TCPEndpoint.java:614)
     at sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:198)
     at sun.rmi.transport.tcp.TCPChannel.newConnection(TCPChannel.java:184)
     at sun.rmi.server.UnicastRef.newCall(UnicastRef.java:322)
     at sun.rmi.registry.RegistryImpl_Stub.lookup(Unknown Source)
     at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:97)
     ... 12 more
 Caused by: java.net.SocketException: java.security.NoSuchAlgorithmException: Error constructi
     at javax.net.ssl.DefaultSSLSocketFactory.throwException(SSLSocketFactory.java:179)
```

```
        at javax.net.ssl.DefaultSSLSocketFactory.createSocket(SSLSocketFactory.java:192)
        at javax.rmi.ssl.SslRMIClientSocketFactory.createSocket(SslRMIClientSocketFactory.java:10
        at sun.rmi.transport.tcp.TCPEndpoint.newSocket(TCPEndpoint.java:595)
        ... 17 more
    Caused by: java.security.NoSuchAlgorithmException: Error constructing implementation (algorit
        at java.security.Provider$Service.newInstance(Provider.java:1245)
        at sun.security.jca.GetInstance.getInstance(GetInstance.java:220)
        at sun.security.jca.GetInstance.getInstance(GetInstance.java:147)
        at javax.net.ssl.SSLContext.getInstance(SSLContext.java:125)
        at javax.net.ssl.SSLContext.getDefault(SSLContext.java:68)
        at javax.net.ssl.SSLSocketFactory.getDefault(SSLSocketFactory.java:102)
        at javax.rmi.ssl.SslRMIClientSocketFactory.getDefaultClientSocketFactory(SslRMIClientSock
        at javax.rmi.ssl.SslRMIClientSocketFactory.createSocket(SslRMIClientSocketFactory.java:10
        ... 18 more
    Caused by: java.io.IOException: Keystore was tampered with, or password was incorrect
        at sun.security.provider.JavaKeyStore.engineLoad(JavaKeyStore.java:771)
        at sun.security.provider.JavaKeyStore$JKS.engineLoad(JavaKeyStore.java:38)
        at java.security.KeyStore.load(KeyStore.java:1185)
        at com.sun.net.ssl.internal.ssl.TrustManagerFactoryImpl.getCacertsKeyStore(TrustManagerFa
        at com.sun.net.ssl.internal.ssl.DefaultSSLContextImpl.getDefaultTrustManager(DefaultSSLCo
        at com.sun.net.ssl.internal.ssl.DefaultSSLContextImpl.<init>(DefaultSSLContextImpl.java:4
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.ja
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccesso
        at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
        at java.lang.Class.newInstance0(Class.java:357)
        at java.lang.Class.newInstance(Class.java:310)
        at java.security.Provider$Service.newInstance(Provider.java:1221)
        ... 25 more
    Caused by: java.security.UnrecoverableKeyException: Password verification failed
        at sun.security.provider.JavaKeyStore.engineLoad(JavaKeyStore.java:769)
        ... 37 more
```

Truststore stack trace

The stack trace below indicates that the truststore you specified in the `javax.net.ssl.trustStore` system property does not exist or cannot be read.

```
   Exception in thread "main" java.io.IOException: Failed to retrieve RMIServer stub: javax.namin
        javax.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.security.I
        at javax.management.remote.rmi.RMIConnector.connect(RMIConnector.java:338)
        at javax.management.remote.JMXConnectorFactory.connect(JMXConnectorFactory.java:248)
        at Main.main(Main.java:31)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
    Caused by: javax.naming.CommunicationException [Root exception is java.rmi.ConnectIOException
        javax.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.security.I
        at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:101)
        at com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:185)
        at javax.naming.InitialContext.lookup(InitialContext.java:392)
        at javax.management.remote.rmi.RMIConnector.findRMIServerJNDI(RMIConnector.java:1886)
        at javax.management.remote.rmi.RMIConnector.findRMIServer(RMIConnector.java:1856)
        at javax.management.remote.rmi.RMIConnector.connect(RMIConnector.java:255)
        ... 7 more
    Caused by: java.rmi.ConnectIOException: error during JRMP connection establishment; nested ex
        javax.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.security.I
        at sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:286)
        at sun.rmi.transport.tcp.TCPChannel.newConnection(TCPChannel.java:184)
```

```
        at sun.rmi.server.UnicastRef.newCall(UnicastRef.java:322)
        at sun.rmi.registry.RegistryImpl_Stub.lookup(Unknown Source)
        at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:97)
        ... 12 more
    Caused by: javax.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.sec
        at com.sun.net.ssl.internal.ssl.Alerts.getSSLException(Alerts.java:190)
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.fatal(SSLSocketImpl.java:1747)
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.fatal(SSLSocketImpl.java:1708)
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.handleException(SSLSocketImpl.java:1691)
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.handleException(SSLSocketImpl.java:1617)
        at com.sun.net.ssl.internal.ssl.AppOutputStream.write(AppOutputStream.java:105)
        at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:65)
        at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:123)
        at java.io.DataOutputStream.flush(DataOutputStream.java:106)
        at sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:211)
        ... 16 more
    Caused by: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParame
        at sun.security.validator.PKIXValidator.<init>(PKIXValidator.java:57)
        at sun.security.validator.Validator.getInstance(Validator.java:161)
        at com.sun.net.ssl.internal.ssl.X509TrustManagerImpl.getValidator(X509TrustManagerImpl.ja
        at com.sun.net.ssl.internal.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerI
        at com.sun.net.ssl.internal.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerI
        at com.sun.net.ssl.internal.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:
        at com.sun.net.ssl.internal.ssl.ClientHandshaker.processMessage(ClientHandshaker.java:135
        at com.sun.net.ssl.internal.ssl.Handshaker.processLoop(Handshaker.java:593)
        at com.sun.net.ssl.internal.ssl.Handshaker.process_record(Handshaker.java:529)
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.readRecord(SSLSocketImpl.java:943)
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.performInitialHandshake(SSLSocketImpl.java:
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.writeRecord(SSLSocketImpl.java:654)
        at com.sun.net.ssl.internal.ssl.AppOutputStream.write(AppOutputStream.java:100)
        ... 20 more
    Caused by: java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must
        at java.security.cert.PKIXParameters.setTrustAnchors(PKIXParameters.java:183)
        at java.security.cert.PKIXParameters.<init>(PKIXParameters.java:103)
        at java.security.cert.PKIXBuilderParameters.<init>(PKIXBuilderParameters.java:87)
        at sun.security.validator.PKIXValidator.<init>(PKIXValidator.java:55)
        ... 32 more
```

Conditional stack trace

The stack trace below indicates that one of the following conditions is true:

- The username you specified does not exist.
- The password you specified is incorrect.
- The credentials field is not an Object array containing the username as a String in the array's first slot and the password as a char[] in the array's second slot.

```
Exception in thread "main" java.lang.SecurityException: Username and/or password is not va
    at com.tc.management.EnterpriseL2Management$1.authenticate(EnterpriseL2Management.java
    at javax.management.remote.rmi.RMIServerImpl.doNewClient(RMIServerImpl.java:213)
    at javax.management.remote.rmi.RMIServerImpl.newClient(RMIServerImpl.java:180)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:2
    at java.lang.reflect.Method.invoke(Method.java:597)
    at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:303)
    at sun.rmi.transport.Transport$1.run(Transport.java:159)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.rmi.transport.Transport.serviceCall(Transport.java:155)
    at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:535)
```

```
at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0(TCPTransport.java:790)
at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:649)
at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:895)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:918)
at java.lang.Thread.run(Thread.java:680)
at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall.jav
at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:233)
at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:142)
at javax.management.remote.rmi.RMIServerImpl_Stub.newClient(Unknown Source)
at javax.management.remote.rmi.RMIConnector.getConnection(RMIConnector.java:2327)
at javax.management.remote.rmi.RMIConnector.connect(RMIConnector.java:277)
at javax.management.remote.JMXConnectorFactory.connect(JMXConnectorFactory.java:248)
at Main.main(Main.java:31)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:2
at java.lang.reflect.Method.invoke(Method.java:597)
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
```

Logging

Introduction

BigMemory Max uses the SLF4J logging facade, so you can plug in your own logging framework. This page covers Ehcache logging. For more information about slf4j in general, refer to the SLF4J site.

SLF4J Logging

With SLF4J, users must choose a concrete logging implementation at deploy time. The options include Maven and the download kit.

Concrete Logging Implementation Use in Maven

The maven dependency declarations are reproduced here for convenience. Add *one* of these to your Maven POM.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jdk14</artifactId>
    <version>1.5.8</version>
</dependency>
 <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.5.8</version>
</dependency>
 <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.5.8</version>
</dependency>
```

Concrete Logging Implemenation Use in the Download Kit

The slf4j-api and slf4j-jdk14 jars are in the kit along with the BigMemory Max jars so that, if the app does not already use SLF4J, you have everything you need. Additional concrete logging implementations can be downloaded from SLF4J website.

Recommended Logging Levels

BigMemory Max seeks to trade off informing production support developers or important messages and cluttering the log. ERROR messages should not occur in normal production and indicate that action should be taken.

WARN messages generally indicate a configuration change should be made or an unusual event has occurred. DEBUG and TRACE messages are for development use. All DEBUG level statements are surrounded with a guard so that no performance cost is incurred unless the logging level is set. Setting the logging level to DEBUG should provide more information on the source of any problems. Many logging systems enable a logging level change to be made without restarting the application.

Shutting Down BigMemory

Introduction

BigMemory is shut down through the Ehcache API. Note that Hibernate automatically shuts down its Ehcache `CacheManager`.

The recommended way to shutdown the Ehcache is:

- to call `CacheManager.shutdown()`
- in a web app, register the Ehcache `ShutdownListener`

Note that when the CacheManager is shut down, existing cache data is removed locally but may still remain in the TSA (with distributed caches) or disk (if restartable). This is the same as occurs when only the cache is removed (`Cache.dispose()` or `CacheManager.removeCache()`). To ensure that unwanted data is not persisted, call `Cache.removeAll()` in all caches whose data is no longer wanted.

Explicitly Removing Data

Specific entries can be removed from a cache using `Cache.remove()`. To empty the cache, `Cache.removeAll()`. If the cache itself is removed (`Cache.dispose()` or `CacheManager.removeCache()`), then any data still remaining in the cache is also removed locally. However, that remaining data is *not* removed from the TSA or disk (if restartable).

Though not recommended, Ehcache also lets you register a JVM shutdown hook.

ServletContextListener

Ehcache provides a ServletContextListener that shuts down the CacheManager. Use this to shut down Ehcache automatically, when the web application is shut down. To receive notification events, this class must be configured in the deployment descriptor for the web application. To do so, add the following to web.xml in your web application:

```
<listener>
  <listener-class>net.sf.ehcache.constructs.web.ShutdownListener</listener-class>
</listener>
```

The Shutdown Hook

The Ehcache CacheManager can optionally register a shutdown hook. To do so, set the system property `net.sf.ehcache.enableShutdownHook=true`. This will shut down the CacheManager when it detects the Virtual Machine shutting down and it is not already shut down.

Use the shutdown hook when the CacheManager is not already being shutdown by a framework you are using, or by your application.

Note: Shutdown hooks are inherently problematic. The JVM is shutting down, so sometimes things that can never be null are. Ehcache guards against as many of these as it can, but the shutdown hook should be the last

option to use.

The shutdown hook is on CacheManager. It simply calls the shutdown method. The sequence of events is:

- call dispose for each registered CacheManager event listener.
- call dispose for each Cache. Each Cache will:
- shutdown the MemoryStore. The MemoryStore will flush to the DiskStore.
- shutdown the DiskStore. If the DiskStore is persistent ("localRestartable"), it will write the entries and index to disk.
- shutdown each registered CacheEventListener.
- set the Cache status to shutdown, preventing any further operations on it.
- set the CacheManager status to shutdown, preventing any further operations on it.

The shutdown hook runs when:

- A program exists normally. For example, `System.exit()` is called, or the last non-daemon thread exits.
- the Virtual Machine is terminated, e.g. CTRL-C. This corresponds to `kill -SIGTERM pid` or `kill -15 pid` on Unix systems.

The shutdown hook will not run when:

- the Virtual Machine aborts.
- A SIGKILL signal is sent to the Virtual Machine process on Unix systems, e.g. `kill -SIGKILL pid` or `kill -9 pid`.
- A `TerminateProcess` call is sent to the process on Windows systems.

Dirty Shutdown

If Ehcache is shutdown dirty, all in-memory data will be retained if BigMemory is configured for restartability. For more information, refer to Fast Restartability.

BigMemory Max Best Practices

The following sections contain advice for optimizing BigMemory Max operations.

Tuning Off-Heap Store Performance

Memory-related or performance issues that arise during operations can be related to improper allocation of memory to the off-heap store. If performance or functional issues arise that can be traced back to the off-heap store, see the suggested tuning tips in this section.

General Memory allocation

Committing too much of a system's physical memory is likely to result in paging of virtual memory to disk, quite likely during garbage-collection operations, leading to significant performance issues. On systems with multiple Java processes, or multiple processes in general, the sum of the Java heaps and off-heap stores for those processes should also not exceed the size of the physical RAM in the system. Besides memory allocated to the heap, Java processes require memory for other items, such as code (classes), stacks, and PermGen.

Note that MaxDirectMemorySize sets an upper limit for the JVM to enforce, but does not actually allocate the specified memory. Overallocation of direct memory (or buffer) space is therefore possible, and could lead to paging or even memory-related errors. The limit on direct buffer space set by MaxDirectMemorySize should take into account the total physical memory available, the amount of memory that is allotted to the JVM object heap, and the portion of direct buffer space that other Java processes may consume.

In addition, be sure to allocate at least 15 percent more off-heap memory than the size of your data set. To maximize performance, a portion of off-heap memory is reserved for meta-data and other purposes.

Note also that there could be other users of direct buffers (such as NIO and certain frameworks and containers). Consider allocating additional direct buffer memory to account for that additional usage.

Compressed References

For 64-bit JVMs running Java 6 Update 14 or higher, consider enabling compressed references to improve overall performance. For heaps up to 32GB, this feature causes references to be stored at half the size, as if the JVM is running in 32-bit mode, freeing substantial amounts of heap for memory-intensive applications. The JVM, however, remains in 64-bit mode, retaining the advantages of that mode.

For the Oracle HotSpot, compressed references are enabled using the option `-XX:+UseCompressedOops`. For IBM JVMs, use `-Xcompressedrefs`.

Slow Off-Heap Allocation

Based on configuration, usage, and memory requirements, BigMemory could allocate off-heap memory multiple times. If off-heap memory comes under pressure due to over-allocation, the host OS may begin paging to disk, thus slowing down allocation operations. As the situation worsens, an off-heap buffer too large to fit in memory can quickly deplete critical system resources such as RAM and swap space and crash the host OS.

To stop this situation from degrading, off-heap allocation time is measured to avoid allocating buffers too large to fit in memory. If it takes more than 1.5 seconds to allocate a buffer, a warning is issued. If it takes more than 15 seconds, the JVM is halted with `System.exit()` (or a different method if the Security Manager prevents this).

To prevent a JVM shutdown after a 15-second delay has occurred, set the `net.sf.ehcache.offheap.DoNotHaltOnCriticalAllocationDelay` system property to true. In this case, an error is logged instead.

Maximum Serialized Size of an Element

This section applies when using BigMemory through the Ehcache API.

Unlike the memory and the disk stores, by default the off-heap store has a 4MB limit for classes with high quality hashcodes, and 256KB limit for those with pathologically bad hashcodes. The built-in classes such as `String` and the `java.lang.Number` subclasses Long and Integer have high quality hashcodes. This can issues when objects are expected to be larger than the default limits.

To override the default size limits, set the system property `net.sf.ehcache.offheap.cache_name.config.idealMaxSegmentSize` to the size you require.

For example,

```
net.sf.ehcache.offheap.com.company.domain.State.config.idealMaxSegmentSize=30M
```

Reducing Faulting

While the memory store holds a hotset (a subset) of the entire data set, the off-heap store should be large enough to hold the entire data set. The frequency of misses (get operations that fail to find the data in memory) begins to rise when the data is too large to fit into off-heap memory, forcing gets to fetch data from the disk store (called *faulting*). More misses in turn raise latency and lower performance.

For example, tests with a 4GB data set and a 5GB off-heap store recorded no misses. With the off-heap store reduced to 4GB, 1.7 percent of cache operations resulted in misses. With the off-heap store at 3GB, misses reached 15 percent.

Swappiness and Huge Pages

An operating system (OS) that is swapping to disk can substantially slow down or even stop your application. If the OS is under pressure because Terracotta servers—along with other processes running on a host—are squeezing the available memory, then memory will start to be paged in and out. This type of operation, when too frequent, requires either tuning of the swap parameters or a permanent solution to a chronic lack of RAM.

An OS could swap data from memory to disk even if memory is not running low. For the purpose of optimization, data that appears to be unused may be a target for swapping. Because BigMemory can store substantial amounts of data in RAM, its data may be swapped by the OS. But swapping can degrade overall cluster performance by introducing thrashing, the condition where data is frequently moved forth and back between memory and disk.

To make heap memory use more efficient, Linux, Microsoft Windows, and Oracle Solaris users should review their configuration and usage of swappiness as well as the size of the swapped memory pages. In general, BigMemory benefits from lowered swappiness and the use of *huge pages* (also known as *big pages*, *large pages*, and *superpages*). Settings for these behaviors vary by OS and JVM. For Oracle HotSpot, `-XX:+UseLargePages` and `-XX:LargePageSizeInBytes=<size>` (where <size> is a value allowed by the OS for specific CPUs) can be used to control page size. However, note that this setting does not affect how off-heap memory is allocated. Over-allocating huge pages while also configuring substantial off-heap memory *can starve off-heap allocation and lead to memory and performance problems*.

Reduce Swapping

Many tools are available to help you diagnose swapping. Popular options include using a built-in command-line utility. On Linux, for example:

- See available RAM with `free -m` (display memory statistics in megabtyes). Pay attention to swap utilization.
- `vmstat` displays swap-in ("si") and swap-out ("so") numbers. Non-zero values indicate swapping activity. Set `vmstat` to refresh on a short interval to detect trends.
- Process status can be used to get detailed information on all processes running on a node. For example, `ps -eo pid,ppid,rss,vsize,pcpu,pmem,cmd -ww --sort=pmem` displays processes ordered by memory use. You can also sort by virtual memory size ("vsize") and real memory size ("rss") to focus on both the most memory-consuming processes and their in-memory footprint.

NOTE: If the JVM is running in a guest virtual host, analyze swapping by both the virtual and underlying host.

If swappiness is being caused by memory pressure, offloading unnecessary or unrelated processes along with running smaller JVMs is often a successful cure. When computing the memory footprint of a JVM, be sure to include the off-heap being allocated.

Tuning Heap Memory Performance

Long garbage collection (GC) cycles are one of the most common causes of issues in a Terracotta cluster because a full GC event pauses all threads in the JVM. Servers disconnecting clients, clients dropping servers, and timed-out processes are just some of the problems long GC cycles can cause. Having a clear understanding of how your application behaves with respect to creating garbage, and how that garbage is being collected, is necessary for avoiding or solving these issues.

Printing and Analyzing GC Logs

The most effective way to gain that understanding is to create a profile of GC in your application by using tools made for that purpose. Consider using JVM options to generate logs of GC activity:

- `-verbose:gc`
- `-Xloggc:<filename>`
- `-XX:+PrintGCDetails`
- `-XX:+PrintGCTimeStamps`

Apply an appropriate parsing and visualization tool to GC log files to help analyze their contents.

Observing GC Statistics With jstat

One way to observe GC statistics is by using the Java utility jstat. The following command will produce a log of GC statistics, updated every ten seconds:

```
jstat -gcutil <pid> 10 1000000
```

An important statistic is the Full Garbage Collection Time. The difference between the total time for each reading is the amount of time the system was paused. A jump of more than a few seconds will not be acceptable in most application contexts.

Solutions to Problematic GC

Once your application's typical GC cycles are understood, consider one or more of the following solutions:

- Maximizing BigMemory to eliminate the drag GC imposes on performance in large heaps.

  BigMemory opens up off-heap memory for use by Java applications, and off-heap memory is not subject to GC.
- Configuring the HealthChecker parameters in the Terracotta cluster to account for the observed GC cycles.

  Increase nodes' tolerance of inactivity in other nodes due to GC cycles.
- Tuning the GC parameters to change the way GC runs in the heap.

  If running multi-core machines and no collector is specifically configured, consider `-XX:+UseParallelGC` and `-XX:+UseParallelOldGC`.

  If running multiple JVMs or application processes on the same machine, tune the number of concurrent threads in the parallel collector with `-XX:ParallelGCThreads=<number>`.

  Another collector is called Concurrent Mark Sweep (CMS). This collector is normally not recommended (especially for Terracotta servers) due to certain performance and operational issues it raises. However, under certain circumstances related to the type of hosting platform and application data usage characteristics, it may boost performance and may be worth testing with.
- If running on a 64-bit JVM, and if your JDK supports it, use `-XX:+UseCompressedOops`.

  This setting can reduce substantially the memory footprint of object pointer used by the JVM.
- Refactoring clustered applications that unnecessarily create too much garbage.
- Ensuring that the problem node has enough memory allocated to the heap.

Common Causes of Failures in a Cluster

The most common causes of failures in a cluster are interruptions in the network and long Java GC cycles on particular nodes. Tuning the HealthChecker and reconnect features can reduce or eliminate these two problems. However, additional actions should also be considered.

Sporadic disruptions in network connections between L2s and between L2s and L1s can be difficult to track down. Be sure to thoroughly test all network segments connecting the nodes in a cluster, and also test network hardware. Check for speed, noise, reliability, and other applications that grab bandwidth.

Other sources of failures in a cluster are disks that are nearly full or are running slowly, and running other applications that compete for a node's resources.

Do Not Interrupt!

Ensure that your application does not interrupt clustered threads. This is a common error that can cause the Terracotta client to shut down or go into an error state, after which it will have to be restarted.

The Terracotta client library runs with your application and is often involved in operations which your application is not necessarily aware of. These operations can get interrupted, something the Terracotta client cannot anticipate. Interrupting clustered threads, in effect, puts the client into a state which it cannot handle.

Diagnose Client Disconnections

If clients disconnect on a regular basis, try the following to diagnose the cause:

- Analyze the Terracotta client logs for potential issues, such as long GC cycles.
- Analyze the Terracotta server logs for disconnection information and any rejections of reconnection attempts by the client.
- See the operator events panel in the Terracotta Management Console for disconnection events, and note the reason.

If the disconnections are due to long GC cycles or inconsistent network connections in the client, consider the remedies suggested in this section. If disconnections continue to happen, consider configuring caches with nonstop behavior and enabling rejoin.

Detect Memory Pressure Using the Terracotta Logs

Terracotta server and client logs contain messages that help you track memory usage. Locations of server and client logs are configured in the Terracotta configuration file, `tc-config.xml`.

You can view the state of memory usage in a node by finding messages similar to the following:

```
2011-12-04 14:47:43,341 [Statistics Logger] ... memory free : 39.992699 MB
2011-12-04 14:47:43,341 [Statistics Logger] ... memory used : 1560.007301 MB
2011-12-04 14:47:43,341 [Statistics Logger] ... memory max : 1600.000000 MB
```

These messages can indicate that the node is running low on memory.

Disk usage with both Search and Fast Restart enabled

The TSA may be configured to be restartable in addition to including searchable caches, but both of these features require disk storage. When both are enabled, be sure that enough disk space is available. Depending upon the number of searchable attributes, the amount of disk storage required may be up to 1.5 times the amount of in-memory data.

Manage Sessions in a Cluster

- Make sure the configured time zone and system time is consistent between all application servers. If they are different a session may appear expired when accessed on different nodes.

- Set `-Dcom.tc.session.debug.sessions=true` and `-Dcom.tc.session.debug.invalidate=true` to generate more debugging information in the client logs.
- All clustered session implementations (including terracotta Sessions) require a mutated session object be put back into the session after it's mutated. If the call is missing, then the change isn't known to the cluster, only to the local node. For example:

```
Session session = request.getSession();
 Map m  = session.getAttribute("foo");
 m.clear();
 session.setAttribute("foo", m); // Without this call, the clear() is not effective across
```

Without a `setAttribute()` call, the session becomes inconsistent across the cluster. Sticky sessions can mask this issue, but as soon as the session is accessed on another node, its state does not match the expected one. To view the inconsistency on a single client node, add the Terracotta property `-Dcom.tc.session.clear.on.access=true` to force locally cached sessions to be cleared with every access.

If third-party code cannot be refactored to fix this problem, and you are running Terracotta 3.6.0 or higher, you can write a servlet filter that calls `setAttribute()` at the end of every request. Note that this solution may substantially degrade performance.

```
package controller.filter;

 import java.io.IOException;
 import java.util.Enumeration;

 import javax.servlet.Filter;
 import javax.servlet.FilterChain;
 import javax.servlet.FilterConfig;
 import javax.servlet.ServletException;
 import javax.servlet.ServletRequest;
 import javax.servlet.ServletResponse;
 import javax.servlet.http.HttpServletRequest;
 import javax.servlet.http.HttpSession;

 public class IterateFilter implements Filter {

   public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
   throws IOException, ServletException {
     HttpSession session = ((HttpServletRequest) request).getSession();
     if (session != null) {
       @SuppressWarnings("rawtypes")
       Enumeration e = session.getAttributeNames();
       while (e.hasMoreElements()) {
         String name = (String)e.nextElement();
         Object value = session.getAttribute(name);
         session.setAttribute(name, value);
       }
     }
   }

   public void init(FilterConfig filterConfig) throws ServletException {
     // TODO Auto-generated method stub
   }

   public void destroy() {
     // TODO Auto-generated method stub
```

```
    }
  }
```

A Safe Failover Procedure

To safely migrate clients to a standby server without stopping the cluster, follow these steps:

1. If it is not already running, start the standby server using the start-tc-server script. The standby server must already be configured in the Terracotta configuration file.
2. Ensure that the standby server is ready for failover (PASSIVE-STANDBY status). In the TMC, the status light will be cyan.
3. Shut down the active server using the stop-tc-server script.

   NOTE: If the script detects that the mirror server in STANDBY state isn't reachable, it issues a warning and fails to shut down the active server. If failover is not a concern, you can override this behavior with the `--force` flag.

   Clients will connect to the new active server.
4. Restart any clients that fail to reconnect to the new active server within the configured reconnection window.

The previously active server can now rejoin the cluster as a standby server. If restartable mode had been enabled, its data is first removed and then the current data is read in from the now active server.

A Safe Cluster Shutdown Procedure

A safe cluster shutdown should follow these steps:

1. Shut down the standby servers using the stop-tc-server script.
2. Shut down the clients. The Terracotta client will shut down when you shut down your application.
3. Shut down the active server using the stop-tc-server script.

To restart the cluster, first start the server that was last active. If clustered data is not persisted, any of the servers could be started first as no data conflicts can take place.

BigMemory Max FAQ

This FAQ answers questions about how to use BigMemory Max and Terracotta products, integration with other products, and solving issues. If your question doesn't appear here, consider posting it on the Terracotta forums.

Other resources for resolving issues can be found on the Release and Compatibility Information page, including:

- Release Notes – Lists features and issues for specific versions of Terracotta products.
- Compatibility Information – Includes tables on compatible versions of Terracotta products, JVMs, and application servers.

The FAQ is divided into the following sections:

Getting Started

{toc-zone|3:3}

What's the difference between BigMemory Go and BigMemory Max?

BigMemory Go is for in-memory data management on a single JVM (in-process) and comes with 32GB free. BigMemory Max is for distributed in-memory management across an array of servers. For more on Go vs. Max, see the BigMemory Overview.

What platforms does Terracotta software run on? Which application stacks does Terracotta support?

Terracotta is designed to work with as broad an array of platforms, JVMs and application server versions as possible. Supported platforms are listed in the Release and Platform Compatibility Information.

What is the Terracotta Client?

The Terracotta Client is functionality in a Java library that operates inside your JVM that enables clustering. When your JVM starts and code is called that initializes Terracotta, the Terracotta Client automatically connects to the Terracotta Server Array to engage clustering services such as the lock manager, object manager, and memory manager.

What is the Terracotta Server Array?

The Terracotta Server Array is a set of one or more processes that coordinate data sharing among all Terracotta Clients in the cluster. Each Terracotta Server Array process is a simple Java application that runs directly on the JVM (ie without an application server or container). The Terracotta Server Array is designed to provide maximal High Availability and Scalability.

What is the Terracotta Toolkit?

The Terracotta Toolkit is a powerful library designed for developers working on scalable applications, frameworks, and software tools. The Terracotta Toolkit provides a simple set of APIs, which themselves are used to create Terracotta products. Supported APIs include concurrent maps, cyclic barriers, locks, counters, and queues. The Toolkit is designed to be platform independent and runs on any JVM without requiring boot-jars, agents, or container-specific code.

{/toc-zone}

Configuration, Development, and Operations

{toc-zonel3:3}

How do I enable restartable mode?

In the servers section of your tc-config.xml, include the following configurations:

```
<servers>
   <server host="host1" name="Server1">
       <data-backup>path/to/my/backup/directory</data-backup>
       <offheap enabled="true" maxDataSize="2tb"/>
   </server>
   <restartable enabled="true"/>
</servers>
```

If persistence of shared data is not required across restarts, set `<restartable enabled>` to "false".

How do I configure failover to work properly with two Terracotta servers?

Configure both servers in the `<servers>` section of the Terracotta configuration file. Start the two Terracotta server instances that use that configuration file, one server assumes control of the cluster (the ACTIVE) and the second becomes the mirror (the PASSIVE). See the high-availability page for more information.

How do I know that my application has started up with a Terracotta client and is sharing data?

The Terracotta Management Console (TMC) displays cluster topology by listing Terracotta server groups and connected client nodes in a navigation tree.

In addition, check standard output for messages that the Terracotta client has started up without errors. Terracotta clients also log messages to a file specified in the `<clients>` section of the tc-config file.

Is there a maximum number of objects that can be held by one Terracotta server instance?

Is there a maximum number of objects that can be held by oneTerracotta server instance?

The number of objects that can be held by a Terracotta server instance is two billion, a limit imposed by the design of Java collections. It is unlikely that a Terracotta cluster will need to approach even 50 percent of that maximum. However, if it does, other issues may arise that require the rearchitecting of how application data is handled in the cluster.

How many Terracotta clients (L1s) can connect to the Terracotta Server Array (L2s) in a cluster?

While the number of L1s that can exist in a Terracotta cluster is theoretically unbounded (and cannot be configured), effectively planning for resource limitations and the size of the shared data set should yield an optimum number. Typically, the most important factors that will impact that number are the requirements for performance and availability. Typical questions when sizing a cluster:

- What is the desired transactions-per-second?
- What are the failover scenarios?
- How fast and reliable is the network and hardware? How much memory and disk space will each machine have?
- How much shared data is going to be stored in the cluster? How much of that data should be on the L1s? Will BigMemory be used?
- How many stripes (active Terracotta servers) does the cluster have?
- How much load will there be on the L1s? On the L2s?

The most important method for determining the optimum size of a cluster is to test various cluster configurations under load and observe how well each setup meets overall requirements.

What's the best way for my application to listen to Terracotta cluster events such as lost application nodes?

If you are using Ehcache, use the cluster-events Ehcache API. In general, you can use the Terracotta Toolkit API to set up cluster-events listeners.

How can my application check that the Terracotta process is alive at runtime?

Your application can check to see if the system property `tc.active` is true. For example, the following line of code would return true if Terracotta is active at the time it is run:

```
Boolean.getBoolean("tc.active");
```

How do I confirm that my Terracotta servers are up and running correctly?

Here are some ways to confirm that your Terracotta servers are running:

- Connect to the servers using the Terracotta Management Console.
- Check the standard output messages to see that each server started without errors.
- Check each server's logs to see that each server started without errors. The location of server logs is specified in `tc-config.xml`.

How do I confirm that my Terracotta servers are up and runningcorrectly?

- Use the Terracotta script `server-stat.sh` or `server-stat.bat` to generate a short status report on one or more Terracotta servers.
- Use a tool such as wget to access the /config or /version servlet. For example, for a server running on localhost and port 9889, use the following wget command to connect to the version servlet:

```
[PROMPT] wget http://localhost:9889/version
```

Are there ways I can monitor the cluster that don't involve using the Terracotta Management Console?

You can monitor the cluster using JMX. .

Cluster events are available over JMX via the object name "org.terracotta:type=TC Operator Events,name=Terracotta Operator Events Bean". Use a tool such as JConsole to view other MBeans needed for monitoring

How can I control the logging level for Terracotta servers and clients?

Create a file called `.tc.custom.log4j.properties` and edit it as a standard `log4j.properties` file to configure logging, including level, for the Terracotta node that loads it. This file is searched for in the path specified by the environment variable TC_INSTALL_DIR (if defined), `user.home`, and `user.dir`.

Why is it a bad idea to change shared data in a shutdown hook?

If a node attempts to change shared data while exiting, and the shutdown thread blocks, the node may hang and be dropped from the cluster, failing to exit as planned. The thread may block for any number of reasons, such as the failure to obtain a lock. A better alternative is to use the cluster events API to have a second node (one that is not exiting) execute certain code when it detects that the first node is exiting. If you are using Ehcache, use the cluster-events Ehcache API. In general, you can use the Terracotta Toolkit API to set up cluster-events listeners.

You can call `org.terracotta.api.Terracotta.registerBeforeShutdownHook(Runnable beforeShutDownHook)` to perform various cleanup tasks before the Terracotta client disconnects and shuts down.

Note that a Terracotta client is not required to release locks before shutting down. The Terracotta server will reclaim those locks, although any outstanding transactions are not committed.

Can I store collections inside collections that are stored as values in a cache?

This is not recommended unless your application logic takes careful account of how such values can be modified safely. Race conditions, undefined results, ConcurrentModificationException, and other problems can arise.

{/toc-zone}

Environment and Interoperability

{toc-zonel3:3}

Where is there information on platform compatibility for my version of Terracotta software?

Information on the latest releases of Terracotta products, including a link to the latest platform support, is found on the Product Information page. This page also contains a table with links to information on previous releases.

Can I run the Terracotta process as a Microsoft Windows service?

While running an application as a Microsoft Windows service has many advantages, such as scheduling and automatic start and restart, there is no official supported configuration for doing this with Terracotta software. However, there are solutions available that have been tried successfully, including some with Java Service Wrapper. This blog entry includes a short procedure, and this blog entry shows how to do the same on a 64-bit system.

Try an Internet search on "windows java service" to find other possible solutions and articles.

Do you have any advice for running Terracotta software on Ubuntu?

The known issues when trying to run Terracotta software on Ubuntu are:

- Default shell is *dash bash*. Terracotta scripts don't behave under dash. You might solve this issue by setting your default shell to bash or changing `/bin/sh` in our scripts to `/bin/bash`.
- The Ubuntu default JDK is from GNU. Terracotta software compatibility information is on the Product Information page.
- See the UnknownHostException topic below.

Which Garbage Collector should I use with the Terracotta Server (L2) process?

The Terracotta Server performs best with the default garbage collector. This is pre-configured in the startup scripts. If you believe that Java GC is causing performance degradation in the Terracotta Server, the simplest and best way to reduce latencies by reducing collection times is to store more data in BigMemory Max.

Generally, the use of the Concurrent Mark Sweep collector (CMS) is discouraged as it is known to cause heap fragmentation for certain application-data usage patterns. Expert developers considering use of CMS should consult the Oracle tuning and best-practice documentation.

Can I substitute Terracotta for JMS? How do you do messaging in Terracotta clusters?

Using Terracotta with a simple data structure (such as `java.util.concurrent.LinkedBlockingQueue`), you can easily create message queues that can replace JMS. Your particular use case should dictate whether to replace JMS or continue using it alongside

Terracotta. See the Terracotta Toolkit API for more information on using a clustered queue.

Does Terracotta clustering work with Hibernate?

Through Ehcache, you can enable and cluster Hibernate second-level caches.

What other technologies does Terracotta software work with?

Terracotta software integrates with most popular Java technologies being used today. For a full list, contact us at {$contact_email}.

{/toc-zone}

Troubleshooting

{toc-zonel3:3}

After my application interrupted a thread (or threw InterruptedException), why did the Terracotta client die?

The Terracotta client library runs with your application and is often involved in operations which your application is not necessarily aware of. These operations may get interrupted, too, which is not something the Terracotta client can anticipate. Ensure that your application does not interrput clustered threads. This is a common error that can cause the Terracotta client to shut down or go into an error state, after which it will have to be restarted.

Why does the cluster seem to be running more slowly?

There can be many reasons for a cluster that was performing well to slow down over time. The most common reason for slowdowns is Java Garbage Collection (GC) cycles. Another reason may be that near-memory-full conditions have been reached, and the TSA needs to clear space for continued operations by additional evictions. For more information, refer to Terracotta Server Array Architecture.

Another possible cause is when an active server is syncing with a mirror server. If the active is under substantial load, it may be slowed by syncing process. In addition, the syncing process itself may appear to slow down. This can happen when the mirror is waiting for specific sequenced data before it can proceed, indicated by log messages similar to the following:

```
WARN com.tc.l2.ha.L2HACoordinator - 10 messages in pending queue.
    Message with ID 2273677 is missing still
```

If the message ID in the log entries changes over time, no problems are indicated by these warnings.

Another indication that slowdowns are occurring on the server and that clients are throttling their transaction commits is the appearance of the following entry in client logs:

```
INFO com.tc.object.tx.RemoteTransactionManagerImpl - ClientID[2](: TransactionID=[65037) :
    Took more than 1000ms to add to sequencer  : 1497 ms
```

Why do all of my objects disappear when I restart the server?

If you are not running the server in restartable mode, the server will remove the object data when it restarts. If you want object data to persist across server restarts, run the server in restartable mode.

Why are old objects still there when I restart the server?

If you are running the server in restartable mode, the server keeps the object data across restarts. If you want objects to disappear when you restart the server, you can either disable restartable mode or remove the data files from disk before you restart the server. See this question.

Why can't certain nodes on my Terracotta cluster see each other on the network?

A firewall may be preventing different nodes on a cluster from seeing each other. If Terracotta clients attempt to connect to a Terracotta server, for example, but the server seems to not not have any knowledge of these attempts, the clients may be blocked by a firewall. Another example is a backup Terracotta server that comes up as the active server because it is separated from the active server by a firewall.

Client and/or server nodes are exiting regularly without reason.

Client or server processes that quit ("L1 Exiting" or "L2 Exiting" in logs) for seemingly no visible reason may have been running in a terminal session that has been terminated. The parent process must be maintained for the life of the node process, or use another workaround such as the `nohup` option.

I have a setup with one active Terracotta server instance and a number of standbys, so why am I getting errors because more than one active server comes up?

Due to network latency or load, the Terracotta server instances may not may be have adequate time to hold an election. Increase the `<election-time>` property in the Terracotta configuration file to the lowest value that solves this issue.

If you are running on Ubuntu, see the note at the end of the UnknownHostException topic below.

I have a cluster with more than one stripe (more than one active Terracotta server) but data is distributed very unevenly between the two stripes.

The Terracotta Server Array distributes data based on the hashcode of keys. To enhance performance, each server stripe should contain approximately the same amount of data. A grossly uneven distribution of data on Terracotta servers in a cluster with more than one active server can be an indication that keys are not being hashed well. If your application is creating keys of a type that does not hash well, this may be the cause of the uneven distribution.

Why is a crashed Terracotta server instance failing to come up when I restart it?

If running in retartable mode, the ACTIVE Terracotta server instance should come up with all shared data intact. However, if the server's database has somehow become corrupt, you must clear the crashed server's data directory before restarting.

I lost some data after my entire cluster lost power and went down. How can I ensure that all data persists through a failure?

If only some data was lost, then Terracotta servers were configured to persist data. The cause for losing a small amount of data could be disk "write" caching on the machines running the Terracotta server instances. If every Terracotta server instance lost power when the cluster went down, data remaining in the disk cache of each machine is lost.

Turning off disk caching is not an optimal solution because the machines running Terracotta server instances will suffer a substantial performance degradation. A better solution is to ensure that power is never interrupted at any one time to every Terracotta server instance in the cluster. This can be achieved through techniques such as using uninterruptible power supplies and geographically subdividing cluster members.

Do I have to restart Terracotta clients after redeploying in a container?

Errors could occur if a client runs with a web application that has been redeployed, causing the client to not start properly or at all. If the web application is redeployed, be sure to restart the client.

Why does the JVM on my SPARC machines crash regularly?

You may be encountering a known issue with the Hotspot JVM for SPARC. The problem is expected to occur with Hotspot 1.6.0_08 and higher, but may have been fixed in a later version. For more information, see this bug report.

{/toc-zone}

Specific Errors and Warnings

{toc-zone|3:3}

Why, after restarting an application server, does the error Client Cannot Reconnect... repeat endlessly until the Terracotta server's database is wiped?

The default value of the client reconnection setting `l1.max.connect.retries` is set to "-1" (infinite). If you frequently encounter the situation described in this question and do not want to wipe the database and restart the cluster, change the retry setting to finite value. See the high-availability page for more information.

What does the warning "WARN com.tc.bytes.TCByteBufferFactory - Asking for a large amount of memory..." mean?

If you see this warning repeatedly, objects larger than the recommended maximum are being shared in the Terracotta cluster. These objects must be sent between clients and servers. In this case, related warnings containing text similar to **`Attempt to read a byte array of len: 12251178; threshold=8000000`** and **`Attempting to send a message (com.tc.net.protocol.delivery.OOOProtocolMessageImpl) of size`** may also appear in the logs.

If there are a large number of over-sized objects being shared, low-memory issues and degradation of performance may result.

In addition, if elements too large to fit in a client are cached, the value will be stored on the server, thus degrading performance (reads will be slower). In this case, a warning is logged.

When starting a Terracotta server, why does it throw a DBVersionMismatchException?

The server is expecting a Terracotta database with a compatible version, but is finding one with non-compatible version. This usually occurs when starting a Terracotta server with an older version of the database. Note that this can only occur with servers in the restartable mode.

Why am I getting MethodNotFound and ClassNotFound exceptions?

If you've integrated a Terracotta product with a framework such as Spring or Hibernate and are getting one of these exceptions, make sure that an older version of that Terracotta product isn't on the classpath. With Maven involved, sometimes an older version of a Terracotta product is specified in a framework's POM and ends up ahead of the current version you've specified. You can use tools such as jconsole or jvisualvm to debug, or specify `-XX:+TraceClassLoading` on the command line.

If a ClassNotFound exception is thrown at startup, check that a supported JDK (not a JRE) is installed.

When I start a Terracotta server, why does it fail with a schema error?

You may get an error similar to the following when a Terracotta server fails to start:

```
Error Message:
Starting BootJarTool...
2008-10-08 10:29:29,278 INFO - Terracotta 2.7.0, as of 20081001-101049
(Revision 10251 by cruise@rh4mo0 from 2.7)
2008-10-08 10:29:30,459 FATAL -
**********************************************************************
The configuration data in the file at "/opt/terracotta/conf/tc-config.xml"
does not obey the Terracotta schema:
[0]: Line 8, column 3: Element not allowed: server in element servers

**********************************************************************
```

This error occurs when there's a schema violation in the Terracotta configuration file, at the line indicated by the error text. To confirm that your configuration file follows the required schema, see the schema file

included with the Terracotta kit. The kit includes schema files (*.xsd) for Terracotta, Ehcache, and Quartz configurations.

The Terracotta servers crash regularly and I see a ChecksumException.

If the logs reveal an error similar to `com.sleepycat.je.log.ChecksumException: Read invalid log entry type: 0 LOG_CHECKSUM`, there is likely a corrupted disk on at least one of the servers.

Why is `java.net.UnknownHostException` thrown when I try to run Terracotta sample applications?

If an UnknownHostException occurs, and you experience trouble running the Terracotta Welcome application and the included sample applications on Linux (especially Ubuntu), you may need to edit the etc/hosts file.

The UnknownHostException may be followed by "unknown-ip-address".

For example, your etc/hosts file may contain settings similar to the following:

```
127.0.0.1       localhost
127.0.1.1    myUbuntu.usa myUbuntu
```

If myUbuntu is the host, you must change 127.0.1.1 to the host's true IP address.

NOTE: You may be able to successfully start Terracotta server instances even with the "invalid" etc/hosts file, and receive no exceptions or errors, but other connectivity problems can occur. For example, when starting two Terracotta servers that should form a mirror group (one active and one standby), you may see behavior that indicates that the servers cannot communicate with each other.

On a node with plenty of RAM and disk space, why is there a failure with errors stating that a "native thread" cannot be created?

You may be exceeding a limit at the system level. In *NIX, run the following command to see what the limits are:

```
ulimit -a
```

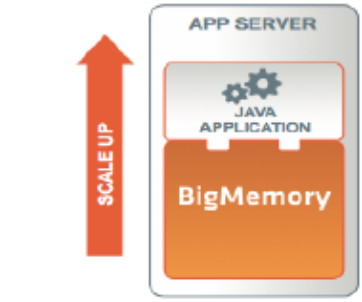
For example, a limit on the number of processes that can run in the shell may be responsible for the errors.

Why does the Terracotta server crash regularly with *java.io.IOException: File exists*?

Early versions of JDK 1.6 had a JVM bug that caused this failure. Update JDK to avoid this issue.

{/toc-zone}

Welcome to BigMemory Go



BigMemory Go lets you put ALL your data in memory,

limited only by how much RAM you have

in your machine.

With BigMemory Go, you get:

- Unlimited in-memory data management on one Java Virtual Machine (JVM)
- 32GB free
- Advanced monitoring, search, and management for in-memory data

To get started, download BigMemory Go. Then check out the handy Get Started guide for instructions on installing your license key, configuring BigMemory, and connecting your apps to your new BigMemory data stores.

If you would like to purchase a license for more than 32GB of BigMemory, with professional support and maintenance, contact us. If you're looking to run BigMemory across distributed servers, check out BigMemory Max.

More questions? Post them to the BigMemory Go support forum.

Getting Started With BigMemory Go

BigMemory Go version 4.0.x

Installing BigMemory Go

Installing BigMemory Go is as easy as downloading the kit and ensuring that the correct files are on your application's classpath. The only platform requirement is using JDK 1.6 or higher.

1. If you do not have a BigMemory Go kit, download it from here.

   The kit is packaged as a tar.gz file. Unpack it on the command line or with the appropriate decompression application.
2. The following JARs are found in the kit's `lib` directory and must be added to your application's classpath:

   - `bigmemory-<version>.jar` – This is the main JAR to enable BigMemory.
   - `ehcache-ee-<version>.jar` – This file contains the API to BigMemory Go.
   - `slf4j-api-<version>.jar` – This file is the bridge, or logging facade, to the BigMemory Go logging framework.
   - `slf4j-jdk14-<version>.jar` – This is a binding JAR for the provided SLF4J logging framework, java.util.logging. Binding JARs for other frameworks are available from the SLF4J website.
3. Save the BigMemory Go license-key file to the BigMemory Go home directory. This file, called `terracotta-license.key`, was attached to an email you received after registering for the BigMemory Go download.

   Alternatively, you can add the license-key file to your application's classpath, or specify it with the following Java system property:

   ```
   -Dcom.tc.productkey.path=/path/to/terracotta-license.key
   ```
4. BigMemory Go uses Ehcache as its user-facing interface. To configure BigMemory Go, create or update an Ehcache configuration file to specify how much off-heap in-memory storage your want to use. You may also configure BigMemory to write data to a local disk store for fast restart. For example:

   ```
   <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
        name="myBigMemoryGoConfig">

     <!-- Tell BigMemory where to write its data to disk. -->
     <diskStore path="/path/to/my/disk/store/directory"/>

     <!-- set "maxBytesLocalOffHeap" to the amount of off-heap memory you
     want to use.  This memory is invisible to the Java garbage collector,
     providing gigabytes to terabytes of in-memory data without garbage
     collection pauses. -->
     <cache name="myBigMemoryGoStore"
          maxBytesLocalHeap="512M"
          maxBytesLocalOffHeap="32G">
       <!-- Tell BigMemory to use the "localRestartable" persistence
       strategy for fast restart (optional). -->
       <persistence strategy="localRestartable"/>
   ```

```
  </cache>
</ehcache>
```

Set `maxBytesLocalOffHeap` to the amount of off-heap storage you want to use. Depending on your data and how much physical RAM you have available, you can use just a few gigabytes to multiple terabytes of off-heap memory in a single JVM for ultra-fast access with no garbage collection pauses—and you get up to 32GB for free.

Name this configuration file `ehcache.xml` and place it in the top-level of your classpath.

For more information on configuration options, refer to the configuration documentation and to the reference ehcache.xml configuration file in the `config-samples` directory of the BigMemory Go kit.

5. Use the `-XX:MaxDirectMemorySize` Java option to allocate enough direct memory in the JVM to accomodate the off-heap storage specified in your configuration, plus at least 250MB to allow for other direct memory usage that might occur in your application. For example:

   ```
   -XX:MaxDirectMemorySize=33G
   ```

   Set `MaxDirectMemorySize` to the amount of BigMemory you have. For more information about this step, refer to Allocating Direct Memory in the JVM.

   Also, allocate at least enough heap using the `-Xmx` Java option to accomodate the on-heap storage specified in your configuration, plus enough extra heap to run the rest of your application. For example:

   ```
   -Xmx1g
   ```

6. Look through the code samples in the documentation and in the download kit for examples of how to employ the various features and capabilities of BigMemory Go.

Configuring BigMemory Go

For a general overview to configuring BigMemory Go, see this introductory page. Specific configuration topics are introduced below.

Automatic Resource Control

Automatic Resource Control (ARC) gives you fine-grained controls for tuning performance and enabling trade-offs between throughput, latency and data access. Independently adjustable configuration parameters include differentiated tier-based sizing and pinning hot or eternal data in the most effective tier.

Dynamically Sizing Stores

Tuning often involves sizing stores appropriately. There are a number of ways to size the different BigMemory Go data tiers using simple configuration sizing attributes. The sizing page explains how to tune tier sizing by configuring dynamic allocation of memory and automatic balancing.

Pinning Data

One of the most important aspects of running an in-memory data store involves managing the life of the data in each BigMemory Go tier. See the data-life page for more information on the pinning, expiration, and

eviction of data.

Fast Restartability

BigMemory Go has full fault tolerance, allowing for continuous access to in-memory data after a planned or unplanned shutdown, with the option to store a fully consistent record of the in-memory data on the local disk at all times. The fast-restart page covers data persistence, fast restartability, and using the local disk as a storage tier for in-memory data (both heap and off-heap stores).

Using the BigMemory Go API

BigMemory Go provides a full-featured API. See the code-samples page for a beginner's view of using the API. Selected advanced API features are introduced below.

Search

Search billions of entries—gigabytes, even terabytes of data—with results returned in less than a second. Data is indexed without significant overhead, and features like "GroupBy' are included.

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

Transactional Caching

Transactional modes are a powerful extension for performing atomic operations on data stores, keeping your data in sync with your database.

The transactions page covers the background and configuration information for BigMemory Go transactional modes. Explicit Locking is another API that can be used as a custom alternative to XA Transactions or Local transactions.

Administration and Monitoring

The Terracotta Management Console (TMC) is a web-based monitoring and administration application for tuning cache usage, detecting errors, and providing an easy-to-use access point to integrate with production management systems.

As an alternative to the TMC, standard JMX-based administration and monitoring is available.

For logging, BigMemory Go uses the flexible SLF4J logging framework.

Learn More About How BigMemory Go Works

- Code Samples to learn what you can do with BigMemory Go.
- Configuration Overview to learn more about how BigMemory Go is configured.
- BigMemory Go Architecture to learn more about how BigMemory Go works.

Code Samples

Introduction

The following code samples illustrate various features of BigMemory Go. They are also available in the BigMemory Go kit in the /code-samples directory.

Title	Description
example01-config-file	BigMemory may be configured declaratively, using an XML configuration file, or programmatically via the fluent configuration API. This sample shows how to configure a basic instance of BigMemory Go declaratively with the XML configuration file.
example02-config-programmatic	Configure a basic instance of BigMemory Go programmatically with the fluent configuration API.
example03-crud	Basic create, retrieve, update and delete (CRUD) operations available in BigMemory Go.
example04-search	Basic in-memory search features of BigMemory Go.
example05-arc	Automatic Resource Control (ARC) is a powerful capability of BigMemory Go that gives users the ability to control how much data is stored in heap memory and off-heap memory. This sample shows the basic configuration options for data tier sizing using ARC.
example06-cache	BigMemory Go is a powerful in-memory data management solution. Among its many applications, BigMemory Go may be used as a cache to speed up access to data from slow or expensive databases and other remote data sources. This example shows how to enable and configure the caching features available in BigMemory Go.

To run the code samples with Maven, you will need to add the Terracotta Maven repositories to your Maven settings.xml file. Add the following repository information to your settings.xml file:

```
<repository>
    <id>terracotta-repository</id>
    <url>http://www.terracotta.org/download/reflector/releases</url>
    <releases>
        <enabled>true</enabled>
    </releases>
</repository>
```

For further information, refer to Working with Apache Maven.

Example 1: Declarative Configuration via XML

To configure BigMemory declaratively with an XML file, create a CacheManager instance, passing the a file name or an URL object to the constructor.

The following example shows how to create a CacheManager with an URL of the XML file in the classpath at `/xml/ehcache.xml`.

```
CacheManager manager = CacheManager.newInstance(
                          getClass().getResource("/xml/ehcache.xml"));
```

Example 1: Declarative Configuration via XML

```
try {
  Cache bigMemory = manager.getCache("BigMemory");
  // now do stuff with it...

} finally {
  if (manager != null) manager.shutdown();
}
```

Here are the contents of the XML configuration file used by this sample:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
         name="config">
  <cache name="BigMemory"
         maxBytesLocalHeap="512M"
         maxBytesLocalOffHeap="32G"
         copyOnRead="true"
         statistics="true"
         eternal="true">
  </cache>
</ehcache>
```

The configuration element `maxBytesLocalOffHeap` lets you set how much off-heap memory to use. BigMemory's unique off-heap memory storage lets you use all of the memory available on a server in a single JVM—from gigabytes to multiple terabytes—without causing garbage collection pauses.

Example 2: Programmatic Configuration

To configure BigMemory Go programmatically, use the Ehcache fluent configuration API.

```
Configuration managerConfiguration = new Configuration()
    .name("bigmemory-config")
    .cache(new CacheConfiguration()
        .name("BigMemory")
        .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
        .maxBytesLocalOffHeap(1, MemoryUnit.GIGABYTES)
        .copyOnRead(true)
        .statistics(true)
        .eternal(true)
    );

CacheManager manager = CacheManager.create(managerConfiguration);
try {
  Cache bigMemory = manager.getCache("BigMemory");
  // now do stuff with it...

} finally {
  if (manager != null) manager.shutdown();
}
```

Example 3: Create, Read, Update and Delete (CRUD)

The CRUD sample demonstrates basic create, read, update and delete operations.

First, we create a BigMemory data store configured to use 512 MB of heap memory and 32 GB of off-heap memory:

```
Configuration managerConfiguration = new Configuration();
managerConfiguration.updateCheck(true)
    .monitoring(Configuration.Monitoring.AUTODETECT)
    .name("config")
    .cache(new CacheConfiguration()
        .name("BigMemory-Crud")
        .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
        .maxBytesLocalOffHeap(32, MemoryUnit.GIGABYTES)
    );

CacheManager manager = CacheManager.create(managerConfiguration);
Cache bigMemory = manager.getCache("BigMemory-Crud");
```

This instructs the Automatic Resource Control (ARC) capability of BigMemory to keep a maximum of 512 MB of its data in heap for nanosecond to microsecond access. In this example, as the 512 MB of heap memory fills up, ARC will automatically move data to the 32 GB off-heap store where it is available at microsecond speed. This configuration keeps heap sizes small to avoid garbage collection pauses and tuning, but still uses large amounts of in-process memory for ultra-fast access to data.

Important: *BigMemory Go is capable of addressing gigabytes to terabytes of in-memory data in a single JVM and it's free to use up to 32 GB. However, to avoid swapping take care not to configure BigMemory Go to use more memory than is physically available on your hardware.*

Now that we have a BigMemory instance configured and available, we can start creating data in it:

```
final Person timDoe = new Person("Tim Doe", 35, Person.Gender.MALE,
    "eck street", "San Mateo", "CA");
bigMemory.put(new Element("key-1", timDoe));
```

Then, we can read data from it:

```
final Element element = bigMemory.get("key-1");
System.out.println("The value for key-1 is  " + element.getObjectValue());
```

And update it:

```
final Person pamelaJones = new Person("Pamela Jones", 23, Person.Gender.FEMALE,
    "berry st", "Parsippany", "LA");
bigMemory.put(new Element("key-1", pamelaJones));
final Element updated = bigMemory.get("key-1");
System.out.println("The value for key-1 is now " + updated.getObjectValue() +
                    ". key-1 has been updated.");
```

And delete it:

```
bigMemory.remove("key-1");
System.out.println("Try to retrieve key-1.");
final Element removed = bigMemory.get("key-1");
System.out.println("Value for key-1 is " + removed +
                   ". Key-1 has been deleted.");
```

You can also create or update multiple entries at once:

```
Collection<Element> elements = new ArrayList<Element>();
elements.add(new Element("1", new Person("Jane Doe", 35,
    Person.Gender.FEMALE, "eck street", "San Mateo", "CA")));
elements.add(new Element("2", new Person("Marie Antoinette", 23,
```

```
    Person.Gender.FEMALE, "berry st", "Parsippany", "LA")));
elements.add(new Element("3", new Person("John Smith", 25,
    Person.Gender.MALE, "big wig", "Beverly Hills", "NJ")));
elements.add(new Element("4", new Person("Paul Dupont", 25,
    Person.Gender.MALE, "big wig", "Beverly Hills", "NJ")));
elements.add(new Element("5", new Person("Juliet Capulet", 25,
    Person.Gender.FEMALE, "big wig", "Beverly Hills", "NJ")));

bigMemory.putAll(elements);
```

And read multiple entries at once:

```
final Map<Object, Element> elementsMap = bigMemory.getAll(
    Arrays.asList("1", "2", "3"));
```

And delete multiple entries at once:

```
bigMemory.removeAll(Arrays.asList("1", "2", "3"));
```

And delete everything at once:

```
bigMemory.removeAll();
```

Example 4: Search

BigMemory Go comes with powerful in-memory search capabilities. This sample shows how to perform basic search operations on your in-memory data.

First, create an instance of BigMemory Go with searchable attributes:

```
Configuration managerConfig = new Configuration()
    .cache(new CacheConfiguration().name("MySearchableDataStore")
        .eternal(true)
        .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
        .maxBytesLocalOffHeap(32, MemoryUnit.GIGABYTES)
        .searchable(new Searchable()
            .searchAttribute(new SearchAttribute().name("age"))
            .searchAttribute(new SearchAttribute().name("gender")
                .expression("value.getGender()"))
            .searchAttribute(new SearchAttribute().name("state")
                .expression("value.getAddress().getState()"))
            .searchAttribute(new SearchAttribute().name("name")
                .className(NameAttributeExtractor.class.getName()))
        )
    );

CacheManager manager = CacheManager.create(managerConfig);
Ehcache bigMemory = manager.getEhcache("MySearchableDataStore");
```

Now, let's put a bunch of stuff into it:

```
bigMemory.put(new Element(1, new Person("Jane Doe", 35, Gender.FEMALE,
    "eck street", "San Mateo", "CA")));
bigMemory.put(new Element(2, new Person("Marie Antoinette", 23, Gender.FEMALE,
    "berry st", "Parsippany", "LA")));
bigMemory.put(new Element(3, new Person("John Smith", 25, Gender.MALE,
    "big wig", "Beverly Hills", "NJ")));
```

Example 4: Search

```
bigMemory.put(new Element(4, new Person("Paul Dupont", 45, Gender.MALE,
    "cool agent", "Madison", "WI")));
bigMemory.put(new Element(5, new Person("Juliet Capulet", 30, Gender.FEMALE,
    "dah man", "Bangladesh", "MN")));
for (int i = 6; i < 1000; i++) {
  bigMemory.put(new Element(i, new Person("Juliet Capulet" + i, 30,
      Person.Gender.MALE, "dah man", "Bangladesh", "NJ")));
}
```

Next, create some search attributes and construct a query:

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

Query query = bm.createQuery();
query.includeKeys();
query.includeValues();
query.addCriteria(name.ilike("Jul*").and(gender.eq(Gender.FEMALE)))
    .addOrderBy(age, Direction.ASCENDING).maxResults(10);
```

Then, execute the query and look at the results:

```
Results results = query.execute();
System.out.println(" Size: " + results.size());
System.out.println("----Results-----\n");
for (Result result : results.all()) {
  System.out.println("Got: Key[" + result.getKey()
                     + "] Value class [" + result.getValue().getClass()
                     + "] Value [" + result.getValue() + "]");
}
```

We can also use Aggregators to perform computations across query results. Here's an example that computes the average age of all people in the data set:

```
Query averageAgeQuery = bigMemory.createQuery();
averageAgeQuery.includeAggregator(Aggregators.average(age));
System.out.println("Average age: "
                   + averageAgeQuery.execute().all().iterator().next()
                       .getAggregatorResults());
```

We can also restrict the calculation to a subset based on search attributes. Here's an example that computes the average age of all people in the data set between the ages of 30 and 40:

```
Query agesBetween = bigMemory.createQuery();
agesBetween.addCriteria(age.between(30, 40));
agesBetween.includeAggregator(Aggregators.average(age));
System.out.println("Average age between 30 and 40: "
                   + agesBetween.execute().all().iterator().next()
                       .getAggregatorResults());
```

Using Aggregators, we can also find number of entries that match our search criteria. Here's an example that finds the number of people in the data set who live in New Jersey:

```
Query newJerseyCountQuery = bigMemory.createQuery().addCriteria(
                                state.eq("NJ"));
newJerseyCountQuery.includeAggregator(Aggregators.count());
```

```
System.out.println("Count of people from NJ: "
                   + newJerseyCountQuery.execute().all().iterator().next()
                       .getAggregatorResults());
```

Example 5: Automatic Resource Control (ARC)

Automatic Resource Control (ARC) is a powerful capability of BigMemory Go that gives users the ability to control how much data is stored in heap memory and off-heap memory.

The following XML configuration instructs ARC to allocate a maximum of 512 M of heap memory and 32 G of off-heap memory. In this example, when 512 M of heap memory is used, ARC will automatically move data into off-heap memory up to a maximum of 32 G. The amount of off-heap memory you can use is limited only by the amount of physical RAM you have available.

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
     updateCheck="false" monitoring="autodetect"
     dynamicConfig="true"
     name="MyManager" maxBytesLocalHeap="512M"
     maxBytesLocalOffHeap="32G">

  <defaultCache>
  </defaultCache>

  <cache name="BigMemory1">
  </cache>

  <cache name="BigMemory2">
  </cache>

</ehcache>
```

It's also possible to allocate resources on a per-data set basis. Here's an example of allocating 8 G of off-heap memory to the BigMemory1 data set and 24 G of off-heap memory to the BigMemory2 data set:

```
<ehcache xmlns
 ...
     name="MyManager"
     maxBytesLocalHeap="512M"
     maxBytesLocalOffHeap="32G"
     maxBytesLocalDisk="128G">

  <cache name="BigMemory1"
         maxBytesLocalOffHeap="8G">
  </cache>

  <cache name="BigMemory2"
         maxBytesLocalOffHeap="24G">
  </cache>

</ehcache>
```

Example 6: Using BigMemory As a Cache

BigMemory Go is a powerful in-memory data management solution. Among its many applications, BigMemory Go may be used as a cache to speed up access to data from slow or expensive databases and other

remote data sources. This example shows how to enable and configure the caching features available in BigMemory Go.

The following programmatic configuration snippet shows how to set time-to-live (TTL) and time-to-idle (TTI) policies on a data set:

```
Configuration managerConfiguration = new Configuration();
managerConfiguration.updateCheck(true)
    .monitoring(Configuration.Monitoring.AUTODETECT)
    .name("cacheManagerCompleteExample")
    .addCache(
        new CacheConfiguration()
            .name("sample-cache")
            .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
            .maxBytesLocalOffHeap(1, MemoryUnit.GIGABYTES)
            .timeToLiveSeconds(4)
            .timeToIdleSeconds(2)

    );

CacheManager manager = CacheManager.create(managerConfiguration);
```

The `timeToLiveSeconds` directive sets the maximum age of an element in the data set. Elements older than the maximum TTL will not be returned from the data store. This is useful when BigMemory is used as a cache of external data and you want to ensure the freshness of the cache.

The `timeToIdleSeconds` directive sets the maximum time since last access of an element. Elements that have been idle longer than the maximum TTI will not be returned from the data store. This is useful when BigMemory is being used as a cache of external data and you want to bias the eviction algorithm towards removing idle entries.

If neither TTL nor TTI are set (or set to zero), data will stay in BigMemory until it is explicitly removed.

Configuration Overview

Introduction

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

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

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

This documentation focuses on XML declarative configuration. Programmatic configuration is explored in certain examples and is documented in Javadocs.

XML Configuration

BigMemory Go uses Ehcache as its user-facing interface and is configured using the Ehcache configuration system. By default, Ehcache looks for an ASCII or UTF8 encoded XML configuration file called `ehcache.xml` at the top level of the Java classpath. You may specify alternate paths and filenames for the XML configuration file by using the various CacheManager constructors.

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

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

Dynamically Changing Cache Configuration

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

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

cache. The default value is 0, which means no TTI eviction takes place (infinite lifetime).

- Local sizing attributes
    - maxEntriesLocalHeap
    - maxBytesLocalHeap
    - maxEntriesLocalDisk
    - maxBytesLocalDisk.
- memory-store eviction policy
- CacheEventListeners can be added and removed dynamically

Note that the `eternal` attribute, when set to "true", overrides `timeToLive` and `timeToIdle` so that no expiration can take place.

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

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

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

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

In `ehcache.xml`, you can disable dynamic configuration by setting the `<ehcache>` element's `dynamicConfig` attribute to "false". For a complete list of configuration parameters that can be changed dynamically, see this introduction to Ehcache configuration.

Passing Copies Instead of References

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

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

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

The default configuration is "false" for both options.

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

```
<cache name="copyCache"
    maxEntriesLocalHeap="10"
```

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

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

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

```
CacheConfiguration cacheConfiguration = new CacheConfiguration("copyCache", 10);

CopyStrategyConfiguration copyStrategyConfiguration = new CopyStrategyConfiguration();
copyStrategyConfiguration.setClass("com.company.ehcache.MyCopyStrategy");

cacheConfiguration.addCopyStrategy(copyStrategyConfiguration);
```

Special System Properties

net.sf.ehcache.disabled

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

net.sf.ehcache.use.classic.lru

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

ehcache.xsd

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

Each BigMemory Go distribution also contains a copy of `ehcache.xsd`.

ehcache-failsafe.xml

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

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

ehcache-failsafe.xml

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

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

About Default Cache

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

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

More Information on Configuration Topics

Topic	Description
Sizing Caches	Tuning Ehcache often involves sizing cached data appropriately. Ehcache provides a number of ways to size the different data tiers using simple cache-configuration sizing attributes. This page explains simplified tuning of cache size by configuring dynamic allocation of memory and automatic load balancing.
Expiration, Pinning, and Eviction	The architecture of an Ehcache node can include a number of tiers that store data. One of the most important aspects of managing cached data involves managing the life of the data in each tier. This page covers managing data life in Ehcache and the Terracotta Server Array, including the pinning features of Automatic Resource Control (ARC).
Fast Restartability	This page covers persistence, fast restartability, and using the local disk as a storage tier. The Fast Restart feature provides enterprise-ready crash resilience, which can serve as a fast recovery system after failures, a hot mirror of the data set on the disk at the application node, and an operational store with in-memory speed for reads and writes.
Code Samples	Examples of working with key classes and methods such as `CacheManager` and `Cache`, loading configuration, and getting statistics.

Storage Tiers Basics

Introduction

BigMemory Go has three storage tiers, summarized here:

- Memory store – Heap memory that holds a copy of the hottest subset of data from the off-heap store. Subject to Java GC.
- Off-heap store – Limited in size only by available RAM. Not subject to Java GC. Can store serialized data only. Provides overflow capacity to the memory store.
- Disk store – Backs up in-memory data and provides overflow capacity to the other tiers. Can store serialized data only.

This document defines these storage tiers and details the suitable element types for each storage tier.

Before running in production, it is strongly recommended that you test the BigMemory Go tiers with the actual amount of data you expect to use in production. For information about sizing the tiers, refer to Cache Configuration Sizing Attributes.

Memory Store

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

The memory store has the following characteristics:

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

Off-Heap Store

The off-heap store extends the in-memory store to memory outside the of the object heap. This store, which is not subject to Java GC, is limited only by the amount of RAM available.

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

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

Allocating Direct Memory in the JVM

The off-heap store uses the direct-memory portion of the JVM. You must allocate sufficient direct memory for the off-heap store by using the JVM property `MaxDirectMemorySize`.

For example, to allocate 2GB of direct memory in the JVM:

```
java -XX:MaxDirectMemorySize=2G ...
```

Since direct memory may be shared with other processes, allocate at least 256MB (or preferably 1GB) more to direct memory than will be allocated to the off-heap store.

Note the following about allocating direct memory:

- If you configure off-heap memory but do not allocate direct memory with -XX:MaxDirectMemorySize, the default value for direct memory depends on your version of your JVM. Oracle HotSpot has a default equal to maximum heap size (-Xmx value), although some early versions may default to a particular value.
- MaxDirectMemorySize must be added to the local node's startup environment.
- Direct memory, which is part of the Java process heap, is separate from the object heap allocated by -Xmx. The value allocated by MaxDirectMemorySize must not exceed physical RAM, and is likely to be less than total available RAM due to other memory requirements.
- The amount of direct memory allocated must be within the constraints of available system memory and configured off-heap memory.
- The maximum amount of direct memory space you can use depends on the process data model (32-bit or 64-bit) and the associated operating system limitations, the amount of virtual memory available on the system, and the amount of physical memory available on the system.

Using Off-Heap Store with 32-bit JVMs

The amount of heap-offload you can achieve is limited by addressable memory. 64-bit systems can allow as much memory as the hardware and operating system can handle, while 32-bit systems have strict limitations on the amount of memory that can be effectively managed.

For a 32-bit process model, the maximum virtual address size of the process is typically 4GB, though most 32-bit operating systems have a 2GB limit. The maximum heap size available to Java is lower still due to particular operating-system (OS) limitations, other operations that may run on the machine (such as mmap operations used by certain APIs), and various JVM requirements for loading shared libraries and other code. A useful rule to observe is to allocate no more to off-heap memory than what is left over after -Xmx is set. For example, if you set -Xmx3G, then off-heap should be no more than 1GB. Breaking this rule may not cause an OutOfMemoryError on startup, but one is likely to occur at some point during the JVM's life.

If Java GC issues are afflicting a 32-bit JVM, then off-heap store can help. However, note the following:

- Everything has to fit in 4GB of addressable space. If 2GB of heap is allocated (with `-Xmx2g`) then at most are are 2GB left for off-heap data.
- The JVM process requires some of the 4GB of addressable space for its code and shared libraries plus any extra Operating System overhead.
- Allocating a 3GB heap with -Xmx, as well as 2047MB of off-heap memory, will not cause an error at startup, but when it's time to grow the heap an OutOfMemoryError is likely.

- If both -Xms3G and -Xmx3G are used with 2047MB of off-heap memory, the virtual machine will start but then complain as soon as the off-heap store tries to allocate the off-heap buffers.
- Some APIs, such as `java.util.zip.ZipFile` on a 1.5 JVM, may <mmap> files in memory. This will also use up process space and may trigger an OutOfMemoryError.

Tuning Off-Heap Store Performance

Memory-related or performance issues that arise during operations can be related to improper allocation of memory to the off-heap store. If performance or functional issues arise that can be traced back to the off-heap store, see the suggested tuning tips in this section.

General Memory allocation

Committing too much of a system's physical memory is likely to result in paging of virtual memory to disk, quite likely during garbage-collection operations, leading to significant performance issues. On systems with multiple Java processes, or multiple processes in general, the sum of the Java heaps and off-heap stores for those processes should also not exceed the size of the physical RAM in the system. Besides memory allocated to the heap, Java processes require memory for other items, such as code (classes), stacks, and PermGen.

Note that MaxDirectMemorySize sets an upper limit for the JVM to enforce, but does not actually allocate the specified memory. Overallocation of direct memory (or buffer) space is therefore possible, and could lead to paging or even memory-related errors. The limit on direct buffer space set by MaxDirectMemorySize should take into account the total physical memory available, the amount of memory that is allotted to the JVM object heap, and the portion of direct buffer space that other Java processes may consume.

In addition, be sure to allocate at least 15 percent more off-heap memory than the size of your data set. To maximize performance, a portion of off-heap memory is reserved for meta-data and other purposes.

Note also that there could be other users of direct buffers (such as NIO and certain frameworks and containers). Consider allocating additional direct buffer memory to account for that additional usage.

Compressed References

For 64-bit JVMs running Java 6 Update 14 or higher, consider enabling compressed references to improve overall performance. For heaps up to 32GB, this feature causes references to be stored at half the size, as if the JVM is running in 32-bit mode, freeing substantial amounts of heap for memory-intensive applications. The JVM, however, remains in 64-bit mode, retaining the advantages of that mode.

For the Oracle HotSpot, compressed references are enabled using the option `-XX:+UseCompressedOops`. For IBM JVMs, use `-Xcompressedrefs`.

Slow Off-Heap Allocation

Based configuration, usage, and memory requirements, BigMemory Go could allocate off-heap memory multiple times. If off-heap memory comes under pressure due to over-allocation, the host OS may begin paging to disk, thus slowing down allocation operations. As the situation worsens, an off-heap buffer too large to fit in memory can quickly deplete critical system resources such as RAM and swap space and crash the host OS.

To stop this situation from degrading, off-heap allocation time is measured to avoid allocating buffers too

large to fit in memory. If it takes more than 1.5 seconds to allocate a buffer, a warning is issued. If it takes more than 15 seconds, the JVM is halted with `System.exit()` (or a different method if the Security Manager prevents this).

To prevent a JVM shutdown after a 15-second delay has occurred, set the `net.sf.ehcache.offheap.DoNotHaltOnCriticalAllocationDelay` system property to true. In this case, an error is logged instead.

Swapiness and Huge Pages

An OS could swap data from memory to disk even if memory is not running low. For the purpose of optimization, data that appears to be unused may be a target for swapping. Because BigMemory Go can store substantial amounts of data in RAM, its data may be swapped by the OS. But swapping can degrade overall cluster performance by introducing thrashing, the condition where data is frequently moved forth and back between memory and disk.

To make heap memory use more efficient, Linux, Microsoft Windows, and Oracle Solaris users should review their configuration and usage of swappiness as well as the size of the swapped memory pages. In general, BigMemory Go benefits from lowered swappiness and the use of *huge pages* (also known as *big pages*, *large pages*, and *superpages*).

Settings for these behaviors vary by OS and JVM. For Oracle HotSpot, `-XX:+UseLargePages` and `-XX:LargePageSizeInBytes=<size>` (where <size> is a value allowed by the OS for specific CPUs) can be used to control page size. However, note that this setting does *not* affect how off-heap memory is allocated. Over-allocating huge pages while also configuring substantial off-heap memory *can starve off-heap allocation and lead to memory and performance problems*.

Maximum Serialized Size of an Element

This section applies when using BigMemory through the Ehcache API.

Unlike the memory and the disk stores, by default the off-heap store has a 4MB limit for classes with high quality hashcodes, and 256KB limit for those with pathologically bad hashcodes. The built-in classes such as `String` and the `java.lang.Number` subclasses Long and Integer have high quality hashcodes. This can issues when objects are expected to be larger than the default limits.

To override the default size limits, set the system property `net.sf.ehcache.offheap.cache_name.config.idealMaxSegmentSize` to the size you require.

For example,

```
net.sf.ehcache.offheap.com.company.domain.State.config.idealMaxSegmentSize=30M
```

Reducing Faulting

While the memory store holds a hotset (a subset) of the entire data set, the off-heap store should be large enough to hold the entire data set. The frequency of misses (get operations that fail to find the data in memory) begins to rise when the data is too large to fit into off-heap memory, forcing gets to fetch data from the disk store (called *faulting*). More misses in turn raise latency and lower performance.

For example, tests with a 4GB data set and a 5GB off-heap store recorded no misses. With the off-heap store reduced to 4GB, 1.7 percent of cache operations resulted in misses. With the off-heap store at 3GB, misses reached 15 percent.

Disk Store

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

For more information about data persistence on disk, refer to the Fast Restartability page.

Serialization

Only data that is `Serializable` can be placed in the disk store. Writes to and from the disk use `ObjectInputStream` and the Java serialization mechanism. Any non-serializable data overflowing to the disk store is removed and a NotSerializableException is thrown.

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

- The serialization time for a Java object consisting of a large Map of String arrays was 126ms, where the serialized size was 349,225 bytes.
- The serialization time for a byte[] was 7ms, where the serialized size was 310,232 bytes.

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

Storage Options

Two disk-store options are available:

- Temporary swap – Allows the memory and off-heap stores to overflow to disk when they become full. This makes the disk a temporary store because overflow data does not survive restarts or failures. When the node is restarted, any existing data on disk is cleared because it is not designed to be reloaded.
- Restartable store – Mirrors the data in memory and provides failure recovery of data. When the node is restarted, the data set is reloaded from disk to the in-memory stores.

Storage Tiers Advanced

Introduction

When using BigMemory Go through its Ehcache API, many aspects of the data storage tiers can be controlled through the Ehcache configuration.

Configuring the Memory Store

The memory store is the top tier and is automatically used by BigMemory Go to store the data hotset because it is the fastest store. It requires no special configuration to enable, and its overall size is taken from the Java heap size. Since it exists in the heap, it is limited by Java GC constraints.

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

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

When an element is added to a cache and it goes beyond its maximum memory size, an existing element is either deleted, if overflow is not enabled, or evaluated for spooling to another tier, if overflow is enabled. The overflow options are `overflowToOffHeap` and `<persistence>` (disk store).

If overflow is enabled, a check for expiry is carried out. If it is expired it is deleted; if not it is spooled. The eviction of an item from the memory store is based on the optional `MemoryStoreEvictionPolicy` attribute specified in the configuration file. Legal values are LRU (default), LFU and FIFO:

- **Least Recently Used (LRU)** *Default**—LRU is the default setting. The last-used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.
- **Least Frequently Used (LFU)**—For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element with least number of hits, the Less Frequently Used element, is evicted.
- **First In First Out (FIFO)**—Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

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

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

TIP: Calculating the Size of a Cache

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

An alternative would be to have an expiry thread. This is a trade-off between lower memory use and short locking periods and CPU utilization. The design is in favour of the latter. For those concerned with memory use, simply reduce the cache's size in memory for more information).

Configuring the Off-Heap Store

If an off-heap store is configured, the corresponding memory store overflows to that off-heap store. Configuring an off-heap store can be done either through XML or programmatically. With either method, the off-heap store is configured on a per-cache basis.

Declarative Configuration

The following XML configuration creates an off-heap cache with an in-heap store (maxEntriesLocalHeap) of 10,000 elements which overflow to a 1-gigabyte off-heap area.

```
<ehcache updateCheck="false" monitoring="off" dynamicConfig="false">
  <defaultCache maxEntriesLocalHeap="10000"
              eternal="true"
              memoryStoreEvictionPolicy="LRU"
              statistics="false" />

  <cache name="sample-offheap-cache"
       maxEntriesLocalHeap="10000"
       eternal="true"
       memoryStoreEvictionPolicy="LRU"
       overflowToOffHeap="true"
       maxBytesLocalOffHeap="1G"/>
</ehcache>
```

The configuration options are:

overflowToOffHeap

Values may be true or false. When set to true, enables the cache to utilize off-heap memory storage to improve performance. Off-heap memory is not subject to Java GC cycles and has a size limit set by the Java property MaxDirectMemorySize. The default value is false.

maxBytesLocalOffHeap

Sets the amount of off-heap memory available to the cache and is in effect only if `overflowToOffHeap` is tr ue. The minimum amount that can be allocated is 1MB. There is no maximum.

For more information on sizing data stores, refer to this page.

NOTE: Heap Configuration When Using an Off-heap Store

You should set maxEntriesLocalHeap to at least 100 elements when using an off-heap store to avoid performance degradation. Lower values for maxEntriesLocalHeap trigger a warning to be logged.

Programmatic Configuration

The equivalent cache can be created using the following programmatic configuration:

```
public Cache createOffHeapCache()
{
  CacheConfiguration config = new CacheConfiguration("sample-offheap-cache", 10000)
  .overflowToOffHeap(true).maxBytesLocalOffHeap("1G");
  Cache cache = new Cache(config);
  manager.addCache(cache);
  return cache;
}
```

Configuring the Disk Store

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

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

Disk Store Storage Options

Two disk store options are available:

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

See the following sections for more information on these options.

localTempSwap

The "localTempSwap" persistence strategy allows local disk usage during BigMemory operation, providing an extra tier for storage. This disk storage is temporary and is cleared after a restart.

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

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

localRestartable

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

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

Disk Store Configuration Element

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

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

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

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

The following system properties are also legal, in which case they are translated:

- user.home - User's home directory
- user.dir - User's current working directory
- java.io.tmpdir - Default temp file path
- ehcache.disk.store.dir - A system property you would normally specify on the command line—for example, `java -Dehcache.disk.store.dir=/u01/myapp/diskdir`.

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

```
user.dir/one
```

To programmatically set a disk store path:

```
DiskStoreConfiguration diskStoreConfiguration = new DiskStoreConfiguration();

diskStoreConfiguration.setPath("/my/path/dir");

// Already created a configuration object ...
configuration.addDiskStore(diskStoreConfiguration);

CacheManager mgr = new CacheManager(configuration);
```

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

Disk Store Expiry and Eviction

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

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

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

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

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

Turning off Disk Stores

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

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

Configuration Examples

These examples show how to allocate 8GB of machine memory to different stores. It assumes a data set of 7GB - say for a cache of 7M items (each 1kb in size).

Those who want minimal application response-time variance (or minimizing GC pause times), will likely want all the cache to be off-heap. Assuming that 1GB of heap is needed for the rest of the app, they will set their Java config as follows:

```
java -Xms1G -Xmx1G -XX:maxDirectMemorySize=7G
```

And their Ehcache config as:

```
<cache
 maxEntriesLocalHeap=100
 overflowToOffHeap="true"
 maxBytesLocalOffHeap="6G"
... />
```

NOTE: Direct Memory and Off-heap Memory Allocations

To accommodate server communications layer requirements, the value of maxDirectMemorySize must be greater than the value of maxBytesLocalOffHeap. The exact amount greater depends upon the size of maxBytesLocalOffHeap. The minimum is 256MB, but if you allocate 1GB more to the maxDirectMemorySize, it will certainly be sufficient. The server will only use what it needs and the rest will remain available.

Those who want best possible performance for a hot set of data, while still reducing overall application repsonse time variance, will likely want a combination of on-heap and off-heap. The heap will be used for the hot set, the offheap for the rest. So, for example if the hot set is 1M items (or 1GB) of the 7GB data. They will set their Java config as follows

```
java -Xms2G -Xmx2G -XX:maxDirectMemorySize=6G
```

And their Ehcache config as:

```
<cache
  maxEntriesLocalHeap=1M
  overflowToOffHeap="true"
  maxBytesLocalOffHeap="5G"
  ...>
```

This configuration will compare VERY favorably against the alternative of keeping the less-hot set in a database (100x slower) or caching on local disk (20x slower).

Where the data set is small and pauses are not a problem, the whole data set can be kept on heap:

```
<cache
  maxEntriesLocalHeap=1M
  overflowToOffHeap="false"
  ...>
```

Where latency isn't an issue, the disk can be used:

```
<cache
  maxEntriesLocalHeap=1M
  <persistence strategy="localTempSwap"/>
  ...>
```

Sizing Storage Tiers

Introduction

Tuning BigMemory Go often involves sizing the data storage tiers appropriately. BigMemory Go provides a number of ways to size the different data tiers using simple sizing attributes. These sizing attributes affect memory and disk resources.

Sizing Attributes

The following table summarizes the sizing attributes in BigMemory Go's Ehcache API.

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

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

Pooling Resources Versus Sizing Individual Data Sets

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

If there is no CacheManager-level pool specified for a tier, an individual data set claims the amount of that tier specified in its configuration. If there is a CacheManager-level pool specified for a tier, an individual data set

claims that amount *from the pool*. In this case, data sets with no size configuration for that tier receive an equal share of the remainder of the pool (after data sets with explicit sizing configuration have claimed their portion).

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

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

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

Memory Store (Heap)

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

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

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

Off-Heap Store

Off-heap sizing can be configured in bytes only, never by entries.

If a CacheManager has a pool configured for off-heap, your application cannot add caches dynamically that have off-heap configuration — doing so generates an error. In addition, if any caches that used the pool are removed programmatically or through the Terracotta Management Console (TMC), other caches in the pool cannot claim the unused portion. To allot the entire off-heap pool to the remaining caches, remove the unwanted cache from the Ehcache configuration and then reload the configuration.

To use off-heap as a data tier, a cache must have `overflowToOffHeap` set to "true". If a CacheManager has a pool configured for using off-heap, the `overflowToOffHeap` attribute is automatically set to "true" for all caches. In this case, you can prevent a specific cache from overflowing to off-heap by explicitly setting its `overflowToOffHeap` attribute to "false".

Note that an exception is thrown if any cache using an off-heap store attempts to put an element that will cause the off-heap store to exceed its allotted size. The exception will contain a message similar to the following:

```
The element '[ key = 25274, value=[B@3ebb2a91, version=1, hitCount=0,
CreationTime = 1367966069431, LastAccessTime = 1367966069431 ]'
```

```
is too large to be stored in this offheap store.
```

Disk Store

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

To use the disk as a temporary tier during BigMemory operation, set the `persistenceStrategy` to "localTempSwap" (refer to Temporary Disk Storage), and use the `maxBytesLocalDisk` setting to configure the size of this tier.

To use the disk for disk persistence, refer to Disk Persistence Implementation.

Sizing Examples

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

Pooled Resources

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

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="100M"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="50G">
...

<cache name="Cache1" ... </cache>
<cache name="Cache2" ... </cache>
<cache name="Cache3" ... </cache>

</ehcache>
```

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

Explicitly Sizing Data Sets

You can explicitly allocate resources to specific caches:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="100M"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="60G">
...

<cache name="Cache1" ...
        maxBytesLocalHeap="50M"
        ...
  </cache>
```

```
<cache name="Cache2" ...
       maxBytesLocalOffHeap="5G"
       ...
  </cache>
<cache name="Cache3" ... </cache>

</ehcache>
```

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

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

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

Mixed Sizing Configurations

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

```
<ehcache xmlns...
        Name="CM2"
        maxBytesLocalHeap="100M">
...

<cache name="Cache4" ...
        maxBytesLocalHeap="50M"
        maxEntriesLocalDisk="100000"
        ...
  </cache>

<cache name="Cache5" ...
       maxBytesLocalOffHeap="10G"
       ...
  </cache>
<cache name="Cache6" ... </cache>

</ehcache>
```

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

Using Percents

The following configuration sets pools for each tier:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="1G"
        maxBytesLocalOffHeap="10G"
```

```
        maxBytesLocalDisk="50G">
...

<!-- Cache1 gets 400Mb of heap, 2.5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache1" ...
maxBytesLocalHeap="40%">
</cache>

<!-- Cache2 gets 300Mb of heap, 5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache2" ...
maxBytesLocalOffHeap="50%">
</cache>

<!-- Cache2 gets 300Mb of heap, 2.5Gb of off-heap, and 40Gb of disk. -->
<cache name="Cache3" ...
maxBytesLocalDisk="80%">
</cache>
</ehcache>
```

NOTE: Configuring Cache Sizes with Percentages

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

Sizing Data Sets Without a Pool

The CacheManager in this example does not pool any resources.

```
<ehcache xmlns...
        Name="CM3"
       ... >
...

<cache name="Cache7" ...
        maxBytesLocalHeap="50M"
        maxEntriesLocalDisk="100000"
        ...
  </cache>

<cache name="Cache8" ...
       maxEntriesLocalHeap="1000"
       maxBytesLocalOffHeap="10G"
       ...
  </cache>
<cache name="Cache9" ...
       maxBytesLocalHeap="50M"
...
</cache>

</ehcache>
```

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

Overflows

Caches that do not specify overflow will overflow if a pool is set for off-heap and disk.

```
<ehcache maxBytesLocalHeap="1g" maxBytesLocalOffHeap="4g"
       maxBytesLocalDisk="100g" >

<cache name="explicitlyAllocatedCache1"
       maxBytesLocalHeap="50m"
       maxBytesLocalOffHeap="200m"
       timeToLiveSeconds="100">
</cache>

<!-- Does not overflow to disk because overflow has been explicitly
disabled. -->
<cache name="explicitlyAllocatedCache2"
       maxLocalHeap="10%"
       maxBytesLocalOffHeap="200m"
       timeToLiveSeconds="100">
       <persistence strategy="none"/>
</cache>

<!-- Overflows automatically to off-heap and disk because no specific override and resources
     are set at the CacheManager level -->
<cache name="automaticallyAllocatedCache1"
       timeToLiveSeconds="100">
</cache>

<!-- Does not use off-heap because overflow has been explicitly
disabled. -->
<cache name="automaticallyAllocatedCache2"
       timeToLiveSeconds="100"
       overflowToOffHeap="false">
</cache>
</ehcache>
```

Overriding Size Limitations

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

Built-In Sizing Computation and Enforcement

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

Sizing of Data Set Entries

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

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

Ignoring for Size Calculations

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

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

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

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

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

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

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

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

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

```
# That field references a common graph between all cached entries
com.pany.domain.cache.MyCacheEntry.sharedInstance

# This will ignore all instances of that type
com.pany.domain.SharedState

# This ignores a package
com.pany.example
```

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

Configuration for Limiting the Traversed Object Graph

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

Size-Of Limitation at the CacheManager Level

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

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

This element has the following attributes

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

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

Size-Of Limitation at the Cache level

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

Debugging of Size-Of Related Errors

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

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

Eviction When Using CacheManager-Level Storage

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

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

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

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

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

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

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

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

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

Simplifying:

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

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

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

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

Pinning, Expiration, and Eviction

Introduction

This page covers managing the life of the data in each of BigMemory's data storage tiers, including the pinning features of Automatic Resource Control (ARC).

The following are options for data life within the BigMemory tiers:

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

Setting Expiration

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

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

- `timeToIdleSeconds` – The maximum number of seconds an element can exist in the BigMemory data store without being accessed. The element expires at this limit and will no longer be returned from BigMemory. The default value is 0, which means no TTI eviction takes place (infinite lifetime).
- `timeToLiveSeconds` – The maximum number of seconds an element can exist in the the BigMemory data store regardless of use. The element expires at this limit and will no longer be returned from BigMemory. The default value is 0, which means no TTL eviction takes place (infinite lifetime).
- `eternal` – If the cache's `eternal` flag is set, it overrides any finite TTI/TTL values that have been set. Individual cache elements may also be set to eternal. Eternal elements and caches do not expire, however they may be evicted.

See How Configuration Affects Element Eviction for more information on how configuration can impact eviction.

Pinning Data

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

Data that should remain in memory can be pinned. You cannot pin individual entries, only an entire cache. There are two types of pinning, depending upon whether the pinning configuration should take precedence over resource constraints or the other way around. See the next sections for details.

Configuring Pinning

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

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

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

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

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

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

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

Pinning and Cache Sizing

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

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

Use caution with `inCache` pinning

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

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

```
<cache name="myCache"
     maxEntriesOnHeap="10000"
     maxBytesLocalOffHeap="8g"
     ... >
```

```
    <pinning store="localMemory" />
</cache>
```

Scope of Pinning

Pinning achieved programmatically will not be persisted — after a restart the pinned entries are no longer pinned. Cache pinning in configuration is reinstated with the configuration file.

Explicitly Removing Data

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

How Configuration Affects Element Flushing and Eviction

The following example shows a cache with certain expiration settings:

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

Note the following about the myCache configuration:

- If a client accesses an entry in myCache that has been idle for more than an hour (`timeToIdleSeconds`), that element is evicted.
- If an entry expires but is not accessed, and no resource constraints force eviction, then the expired entry remains in place.
- Entries in myCache can live forever if accessed at least once per 60 minutes (`timeToLiveSeconds`). However, unexpired entries may still be flushed based on other limitations (see Sizing BigMemory Tiers).

Data Freshness and Expiration

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

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

- **Data Expiration**: Use the eviction algorithms included with Ehcache, along with the `timeToIdleSeconds` and `timetoLiveSeconds` settings, to enforce a maximum time for elements to live in the cache (forcing a re-load from the database or SOR).
- **Message Bus**: Use an application to make all updates to the database. When updates are made, post a message onto a message queue with a key to the item that was updated. All application instances can subscribe to the message bus and receive messages about data that is updated, and can synchronize their local copy of the data accordingly (for example by invalidating the cache entry for updated data)

- **Triggers**: Using a database trigger can accomplish a similar task as the message bus approach. Use the database trigger to execute code that can publish a message to a message bus. The advantage to this approach is that updates to the database do not have to be made only through a special application. The downside is that not all database triggers support full execution environments and it is often unadvisable to execute heavy-weight processing such as publishing messages on a queue during a database trigger.

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

If you choose the Data Expiration method, you can read more about the cache configuration settings at cache eviction algorithms, and review the timeToIdle and timeToLive configuration settings. The most important consideration when using the expiration method is balancing data freshness with database load. The shorter you make the expiration settings - meaning the more "fresh" you try to make the data - the more load you will incur on the database.

Try out some numbers and see what kind of load your application generates. Even modestly short values such as 5 or 10 minutes will afford significant load reductions.

Fast Restartability

Introduction

BigMemory's Fast Restart feature provides enterprise-ready crash resilience by keeping a fully consistent, real-time record of your in-memory data. After any kind of shutdown — planned or unplanned — the next time your application starts up, all of the data that was in BigMemory is still available and very quickly accessible.

The advantages of the Fast Restart feature include:

- In-memory data survives crashes and enables fast restarts. Because your in-memory data does not need to be reloaded from a remote data source, applications can resume at full speed after a restart.
- A real-time record of your in-memory data provides true fault tolerance. Even with BigMemory, where terabytes of data can be held in memory, the Fast Restart feature provides the equivalent of a local "hot mirror," which guarantees full data consistency.
- A consistent record of your in-memory data opens many possibilities for business innovation, such as arranging data sets according to time-based needs or moving data sets around to different locations. The uses of the Fast Restart store can range from a simple key-value persistence mechanism with fast read performance, to an operational store with in-memory speeds during operation for both reads and writes.

Data Persistence Implementation

The BigMemory Fast Restart feature works by creating a real-time record of the in-memory data, which it persists in a Fast Restart store on the local disk. After any restart, the data that was last in memory (both heap and off-heap stores) automatically loads from the Fast Restart store back into memory.

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

```
<cache>
   <persistence strategy="localRestartable|localTempSwap|none|distributed"
      synchronousWrites="false|true"/>
</cache>
```

Strategy Options

The options for the `strategy` attribute are:

- **"localRestartable"** — Enables the Fast Restart feature which automatically logs all BigMemory data. This option provides fast restartability with fault tolerant data persistence.
- **"localTempSwap"** — Enables temporary local disk usage. This option provides an extra tier for data storage during operation, but this store is not persisted. After a restart, the disk is cleared of any BigMemory data.
- **"none"** — Does not offload data to disk. With this option, all of the working data is kept in memory only. This is the default mode.
- **"distributed"** — Defers to the `<terracotta>` configuration for persistence settings. This option is for BigMemory Max only.

Synchronous Writes Options

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

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

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

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

Note: The `synchronousWrites` attribute is also available in the `<terracotta>` sub-element. If configured in both places, it must have the same value.

Disk Store Path

The path to the directory where any required disk files will be created is configured with the `<diskStore>` sub-element of the Ehcache configuration.

- For "localRestartable", a unique and explicitly specified path is required.
- For "localTempSwap", if the disk store path is not specified, a default path is used for the disk tier, and the default path will be auto-resolved in the case of a conflict with another CacheManager.

  Note: The Fast Restart feature does not use the disk tier in the same way that conventional disk persistence does. Therefore, when configured for "localRestartable", diskStore size measures such as Cache.getDiskStoreSize() or Cache.calculateOnDiskSize() are not applicable and will return zero. On the other hand, when configured for "localTempSwap", these measures will return size values.

Configuration Examples

This section presents possible disk usage configurations for BigMemory Go.

Options for Crash Resilience

The following configuration provides fast restartability with fully consistent data persistence:

```
<ehcache>
  <diskStore path="/path/to/store/data"/>
  <cache>
    <persistence strategy="localRestartable" synchronousWrites="true"/>
```

```
  </cache>
</ehcache>
```

The following configuration provides fast restartability with eventually consistent data persistence:

```
<ehcache>
  <diskStore path="/path/to/store/data"/>
  <cache>
    <persistence strategy="localRestartable" synchronousWrites="false"/>
  </cache>
</ehcache>
```

Temporary Disk Storage

The "localTempSwap" persistence strategy creates a local disk tier for in-memory data during BigMemory operation. The disk storage is temporary and is cleared after a restart.

```
<ehcache>
  <diskStore path="java.io.tmpdir"/>
  <cache>
    <persistence strategy="localTempSwap"/>
  </cache>
</ehcache>
```

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

In-memory Only

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

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

Programmatic Configuration Example

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

```
Configuration cacheManagerConfig = new Configuration()
    .diskStore(new DiskStoreConfiguration()
    .path("/path/to/store/data"));
CacheConfiguration cacheConfig = new CacheConfiguration()
    .name("my-cache")
    .maxBytesLocalHeap(16, MemoryUnit.MEGABYTES)
    .maxBytesLocalOffHeap(256, MemoryUnit.MEGABYTES)
    .persistence(new PersistenceConfiguration().strategy(Strategy.LOCALRESTARTABLE));

cacheManagerConfig.addCache(cacheConfig);

CacheManager cacheManager = new CacheManager(cacheManagerConfig);
Ehcache myCache = cacheManager.getEhcache("my-cache");
```

Fast Restart Performance

When configured for fast restartability ("localRestartable" persistence strategy), BigMemory becomes active on restart after all of the in-memory data is loaded. The amount of time until BigMemory is restarted is proportionate to the amount of in-memory data and the speed of the underlying infrastructure. Generally, recovery can occur as fast as the disk speed. With an SSD, for example, if you have a read throughput of 1 GB per second, you will see a similar loading speed during recovery.

Fast Restart Limitations

The following recommendations should be observed when configuring BigMemory for fast restartability:

- The size of on-heap or off-heap stores should not be changed during a shutdown. If the amount of memory allocated is reduced, elements will be evicted upon restart.
- Restartable caches should not be removed from the CacheManager during a shutdown.
- If a restartable cache is disposed, the reference to the cache is deleted, but the cache contents remain in memory and on disk. After a restart, the cache contents are once again recovered into memory and on disk. The way to safely dispose of an unused restartable cache is to call clear on the cache and then dispose, so it doesn't take any space in disk or memory.

Key Classes and Methods of the BigMemory Go API

Introduction

BigMemory currently uses Ehcache as its user-facing data access API. See the Ehcache API documentation for details.

The key Ehcache classes used are:

- CacheManager
- Cache
- Element

These classes form the core of the BigMemory Go API. This document introduces these classes, along with other important components of the BigMemory Go API.

When applications use BigMemory Go through the Ehcache API, a CacheManager is instantiated to manage logical data sets (represented as Cache objects in the Ehcache API—though, they may be used to store any kind of data, not just cache data). These data sets contain name-value pairs called Elements.

The logical representations of these key components are actualized mostly through the classes discussed below. These classes' methods provide the main programmatic access to working with Ehcache.

CacheManager

The class `CacheManager` 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 may 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, BigMemory Go 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.

If a CacheManager is part of a cluster, there will also be listener ports which must be unique.

See the API documentation for more information on these methods, including options for passing in configuration. For examples, see Code Samples.

Cache

This 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.

Caches are instantiated from configuration or programmatically using one of the `Cache()` constructors. Certain cache characteristics, such as 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()`, `isSearchable()`, etc.), or perform certain cache-wide operations (for example, flush, load, initialize, dispose, etc.).

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

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()`.

BigMemory Go Search API

Introduction

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

Searchable attributes may 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();
```

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 which 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 ehcache.xml.

```
<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 with:

```
<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 (described in the Defining Attributes section below):

By Configuration

```
<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 out of them. The following example shows this more typical case. Attribute Extractors are explained in more detail in the following section.

```
<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 this Javadoc.

Defining Attributes

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

Attributes are extracted from keys or values during search. This is done using `AttributeExtractors`. Extracted attributes must be one of the following supported types:

- Boolean
- Byte
- Character
- Double

- Float
- Integer
- Long
- Short
- String
- java.util.Date
- java.sql.Date
- Enum

If an attribute cannot be extracted, due to not being found or being 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 later. For example, if an integer value was initially returned for attribute named "Age" by the attribute extractor, then 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 on the `Query` class. So, for example, the attribute for "key" may be referenced in a query by `Query.KEY`. For even greater readability, it is recommended to statically import so that, in this example, you would just 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 which 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 simply 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 one of either "key", "value", or "element". From the starting object a chain of either method calls or field names follows. Method calls and field names can be freely mixed in the chain. Some more examples:

```
<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 allows you to specify the element to inspect and the attribute name, and it returns the attribute value.

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

Provide your extractor class, as shown in the following example:

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

For example, a custom attribute extractor could be passed an Employee object and then extract a specific attribute:

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

If you need to pass state to your custom extractor, you may do so with properties, as shown in the following example:

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

If properties are provided, then 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);
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 that would 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

- In order to use the `DynamicAttributesExtractor`, the cache must be configured to be searchable and dynamically indexable (refer to Making a Cache Searchable above).
- 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, as 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, then this set of attributes will always be queried for extracted values, regardless of whether or not there is a dynamic search attribute extractor 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 will be thrown.

Creating a Query

BigMemory Go Search uses a fluent, object-oriented Query API, following DSL principles, which should be familiar to Java programmers. Here is a simple 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, as shown in this example:

```
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 given them in the configuration. For example:

```
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 may 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 criteria 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, do the following:

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

More complex compound expressions can be further created with extra nesting. See the Expression JavaDoc for a complete list of expressions.

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)`. The full listing of operator shorthand is shown below.

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
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.
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

Making Queries Immutable

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

Obtaining and Organizing Query Results

Queries return a `Results` object which contains a list of objects of class `Result`. Each `Element` in the cache found with a query will be represented as a `Result` object. So if a query finds 350 elements, there will be 350 `Result` objects. An exception to this would be if no keys or attributes are included but aggregators are -- in this case, there will be 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, refer to the Aggregators JavaDoc.

Ordering Results

Query results may be ordered in ascending or descending order by adding an `addOrderBy` clause to the query, which 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

BigMemory Go query results may be grouped similarly to using an SQL GROUP BY statement. The BigMemory GroupBy feature provides the option to group results according to specified attributes by adding 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 like this:

```
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. This necessitates the following rules and considerations when using GroupBy:

- 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 may not be used in a GroupBy clause. This means that `includeKeys()` and `includeValues()` may not 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 will 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 `discard()` to free up resources. In the distributed implementation with Terracotta, resources may be used to hold results for paging or return.

Interrogating Results

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

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

Finding Null (or Not Null) Values

You can find null values (or not-null values) by replacing the null value with the string "NULL" in every element with a null value. A custom attribute extractor can be used to search for the string "NULL" to find null values (or use a not-equal search for not-null values).

If a string cannot be used for element values, then for each field (that may be a search target) set up a related dummy field and assign it a value of "0" (null) or "1" (not null). Queries can then check the dummy field to find nulls or not-nulls.

Sample Application

We have created a simple standalone sample application with few dependencies for you to easily get started with BigMemory Search. You can also check out the source:

```
git clone git://github.com/sharrissf/Ehcache-Search-Sample.git
```

The Ehcache Test Sources page has further examples on how to use each Search feature.

Scripting Environments

BigMemory Search is readily amenable to scripting. The following example shows how to use it with BeanShell:

```
Interpreter i = new Interpreter();
//Auto discover the search attributes and add them to the interpreter's context
Map<String, SearchAttribute> attributes = cache.getCacheConfiguration().getSearchAttributes();
for (Map.Entry<String, SearchAttribute> entry : attributes.entrySet()) {
i.set(entry.getKey(), cache.getSearchAttribute(entry.getKey()));
LOG.info("Setting attribute " + entry.getKey());
}
//Define the query and results. Add things which would be set in the GUI i.e.
//includeKeys and add to context
```

```
Query query = cache.createQuery().includeKeys();
Results results = null;
i.set("query", query);
i.set("results", results);
//This comes from the freeform text field
String userDefinedQuery = "age.eq(35)";
//Add on the things that we need
String fullQueryString = "results = query.addCriteria(" + userDefinedQuery + ").execute()";
i.eval(fullQueryString);
results = (Results) i.get("results");
assertTrue(2 == results.size());
for (Result result : results.all()) {
LOG.info("" + result.getKey());
}
```

Implementation and Performance

BigMemory uses a Search index that is maintained at the local node. The index is stored under a directory in the DiskStore and is available whether or not persistence is enabled. Any overflow from the on-heap tier of the cache, whether to the off-heap tier or to the disk tier, is searched using indexes.

Search operations perform in O(log(n)) time. For tips that can aid performance, refer to Best Practices.

For caches that are on-heap only, the implementation does not use indexes. Instead, it performs a fast iteration of the cache, relying on the very fast access to do the equivalent of a table scan for each query. Each element in the cache is only visited once. Attributes are not extracted ahead of time. They are done during query execution.

Search operations perform in O(n) time. Check out this Maven-based performance test showing performance of an on-heap-only search. The test shows search performance of an average of representative queries at 4.6 ms for a 10,000 entry cache, and 427 ms for a 1,000,000 entry cache. Accordingly, this implementation is suitable for development and testing.

Best Practices for Optimizing Searches

1. Construct searches wisely by including only the data that is actually required.
    - Only use `includeKeys()` and/or `includeAttribute()` if those values are actually required for your application logic.
    - If you don't 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, then don't use `includeValues()` in the original 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: `includeKeys()` and `includeValues()` have lazy deserialization, which means that keys and values are de-serialized only when `result.getKey()` or `result.getValue()` is called. This means there is a time cost only when the key is needed. However, there is still some time cost with `includeKeys()` and `includeValues()`, so consider carefully when constructing your queries.
2. Searchable keys and values are automatically indexed by default. If you will not be including them in your query, turn off automatic indexing with the following:

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

3. Limit the size of the results set with `query.maxResults(int number_of_results)`. Another recommendation for managing the size of the result set is to use a built-in Aggregator function to return a summary statistic (see the `net.sf.ehcache.search.aggregator` package in this Javadoc).
4. Make your search as specific as possible. Queries with "ILike" criteria and fuzzy (wildcard) searches may take longer than more specific queries. Also, if you are using a wildcard, try making it the trailing part of the string instead of the leading part (`"321*"` instead of `"*123"`). If you want leading wildcard searches, then you should create a `<searchAttribute>` with the string value reversed in it, so that your query can use the trailing wildcard instead.
5. 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))`.
6. Index dates as integers. This can save time and may even be faster if you have to do a conversion later on.

Concurrency Notes

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

Index Updating

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

For caches with concurrency control, an index will not reflect the new state of the cache until a `commit` has been called.

Query Results

There are several ways unexpected results could present:

- A search returns an Element reference which no longer exists.
- Search criteria select an Element, but the Element has been updated and a new Search would no longer match the Element.
- Aggregators, such as `sum()`, might disagree with the same calculation done by redoing the calculation yourself by re-accessing the cache for each key and repeating the calculation.
- Because the cache is always updated before the search index, it is possible that a value reference may refer to a value that has been removed from the cache. If this happens, the value will be null but the key and attributes which were supplied by the now stale cache index will be non-null. Because values in Ehcache are also allowed to be null, you cannot tell whether your value is null because it has been removed from the cache since 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.

Transactions in Ehcache

Introduction

BigMemory Go suports Global Transactions, with "xa_strict" and "xa" modes, and Local Transactions with "local" mode.

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.

Transactional Methods

The following methods require a transactional context to run:

- put()
- get()
- getQuiet()
- remove()
- getKeys()
- getSize()
- containsKey()
- removeAll()
- putWithWriter()
- removeWithWriter()
- putIfAbsent()
- removeElement()
- replace()

This list is applies to all transactional modes.

All other methods work non-transactionally but can be called on a transactional cache, either within or outside of a transactional context.

Change Visibility

The isolation level offered in BigMemory's 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 `COMMIT` has been called, reads such as by `cache.get(...)` by other transactions return the old copy. Reads do not block.

When to use transactional modes

Transactional modes are a powerful extension of Ehcache allowing 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 orJMS, 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

The objects you are going to store in your transactional cache must:

- implement `java.io.Serializable` — This is required to store cached objects when the cache is clustered with Terracotta but it is also required by the copy-on-read / copy-on-write mechanism used to implement isolation.
- override `equals` and `hashcode` — Those must be overridden because the transactional stores rely on element value comparison. See `ElementValueComparator` and the `elementValueComparator` configuration setting.

Configuration

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". Enabling a cache for "xa_strict" transactions is shown in the following example:

```
<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, then 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 (see Transactions Managers).
- Configuring a `<tx:method>` with read-only=true could be problematic with certain transaction managers such as WebSphere.

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. The differences are explained in the sections below.

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 the TransactionManager 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 transactions data, therefore recovery errors after a crash may 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. In accordance with XA, data that has not yet been `prepared` is discarded. The recovery guarantee differs depending on the XA mode.

xa Mode

With "xa", 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", the cache always responds to the TransactionManager'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 Apps

We have three sample applications showing how to use XA with a variety of technologies.

XA Sample App

This sample application uses JBoss application server. It shows an example using User managed transactions. Although most people will use JTA from within 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);
}
```

The source code for the demo can be checked out from the Terracotta Forge. A README.txt explains how to get the sample app going.

XA Banking Application

This application is to show a real world scenario. A Web app reads <account transfer> messages from a queue and tries to execute these account transfers. With JTA turned on, failures are rolled back so that the cached account balance is always the same as the true balance summed from the database. This app is a Spring-based

XA Banking Application

Java web app 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 TransactionManager. 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`

The source code for the demo can be checked out from the Terracotta Forge. A README.txt explains how to get the sample app going.

Transaction Managers

Automatically Detected Transaction Managers

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

- 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 above list or you wish 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`:

```
<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 TransactionManager in GlassFish v3:

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

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, then you need to 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.

Introduction 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, then a `TransactionTimeoutException` will be thrown. To return the cache to a consistent state you need to 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. If not set, 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 could 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

This example shows a transaction which 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

Managing Contention

If two transactions, either standalone or across the cluster, attempt to perform a cache operation on the same element then 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 that 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. The normal cluster semantics are used. Because transactions only work with consistency=strong caches, the first transaction is the thread that manages to atomically place a soft lock on the Element. (This is done with the CAS based putIfAbsent and replace methods.)

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.

Accordingly, "xa_strict" should only be used where full guarantees are required, otherwise one of the other modes should be used.

FAQ

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.

Is IBM Websphere Transaction Manager supported?

Mostly. "xa_strict" is not supported due to each version of Websphere being a custom implementation; that is, there is no stable interface to implement against. However, "xa", which uses TransactionManager callbacks and "local" 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`. Simply get `java:comp/websphere/ExtendedJTATransaction` from JNDI, cast that to `com.ibm.websphere.jtaextensions.ExtendedJTATransaction` and call the `registerSynchronizationCallbackForCurrentTran` method. If you succeed, then Ehcache should too.

How do transactions interact with Write-behind and Write-through caches?

If your transactional enabled cache is being used with a writer, write operations are queued until transaction commit time. Solely a 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, as 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 thrown exception during these write operations would cause the transaction to be rolled back by having UserTransaction.commit() throw a RollbackException.

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 transactional Ehcache?

WebLogic uses an optimization that is not supported by our 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 `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>
```

How do I make Atomikos work with the Ehcache "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 won't work. You have to configure the following property to make it work:

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

Explicit Locking

Introduction

BigMemory Go's 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 behaviour 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.

The 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);
```

Example

Here is a brief example:

```
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 it 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.

Blocking Cache and Self-Populating Cache

Introduction

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. Let's say 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, 10 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 about Blocking Cache, refer to this Javadoc.

SelfPopulatingCache

You want to use the BlockingCache, but the requirement to always release the lock creates gnarly code. You also want to think about what you are doing without thinking about the caching. Enter the SelfPopulatingCache, a subclass of BlockingCache. The name SelfPopulatingCache is synonymous with Pull-through cache, which is a common caching term. SelfPopulatingCache uses a `CacheEntryFactory`, that given a key, knows how to populate the entry. Note: JCache-inspired getWithLoader and getAllWithLoader, available directly in `Ehcache` and which work with a `CacheLoader`, may be used as an alternative to SelfPopulatingCache.

For more information about Self-populating Cache, refer to this Javadoc.

Cache Decorators

Introduction

BigMemory Go 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 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.

Creating a Decorator

Programmatically

Cache decorators are created as follows:

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

The class must implement Ehcache.

By Configuration

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

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

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

Adding decorated caches to the CacheManager

Having created a decorator programmatically it is generally useful to put it in a place where multiple threads may 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 overwridden Ehcache methods will take on new behaviours 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 {

Built-in Decorators

BlockingCache

A blocking decorator for an Ehcache, backed by a {@link Ehcache}.

It 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 is useful for constructing read-through or self-populating caches. BlockingCache is used by `CachingFilter`.

SelfPopulatingCache

A selfpopulating 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

These are decorated. See Cache Exception Handlers for full details.

Event Listeners

CacheManager Event Listeners

BigMemory Go's Ehcache implementation includes CacheManager event listeners. These listeners allow implementers to register callback methods that will be executed when a CacheManager event occurs. Cache 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.

Configuration

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 n
* @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.

Cache Event Listeners

BigMemory Go's Ehcache implementation also includes 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.

Configuration

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` which is used to create 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 the 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. See the page on Classloading for details on how the loading of these classes will be done.

Adding a Listener Programmatically

To add a listener programmatically, follow this example:

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

Cache Exception Handlers

Introduction

BigMemory Go's Ehcache implementation includes Cache 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.

CacheExceptionHandlers may be set either declaratively in the ehcache.xml configuration file, or programmatically.

Declarative Configuration

Cache event listeners are configured per cache. Each cache can have at most one exception handler. An exception handler is configured by adding a `cacheExceptionHandlerFactory` element 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. 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);
}
```

The implementations need to be placed in the classpath accessible to Ehcache. See the page on Classloading for details on how classloading of these classes will be done.

Programmatic Configuration

The following example shows how to add exception handling to a cache, and then adding 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);
```

Cache Extensions

Introduction

BigMemory Go's Ehcache implementation includes 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. See 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 customization.

Declarative Configuration

Cache extensions are configured per cache. Each cache can have zero or more. A `CacheExtension` is configured 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 it's
* 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. See the page on Classloading for details on how class loading of these classes will be done.

Programmatic Configuration

Cache extensions may also be programmatically added to a Cache as shown.

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

Cache Eviction Algorithms

Introduction

A cache eviction algorithm is a way of deciding which element to evict when the cache is full. In BigMemory Go's Ehcache, the memory store and the off-heap store may be limited in size. (For information about sizing, refer to Sizing BigMemory Tiers). When these stores get full, elements are evicted. The eviction algorithms in Ehcache determine which elements are evicted. The default is 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 will be removed. The disk store size by default is unbounded. But a maximum size can be set (see Cache Configuration Sizing Attributes for more information). If the disk store is full, then adding an element will cause one to be evicted unless it is unbounded. The disk store eviction algorithm is not configurable. It uses LFU.

Provided MemoryStore 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 it a theoretical result that is unimplementable without domain knowledge. The Least Recently Used ("LRU") algorithm is often used as a proxy. It works pretty 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.

Altogether 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 Least Frequently Used.

The oldest element is the Less Recently Used (LRU) 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 use follows a Pareto distribution, this algorithm may give better results than LRU.

LFU is an algorithm unique to Ehcache. 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

BigMemory Go allows you to plugin 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, evict an Element if it has been hit more than 10 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 it is full.

Class Loading and Class Loaders

Introduction

Class loading, within the plethora of environments that Ehcache can be running, could be complex. But with BigMemory Go, 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).

Logging

Introduction

BigMemory Go uses the SLF4J logging facade, so you can plug in your own logging framework. This page covers Ehcache logging. For more information about slf4j in general, refer to the SLF4J site.

SLF4J Logging

With SLF4J, users must choose a concrete logging implementation at deploy time. The options include Maven and the download kit.

Concrete Logging Implementation Use in Maven

The maven dependency declarations are reproduced here for convenience. Add *one* of these to your Maven POM.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jdk14</artifactId>
    <version>1.5.8</version>
</dependency>
 <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.5.8</version>
</dependency>
 <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.5.8</version>
</dependency>
```

Concrete Logging Implemenation Use in the Download Kit

The slf4j-api and slf4j-jdk14 jars are in the kit along with the BigMemory Go jars so that, if the app does not already use SLF4J, you have everything you need. Additional concrete logging implementations can be downloaded from SLF4J website.

Recommended Logging Levels

BigMemory Go seeks to trade off informing production support developers or important messages and cluttering the log. ERROR messages should not occur in normal production and indicate that action should be taken.

WARN messages generally indicate a configuration change should be made or an unusual event has occurred. DEBUG and TRACE messages are for development use. All DEBUG level statements are surrounded with a guard so that no performance cost is incurred unless the logging level is set. Setting the logging level to DEBUG should provide more information on the source of any problems. Many logging systems enable a logging level change to be made without restarting the application.

Management and Monitoring using JMX

Introduction

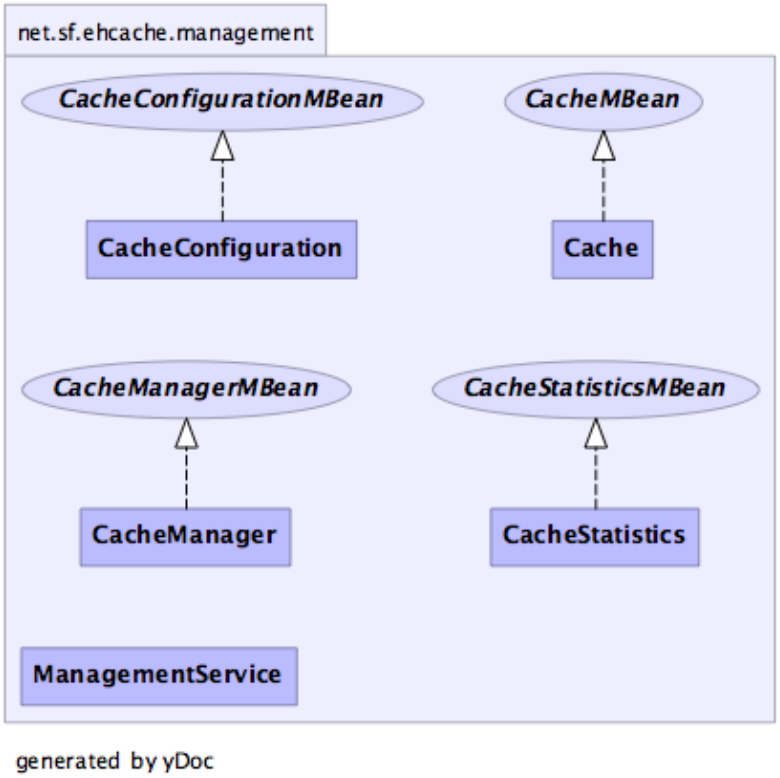
JMX creates a standard way of instrumenting classes and making them available to a management and monitoring infrastructure. This provides an alternative to the Terracotta Management Console for custom or third-party tools.

JMX Overview

The `net.sf.ehcache.management` package contains MBeans and a `ManagementService` for JMX management of BigMemory Go. It is in a separate package so that JMX libraries are only required if you wish to use it - there is no leakage of JMX dependencies into the core Ehcache package.

Use `net.sf.ehcache.management.ManagementService.registerMBeans(...)` static method to register a selection of MBeans to the MBeanServer provided to the method. If you wish to monitor Ehcache but not use JMX, just use the existing public methods on `Cache` and `CacheStatistics`.

The Management package is illustrated in the follwing image.



MBeans

BigMemory Go uses Standard MBeans. MBeans are available for the following:

- CacheManager
- Cache
- CacheConfiguration
- CacheStatistics

All MBean attributes are available to a local MBeanServer. The CacheManager MBean allows traversal to its collection of Cache MBeans. Each Cache MBean likewise allows traversal to its CacheConfiguration MBean and its CacheStatistics MBean.

JMX Remoting

The Remote API allows connection from a remote JMX Agent to an MBeanServer via an `MBeanServerConnection`. Only `Serializable` attributes are available remotely. The following Ehcache MBean attributes are available remotely:

- limited CacheManager attributes
- limited Cache attributes
- all CacheConfiguration attributes
- all CacheStatistics attributes

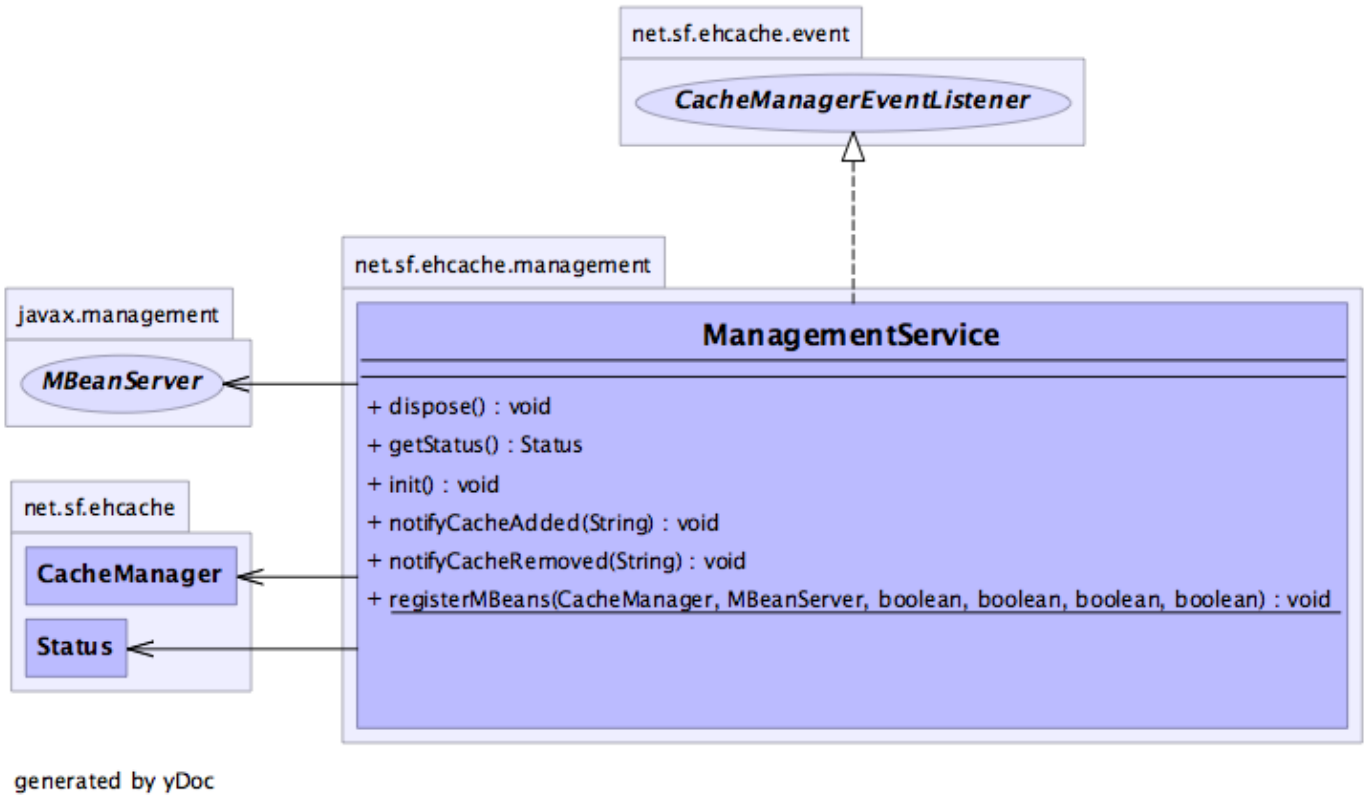
Most attributes use built-in types. To access all attributes, you need to add ehcache.jar to the remote JMX client's classpath. For example, `jconsole -J-Djava.class.path=ehcache.jar`.

`ObjectName` naming scheme

- CacheManager - "net.sf.ehcache:type=CacheManager,name=<CacheManager>"
- Cache - "net.sf.ehcache:type=Cache,CacheManager=<cacheManagerName>,name=<cacheName>"
- CacheConfiguration
- "net.sf.ehcache:type=CacheConfiguration,CacheManager=<cacheManagerName>,name=<cacheName>"
- CacheStatistics - "net.sf.ehcache:type=CacheStatistics,CacheManager=<cacheManagerName>,name=<cacheName>"

The Management Service

The `ManagementService` class is the API entry point.



There is only one method, `ManagementService.registerMBeans` which is used to initiate JMX registration of a CacheManager's instrumented MBeans. The `ManagementService` is a `CacheManagerEventListener` and is therefore notified of any new Caches added or disposed and updates the MBeanServer appropriately. Once initiated the MBeans remain registered in the MBeanServer until the CacheManager shuts down, at which time the MBeans are deregistered. This behaviour ensures correct behavior in application servers where applications are deployed and undeployed.

```
/**
* This method causes the selected monitoring options to be be registered
* with the provided MBeanServer for caches in the given CacheManager.
*
* While registering the CacheManager enables traversal to all of the other
*  items,
* this requires programmatic traversal. The other options allow entry points closer
* to an item of interest and are more accessible from JMX management tools like JConsole.
* Moreover CacheManager and Cache are not serializable, so remote monitoring is not
* possible * for CacheManager or Cache, while CacheStatistics and CacheConfiguration are.
* Finally * CacheManager and Cache enable management operations to be performed.
*
* Once monitoring is enabled caches will automatically added and removed from the
* MBeanServer * as they are added and disposed of from the CacheManager. When the
* CacheManager itself * shutsdown all registered MBeans will be unregistered.
*
* @param cacheManager the CacheManager to listen to
* @param mBeanServer the MBeanServer to register MBeans to
* @param registerCacheManager Whether to register the CacheManager MBean
* @param registerCaches Whether to register the Cache MBeans
* @param registerCacheConfigurations Whether to register the CacheConfiguration MBeans
* @param registerCacheStatistics Whether to register the CacheStatistics MBeans
*/
public static void registerMBeans(
```

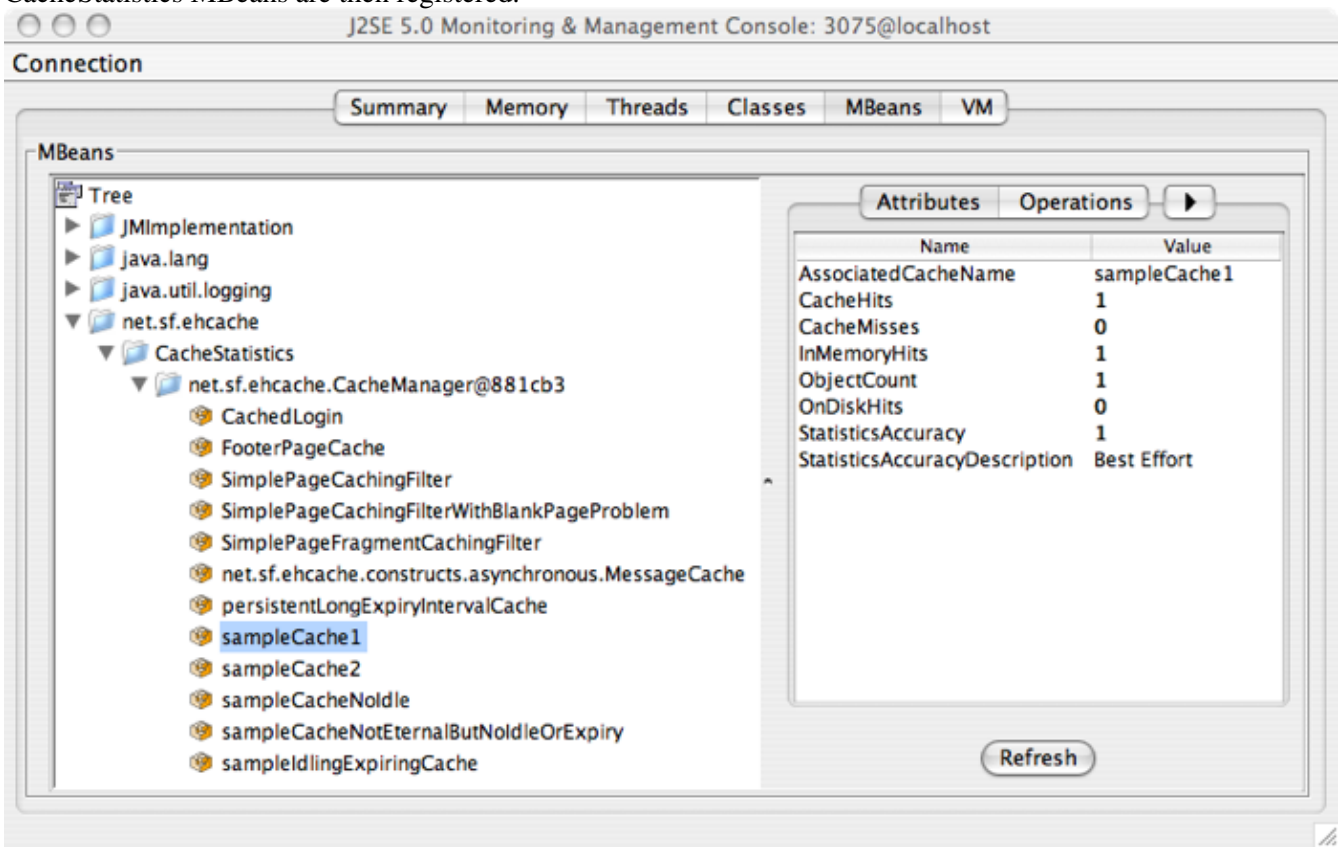
```
    net.sf.ehcache.CacheManager cacheManager,
    MBeanServer mBeanServer,
    boolean registerCacheManager,
    boolean registerCaches,
    boolean registerCacheConfigurations,
    boolean registerCacheStatistics) throws CacheException {
```

JConsole Example

This example shows how to register CacheStatistics in the JDK platform MBeanServer, which works with the JConsole management agent.

```
CacheManager manager = new CacheManager();
MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
ManagementService.registerMBeans(manager, mBeanServer, false, false, false, true);
```

CacheStatistics MBeans are then registered.



CacheStatistics MBeans in JConsole

Performance

Collection of cache statistics is not entirely free of overhead, however, the statistics API switches on/off automatically according to usage. If you need few statistics, you incur little overhead; on the other hand, as you use more statistics, you can incur more. Statistics are off by default.

Shutting Down BigMemory

Introduction

BigMemory is shut down through the Ehcache API. Note that Hibernate automatically shuts down its Ehcache `CacheManager`. The recommended way to shutdown the Ehcache is:

- to call `CacheManager.shutdown()`
- in a web app, register the Ehcache `ShutdownListener`

Though not recommended, Ehcache also lets you register a JVM shutdown hook.

ServletContextListener

Ehcache provides a ServletContextListener that shuts down the CacheManager. Use this to shut down Ehcache automatically, when the web application is shut down. To receive notification events, this class must be configured in the deployment descriptor for the web application. To do so, add the following to web.xml in your web application:

```
<listener>
  <listener-class>net.sf.ehcache.constructs.web.ShutdownListener</listener-class>
</listener>
```

The Shutdown Hook

The Ehcache CacheManager can optionally register a shutdown hook. To do so, set the system property `net.sf.ehcache.enableShutdownHook=true`. This will shut down the CacheManager when it detects the Virtual Machine shutting down and it is not already shut down.

Use the shutdown hook when the CacheManager is not already being shutdown by a framework you are using, or by your application.

Note: Shutdown hooks are inherently problematic. The JVM is shutting down, so sometimes things that can never be null are. Ehcache guards against as many of these as it can, but the shutdown hook should be the last option to use.

The shutdown hook is on CacheManager. It simply calls the shutdown method. The sequence of events is:

- call dispose for each registered CacheManager event listener.
- call dispose for each Cache. Each Cache will:
- shutdown the MemoryStore. The MemoryStore will flush to the DiskStore.
- shutdown the DiskStore. If the DiskStore is persistent ("localRestartable"), it will write the entries and index to disk.
- shutdown each registered CacheEventListener.
- set the Cache status to shutdown, preventing any further operations on it.
- set the CacheManager status to shutdown, preventing any further operations on it.

The shutdown hook runs when:

- A program exists normally. For example, `System.exit()` is called, or the last non-daemon thread exits.
- the Virtual Machine is terminated, e.g. CTRL-C. This corresponds to `kill -SIGTERM pid` or `kill -15 pid` on Unix systems.

The shutdown hook will not run when:

- the Virtual Machine aborts.
- A SIGKILL signal is sent to the Virtual Machine process on Unix systems, e.g. `kill -SIGKILL pid` or `kill -9 pid`.
- A `TerminateProcess` call is sent to the process on Windows systems.

Dirty Shutdown

If Ehcache is shutdown dirty, all in-memory data will be retained if BigMemory is configured for restartability. For more information, refer to Fast Restartability.

Using Hibernate and BigMemory Go

Introduction

Big Memory Go easily integrates with the Hibernate Object/Relational persistence and query service. Gavin King, the maintainer of Hibernate, is also a committer to the BigMemory Go's Ehcache project. This ensures BigMemory Go will remain a first class data store for Hibernate. Configuring BigMemory Go for Hibernate is simple. The basic steps are:

- Download and install BigMemory Go into your project
- Configure BigMemory Go as a cache provider in your project's Hibernate configuration.
- Configure second-level caching in your project's Hibernate configuration.
- Configure Hibernate caching for each entity, collection, or query you wish to cache.
- Configure the `ehcache.xml` file as necessary for each entity, collection, or query configured for caching.

For more information regarding cache configuration in Hibernate see the Hibernate documentation.

Download and Install

The Hibernate provider is in the ehcache-ee module provided in the BigMemory Go kit.

Build with Maven

Dependency versions vary with the specific kit you intend to use. Since kits are guaranteed to contain compatible artifacts, find the artifact versions you need by downloading a kit. Configure or add the following repository to your build (pom.xml):

```
<repository>
  <id>terracotta-releases</id>
  <url>http://www.terracotta.org/download/reflector/releases</url>
  <releases><enabled>true</enabled></releases>
  <snapshots><enabled>false</enabled></snapshots>
</repository>
```

Configure or add the Ehcache and BigMemory modules defined by the following dependency to your build (pom.xml):

```
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <artifactId>ehcache-ee</artifactId>
  <version>${ehcacheVersion}</version>
</dependency>
<dependency>
  <groupId>org.terracotta.bigmemory</groupId>
  <artifactId>bigmemory</artifactId>
  <version>${bigmemoryVersion}</version>
</dependency>
```

For the Hibernate-Ehcache integration, add the following dependency:

Build with Maven

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-ehcache</artifactId>
  <version>${hibernateVersion}</version>
</dependency>
```

For example, the Hibernate-Ehcache integration dependency for Hibernate 4.0.0 is:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-ehcache</artifactId>
  <version>4.0.0</version>
</dependency>
```

NOTE: Some versions of hibernate-ehcache may have a dependency on a specific version of Ehcache. Check the hibernate-ehcache POM for more information.

Configure BigMemory Go as the Second-Level Cache Provider

To configure BigMemory Go as a Hibernate second-level cache, set the region factory property to one of the following in the Hibernate configuration. Hibernate configuration is configured either via hibernate.cfg.xml, hibernate.properties or Spring. The format given is for hibernate.cfg.xml.

Hibernate 3.3

For instance creation:

```
<property name="hibernate.cache.region.factory_class">
         net.sf.ehcache.hibernate.EhCacheRegionFactory</property>
```

To force Hibernate to use a singleton of Ehcache CacheManager:

```
<property name="hibernate.cache.region.factory_class">
         net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory</property>
```

Hibernate 4.x

For Hibernate 4, use `org.hibernate.cache.ehcache.EhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.EhCacheRegionFactory`, or `org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory`.

Enable Second-Level Cache and Query Cache Settings

In addition to configuring the second-level cache provider setting, you will need to turn on the second-level cache (by default it is configured to off - "false" - by Hibernate). This is done by setting the following property in your hibernate config:

```
<property name="hibernate.cache.use_second_level_cache">true</property>
```

You may also want to turn on the Hibernate query cache. This is done by setting the following property in your hibernate config:

```
<property name="hibernate.cache.use_query_cache">true</property>
```

Optional

The following settings or actions are optional.

Configuration Resource Name

The `configurationResourceName` property is used to specify the location of the Ehcache configuration file to be used with the given Hibernate instance and cache provider/region-factory. The resource is searched for in the root of the classpath. It is used to support multiple CacheManagers in the same VM. It tells Hibernate which configuration to use. An example might be "ehcache-2.xml". When using multiple Hibernate instances it is therefore recommended to use multiple non-singleton providers or region factories, each with a dedicated Ehcache configuration resource.

```
net.sf.ehcache.configurationResourceName=/name_of_ehcache.xml
```

Set the Hibernate cache provider programmatically

The provider can also be set programmatically in Hibernate by adding necessary Hibernate property settings to the configuration before creating the SessionFactory:

```
Configuration.setProperty("hibernate.cache.region.factory_class",
                      "net.sf.ehcache.hibernate.EhCacheRegionFactory")
```

For Hibernate 4, use `org.hibernate.cache.ehcache.EhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.EhCacheRegionFactory`.

Putting it all together

If you are enabling both second-level caching and query caching, then your hibernate config file should contain the following:

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property name="hibernate.cache.region.factory_class">
    net.sf.ehcache.hibernate.EhCacheRegionFactory</property>
```

An equivalent Spring configuration file would contain:

```
<prop key="hibernate.cache.use_second_level_cache">true</prop>
<prop key="hibernate.cache.use_query_cache">true</prop>
<prop key="hibernate.cache.region.factory_class">
    net.sf.ehcache.hibernate.EhCacheRegionFactory</prop>
```

For Hibernate 4, use `org.hibernate.cache.ehcache.EhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.EhCacheRegionFactory`.

Configure Hibernate Entities to use Second-Level Caching

In addition to configuring the Hibernate second-level cache provider, Hibernate must also be told to enable caching for entities, collections, and queries. For example, to enable cache entries for the domain object com.somecompany.someproject.domain.Country there would be a mapping file something like the following:

```
<hibernate-mapping>
<class
name="com.somecompany.someproject.domain.Country"
table="ut_Countries"
dynamic-update="false"
dynamic-insert="false"
>
...
</class>
</hibernate-mapping>
```

To enable caching, add the following element.

```
<cache usage="read-write|nonstrict-read-write|read-only" />
```

For example:

```
<hibernate-mapping>
<class
name="com.somecompany.someproject.domain.Country"
table="ut_Countries"
dynamic-update="false"
dynamic-insert="false"
>
 <cache usage="read-write" />
...
</class>
</hibernate-mapping>
```

This can also be achieved using the @Cache annotation, e.g.

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Country {
...
}
```

Definition of the different cache strategies

read-only

Caches data that is never updated.

nonstrict-read-write

Caches data that is sometimes updated without ever locking the cache. If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database. Configure your cache timeout accordingly!

Definition of the different cache strategies

read-write

Caches data that is sometimes updated while maintaining the semantics of "read committed" isolation level. If the database is set to "repeatable read", this concurrency strategy almost maintains the semantics. Repeatable read isolation is compromised in the case of concurrent writes.

Configure

Because the `ehcache.xml` file has a defaultCache, caches will always be created when required by Hibernate. However more control can be exerted by specifying a configuration per cache, based on its name. In particular, because Hibernate caches are populated from databases, there is potential for them to get very large. This can be controlled by capping their maxEntriesLocalHeap and specifying whether to swap to disk beyond that. Hibernate uses a specific convention for the naming of caches of Domain Objects, Collections, and Queries.

Domain Objects

Hibernate creates caches named after the fully qualified name of Domain Objects. So, for example to create a cache for com.somecompany.someproject.domain.Country create a cache configuration entry similar to the following in ehcache.xml.

```
   <?xml version="1.0" encoding="UTF-8"?>
<ehcache>
 <cache
    name="com.somecompany.someproject.domain.Country"
    maxEntriesLocalHeap="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    <persistence strategy="localTempSwap"/>
 />
</ehcache>
```

Hibernate CacheConcurrencyStrategy

read-write, nonstrict-read-write and read-only policies apply to Domain Objects.

Collections

Hibernate creates collection caches named after the fully qualified name of the Domain Object followed by "." followed by the collection field name. For example, a Country domain object has a set of advancedSearchFacilities. The Hibernate doclet for the accessor looks like:

```
/**
* Returns the advanced search facilities that should appear for this country.
* @hibernate.set cascade="all" inverse="true"
* @hibernate.collection-key column="COUNTRY_ID"
* @hibernate.collection-one-to-many class="com.wotif.jaguar.domain.AdvancedSearchFacility"
* @hibernate.cache usage="read-write"
*/
public Set getAdvancedSearchFacilities() {
return advancedSearchFacilities;
}
```

You need an additional cache configured for the set. The ehcache.xml configuration looks like:

```
   <?xml version="1.0" encoding="UTF-8"?>
<ehcache>
 <cache name="com.somecompany.someproject.domain.Country"
    maxEntriesLocalHeap="50"
    eternal="false"
    timeToLiveSeconds="600"
    <persistence strategy="localTempSwap"/>
/>
 <cache
name="com.somecompany.someproject.domain.Country.advancedSearchFacilities"
    maxEntriesLocalHeap="450"
    eternal="false"
    timeToLiveSeconds="600"
    <persistence strategy="localTempSwap"/>
/>
</ehcache>
```

Hibernate CacheConcurrencyStrategy

read-write, nonstrict-read-write and read-only policies apply to Domain Object collections.

Queries

Hibernate allows the caching of query results using two caches.

StandardQueryCache

This cache is used if you use a query cache without setting a name. A typical ehcache.xml configuration is:

```
<cache
name="org.hibernate.cache.StandardQueryCache"
maxEntriesLocalHeap="5"
eternal="false"
timeToLiveSeconds="120"
<persistence strategy="localTempSwap"/>
/>
```

UpdateTimestampsCache

Tracks the timestamps of the most recent updates to particular tables. It is important that the cache timeout of the underlying cache implementation be set to a higher value than the timeouts of any of the query caches. In fact, it is recommend that the the underlying cache not be configured for expiry at all. A typical ehcache.xml configuration is:

```
<cache
name="org.hibernate.cache.UpdateTimestampsCache"
maxEntriesLocalHeap="5000"
eternal="true"
<persistence strategy="localTempSwap"/>
/>
```

Named Query Caches

In addition, a QueryCache can be given a specific name in Hibernate using Query.setCacheRegion(String name). The name of the cache in ehcache.xml is then the name given in that method. The name can be whatever you want, but by convention you should use "query." followed by a descriptive name. E.g.

```
<cache name="query.AdministrativeAreasPerCountry"
maxEntriesLocalHeap="5"
eternal="false"
timeToLiveSeconds="86400"
<persistence strategy="localTempSwap"/>
/>
```

Using Query Caches

For example, let's say we have a common query running against the Country Domain. Code to use a query cache follows:

```
public List getStreetTypes(final Country country) throws HibernateException {
final Session session = createSession();
try {
   final Query query = session.createQuery(
    "select st.id, st.name"
   + " from StreetType st "
   + " where st.country.id = :countryId "
   + " order by st.sortOrder desc, st.name");
   query.setLong("countryId", country.getId().longValue());
   query.setCacheable(true);
   query.setCacheRegion("query.StreetTypes");
   return query.list();
} finally {
   session.close();
}
}
```

The `query.setCacheable(true)` line caches the query. The `query.setCacheRegion("query.StreetTypes")` line sets the name of the Query Cache. Alex Miller has a good article on the query cache here.

Hibernate CacheConcurrencyStrategy

None of read-write, nonstrict-read-write and read-only policies apply to Domain Objects. Cache policies are not configurable for query cache. They act like a non-locking read only cache.

Demo App

We have demo application showing how to use the Hibernate CacheRegionFactory.

Hibernate Tutorial

Check out from the Terracotta Forge.

Performance Tips

Session.load

`Session.load` will always try to use the cache.

Session.find and Query.find

`Session.find` does not use the cache for the primary object. Hibernate will try to use the cache for any associated objects. `Session.find` does however cause the cache to be populated. `Query.find` works in exactly the same way. Use these where the chance of getting a cache hit is low.

Session.iterate and Query.iterate

`Session.iterate` always uses the cache for the primary object and any associated objects. `Query.iterate` works in exactly the same way. Use these where the chance of getting a cache hit is high.

FAQ

If I'm using BigMemory Go with my app and with Hibernate for second-level caching, should I try to use the CacheManager created by Hibernate for my app's caches?

While you could share the resource file between the two CacheManagers, a clear separation between the two is recommended. Your app may have a different lifecycle than Hibernate, and in each case your CacheManager Automatic Resource Control (ARC) settings may need to be different.

Should I use the provider in the Hibernate distribution or in BigMemory Go's Ehcache?

Since Hibernate 2.1, Hibernate has included an Ehcache `CacheProvider`. That provider is periodically synced up with the provider in the Ehcache Core distribution. New features are generally added in to the Ehcache Core provider and then the Hibernate one.

What is the relationship between the Hibernate and Ehcache projects?

Gavin King and Greg Luck cooperated to create Ehcache and include it in Hibernate. Since 2009, Greg Luck has been a committer on the Hibernate project so as to ensure Ehcache remains a first-class 2nd level cache for Hibernate.

Does BigMemory Go support the transactional strategy?

Yes. It was introduced in Ehcache 2.1.

Why do certain caches sometimes get automatically cleared by Hibernate?

Whenever a `Query.executeUpdate()` is run, for example, Hibernate invalidates affected cache regions (those corresponding to affected database tables) to ensure that no data stale data is cached. This should also happen whenever stored procedures are executed.

For more information, see this Hibernate bug report.

How are Hibernate Entities keyed?

Hibernate identifies cached Entities via an object id. This is normally the primary key of a database row.

Are compound keys supported?

Yes.

I am getting this error message: An item was expired by the cache while it was locked. What is it?

Soft locks are implemented by replacing a value with a special type that marks the element as locked, thus indicating to other threads to treat it differently to a normal element. This is used in the Hibernate Read/Write strategy to force fall-through to the database during the two-phase commit - since we don't know exactly what should be returned by the cache while the commit is in process (but the database does). If a soft-locked Element is evicted by the cache during the two-phase commit, then once the two-phase commit completes, the cache will fail to update (since the soft-locked Element was evicted) and the cache entry will be reloaded from the database on the next read of that object. This is obviously non-fatal, but could cause a small rise in database load.

So, in summary the Hibernate messages are not problematic. The underlying cause is the probabilistic evictor can theoretically evict recently loaded items. You can also use the deterministic evictor to avoid this problem. Specify the `java -Dnet.sf.ehcache.use.classic.lru=true` system property to turn on classic LRU which contains a deterministic evictor.

Using Coldfusion and BigMemory Go

Introduction

ColdFusion ships with BigMemory Go's Ehcache. The ColdFusion community has actively engaged with Ehcache and have put out lots of great blogs. Here are three to get you started. For a short introduction, see Raymond Camden's blog. For more in-depth analysis, see Rob Brooks-Bilson's nine-part Blog Series or 14 days of ColdFusion caching, by Aaron West, covering a different topic each day.

Example Integration

To integrate BigMemory Go with ColdFusion, first add the BigMemory Go jars to your web application lib directory.

The following code demonstrates how to call Ehcache from ColdFusion. It will cache a CF object in Ehcache and the set expiration time to 30 seconds. If you refresh the page many times within 30 seconds, you will see the data from cache. After 30 seconds, you will see a cache miss, then the code will generate a new object and put in cache again.

```
<CFOBJECT type="JAVA" class="net.sf.ehcache.CacheManager" name="cacheManager">
<cfset cache=cacheManager.getInstance().getCache("MyBookCache")>
<cfset myBookElement=#cache.get("myBook")#>
<cfif IsDefined("myBookElement")>
   <cfoutput>
    myBookElement: #myBookElement#<br />
   </cfoutput>
   <cfif IsStruct(myBookElement.getObjectValue())>
           <strong>Cache Hit</strong><p/>
           <!-- Found the object from cache -->
           <cfset myBook = #myBookElement.getObjectValue()#>
   </cfif>
</cfif>
<cfif IsDefined("myBook")>
<cfelse>
<strong>Cache Miss</strong>
   <!-- object not found in cache, go ahead create it -->
   <cfset myBook = StructNew()>
   <cfset a = StructInsert(myBook, "cacheTime", LSTimeFormat(Now(), "hh:mm:sstt"), 1)>
   <cfset a = StructInsert(myBook, "title", "EhCache Book", 1)>
   <cfset a = StructInsert(myBook, "author", "Greg Luck", 1)>
   <cfset a = StructInsert(myBook, "ISBN", "ABCD123456", 1)>
   <CFOBJECT type="JAVA" class="net.sf.ehcache.Element" name="myBookElement">
   <cfset myBookElement.init("myBook", myBook)>
   <cfset cache.put(myBookElement)>
</cfif>
<cfoutput>
Cache time: #myBook["cacheTime"]#<br />
Title: #myBook["title"]#<br />
Author: #myBook["author"]#<br />
ISBN: #myBook["ISBN"]#
</cfoutput>
```

Using Spring and BigMemory Go

Introduction

BigMemory Go's Ehcache has had excellent Spring integration for years. Spring 3.1 includes an Ehcache implementation. See the Spring 3.1 JavaDoc.

Spring 3.1

Spring Framework 3.1 has a generic cache abstraction for transparently applying caching to Spring applications. It has caching support for classes and methods using two annotations:

@Cacheable

Cache a method call. In the following example, the value is the return type, a Manual. The key is extracted from the ISBN argument using the id.

```
@Cacheable(value="manual", key="#isbn.id")
public Manual findManual(ISBN isbn, boolean checkWarehouse)
```

@CacheEvict

Clears the cache when called.

```
@CacheEvict(value = "manuals", allEntries=true)
public void loadManuals(InputStream batch)
```

For an excellent blog post covering SpEL expressions, see http://blog.springsource.com/2011/02/23/spring-3-1-m1-caching/.

Spring 2.5 - 3.1: Annotations For Spring

This open source, led by Eric Dalquist, predates the Spring 3.1 project. You can use it with earlier versions of Spring, or you can use it with 3.1.

@Cacheable

As with Spring 3.1 it uses an @Cacheable annotation to cache a method. In this example calls to findMessage are stored in a cache named "messageCache". The values are of type `Message`. The id for each entry is the `id` argument given.

```
@Cacheable(cacheName = "messageCache")
public Message findMessage(long id)
```

@TriggersRemove

And for cache invalidation, there is the @TriggersRemove annotation. In this example, `cache.removeAll()` is called after the method is invoked.

```
@TriggersRemove(cacheName = "messagesCache",
when = When.AFTER_METHOD_INVOCATION, removeAll = true)
public void addMessage(Message message)
```

See http://blog.goyello.com/2010/07/29/quick-start-with-ehcache-annotations-for-spring/ for a blog post explaining its use and providing further links.

The Annotations for Spring Project

To dynamically configure caching of method return values, use the Ehcache Annotations for Spring project at code.google.com. This project will allow you to configure caching of method calls dynamically using just configuration. The way it works is that the parameter values of the method will be used as a composite key into the cache, caching the return value of the method.

For example, suppose you have a method: `Dog getDog(String name)`.

Once caching is added to this method, all calls to the method will be cached using the "name" parameter as a key.

So, assume at time t0 the application calls this method with the name equal to "fido". Since "fido" doesn't exist, the method is allowed to run, generating the "fido" Dog object, and returning it. This object is then put into the cache using the key "fido".

Then assume at time t1 the application calls this method with the name equal to "spot". The same process is repeated, and the cache is now populated with the Dog object named "spot".

Finally, at time t2 the application again calls the method with the name "fido". Since "fido" exists in the cache, the "fido" Dog object is returned from the cache instead of calling the method.

To implement this in your application, follow these steps:

Step 1:

Add the jars to your application as listed on the Ehcache Annotations for Spring project site.

Step 2:

Add the Annotation to methods you would like to cache. Lets assume you are using the Dog getDog(String name) method from above:

```
@Cacheable(name="getDog")
Dog getDog(String name)
{
    ....
}
```

Step 3:

Configure Spring. You must add the following to your Spring configuration file in the beans declaration section:

```
<ehcache:annotation-driven cache-manager="ehCacheManager" />
```

The Annotations for Spring Project

More details can be found at:

- Ehcache Annotations for Spring project
- the project getting started page
- this blog

JSR107 Support

Information on BigMemory Go's Ehcache support of JSR107 is available on github.

BigMemory Go FAQ

The BigMemory Go Technical FAQ answers frequently asked questions on how to use BigMemory Go, integration with other products, and solving issues. Other resources for resolving issues include:

- Release Notes – Lists features and issues for specific versions of BigMemory Go and other Terracotta products.
- Compatibility Information – Includes tables on compatible versions of BigMemory Go and other Terracotta products, JVMs, and application servers.
- Terracotta Forums – If your question doesn't appear below, consider posting it on the Ehcache Forum.

Getting Started

Is BigMemory Go really free for 32GB?

Yes.

What if I want to use BigMemory Go for more than 32GB?

It's easy. To expand your license with a software subscription for more capacity, contact Terracotta.

What if I need enterprise support for BigMemory Go?

Terracotta provides enterprise support for BigMemory Go as part of software subscription. To get enterprise support, contact Terracotta.

Can I use BigMemory Go in production with my free 32GB license?

Yes.

Is there any limit on the number of servers or applications I can use with my free 32GB license for BigMemory Go?

No. Deploy BigMemory Go with as many applications and on as many servers as you like.

What's the difference between BigMemory Go and BigMemory Max?

BigMemory Go is for in-memory data management on a single JVM (in-process) and comes with 32GB free. BigMemory Max is for distributed in-memory management across an array of servers. For more on Go vs. Max, see BigMemory Overview.

Configuration

Where is the source code?

BigMemory Go is free to use, but it is not an open-source product. See the Ehcache website for an open-source caching project.

Where is the source code?

Can you use more than one instance of BigMemory Go in a single JVM?

Yes. Create a CacheManager using `new CacheManager(...)` and keep hold of the reference. The singleton approach, accessible with the `getInstance(...)` method, is still available too. However, hundreds of caches can be supported with one CacheManager, so use separate CacheManagers where different configurations are needed. The Hibernate Provider has also been updated to support this behavior.

What elements are mandatory in `ehcache.xml`?

See the file `ehcache.xsd` in the BigMemory Go kit for the latest information on required configuration elements.

How is auto-versioning of elements handled?

Automatic element versioning works only with memory-store caches only. BigMemory Go does not use auto-versioning.

To enable auto-versioning, set the system property `net.sf.ehcache.element.version.auto` to true (it is false by default). Manual (user provided) versioning of cache elements is ignored when auto-versioning is in effect. Note that if this property is turned on for one of the ineligible caches, auto-versioning will silently fail.

How do I get a memory-only store to persist to disk between JVM restarts?

BigMemory Go offers fast, robust disk persistence set through configuration.

What is the recommended way to write to a database?

There are two patterns available: write-through and write-behind caching. In write-through caching, 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.

Can I use BigMemory Go as a memory store only?

Yes. Just set the persistence strategy (in the <cache> configuration element) to "none":

```
<cache>
  ...
  <persistence strategy="none"/>
  ...
</cache>
```

Can I use BigMemory Go as a disk store only?

No. However, you can minimize the usage of memory using sizing configuration.

Where is the source code?

Is it thread-safe to modify element values after retrieval from a store?

Remember that a value in an element is globally accessible from multiple threads. It is inherently not thread-safe to modify the value. It is safer to retrieve a value, delete the element and then reinsert the value.

The UpdatingCacheEntryFactory does work by modifying the contents of values in place in the cache. This is outside of the core of BigMemory Go and is targeted at high performance CacheEntryFactories for SelfPopulatingCaches.

Can non-serializable objects be stored?

Non-serializable object can be stored only in the BigMemory Go memory store (heap). If an attempt is made to overflow a non-serializable element to the BigMemory Go off-heap or disk stores, the element is removed and a warning is logged.

What is the difference between TTL, TTI, and eternal?

These three configuration attributes can be used to design effective data lifetimes. Their assigned values should be tested and tuned to help optimize performance. `timeToIdleSeconds` (TTI) is the maximum number of seconds that an element can exist in the store without being accessed, while `timeToLiveSeconds` (TTL) is the maximum number of seconds that an element can exist in the store whether or not is has been accessed. If the `eternal` flag is set, elements are allowed to exist in the store eternally and none are evicted. The eternal setting overrides any TTI or TTL settings.

These attributes are set in the configuration file per cache. To set them per element, you must do so programmatically.

If null values are stored in the cache, how can my code tell the difference between "intentional" nulls and non-existent entries?

Your application is querying the database excessively only to find that there is no result. Since there is no result, there is nothing to cache. To prevent the query from being executed unnecessarily, cache a null value, signalling that a particular key doesn't exist.

In code, checking for intentional nulls versus non-existent cache entries may look like:

```
// cache an explicit null value:

cache.put(new Element("key", null));

Element element = cache.get("key");

if (element == null) {

  // nothing in the cache for "key" (or expired) ...

} else {

  // there is a valid element in the cache, however getObjectValue() may be null:

  Object value = element.getObjectValue();

  if (value == null) {
```

Where is the source code?

```
    // a null value is in the cache ...

  } else {

    // a non-null value is in the cache ...

  }

}
```

The cache configuration in `ehcache.xml` may look similar to the following:

```
<cache
  name="some.cache.name"
  maxEntriesLocalHeap="10000"
  eternal="false"
  timeToIdleSeconds="300"
  timeToLiveSeconds="600"
/>
```

Use a finite timeToLiveSeconds setting to force an occasional update.

How many threads does BigMemory Go use, and how much memory does that consume?

The amount of memory consumed per thread is determined by the Stack Size. This is set using -Xss.

What happens when maxEntriesLocalHeap is reached? Are the oldest items expired when new ones come in?

When the maximum number of elements in memory is reached, the Least Recently Used (LRU) element is removed. "Used" in this case means inserted with a `put` or accessed with a `get`. The LRU element is flushed asynchronously to the off-heap store.

Why is there an expiry thread for the disk store but not for the other stores?

Because the in-memory data is allowed a fixed maximum number of elements or bytes, it will have a maximum memory use equal to the number of elements multiplied by the average size. When an element is added beyond the maximum size, the LRU element gets flushed to the disk store. Running an expiry thread in memory turns out to be a very expensive operation and potentially contentious. It is far more efficient to only check expiry when need rather than explicitly search for it. The tradeoff is higher average memory use.

The disk-store expiry thread keeps the disk clean. There is hopefully less contention for the disk store's locks because commonly used values are in memory. If you are concerned about CPU utilization and locking in the disk store, you can set the diskExpiryThreadIntervalSeconds to a high number, such as 1 day. Or, you can effectively turn it off by setting the diskExpiryThreadIntervalSeconds to a very large value.

What eviction strategies are supported?

LRU, LFU and FIFO eviction strategies are supported.

How does element equality work in serialization mode?

An element (key and value) in BigMemory is guaranteed to `.equals()` another as it moves between stores.

Where is the source code?

Can you use BigMemory Go as a second-level cache in Hibernate and BigMemory Go outside of Hibernate at the same time?

Yes. You use one instance of BigMemory Go with one ehcache.xml. You configure your caches with Hibernate names for use by Hibernate. You can have other caches which you interact with directly, outside of Hibernate.

Operations

How do you get an element without affecting statistics?

Use the Cache.getQuiet() method. It returns an element without updating statistics.

Is there a simple way to disable BigMemory Go when testing?

Set the system property `net.sf.ehcache.disabled=true` to disable BigMemory Go. This can easily be done using `-Dnet.sf.ehcache.disabled=true` on the command line. If BigMemory Go is disabled, no elements will be added to the stores.

How do I dynamically change cache attributes at runtime?

This is not possible. However, you can achieve the same result as follows:

1. Create a new cache:

   ```
   Cache cache = new Cache("test2", 1, true, true, 0, 0, true, 120, ...);
   cacheManager.addCache(cache);
   ```

   See the BigMemory API documentation for the full parameters.
2. Get a list of keys using `cache.getKeys`, then get each element and put it in the new cache.

   None of this will use much memory because the new cache elements have values that reference the same data as the original cache.
3. Use `cacheManager.removeCache("oldcachename")` to remove the original cache.

Do you need to explicitly shut down the CacheManager when you finish with BigMemory Go?

There is a shutdown hook which calls the shutdown on JVM exit. If the JVM keeps running after you stop using BigMemory Go, you should call `CacheManager.getInstance().shutdown()` so that the threads are stopped and cache memory is released back to the JVM.

Can you use BigMemory Go after a CacheManager.shutdown()?

When you call `CacheManager.shutdown()` is sets the singleton in CacheManager to null. Using a cache after this generates a CacheException.

However, if you call `CacheManager.create()` to instantiate a new CacheManager, then you can still use BigMemory Go. Internally the CacheManager singleton gets set to the new one, allowing you to create and shut down any number of times.

How do you get an element without affecting statistics?

Why are statistics counters showing 0 for active caches?

Statistics gathering is disabled by default in order to optimize performance. You can enable statistics gathering in caches in one of the following ways:

- In cache configuration by adding `statistics="true"` to the `<cache>` element.
- Programmatically when setting a cache's configuration.
- In the
- In the Terracotta Management Console.

To function, certain features in the Terracotta Management Console require statistics to be enabled.

How do I detect deadlocks in BigMemory Go?

BigMemory Go does not experience deadlocks. However, deadlocks in your application code can be detected with certain tools, such as the JDK tool JConsole.

Troubleshooting

I have created a new cache and its status is STATUS_UNINITIALISED. How do I initialise it?

You need to add a newly created cache to a CacheManager before it gets initialised. Use code like the following:

```
CacheManager manager = CacheManager.create();
Cache myCache = new Cache("testDiskOnly", 0, true, false, 5, 2);
manager.addCache(myCache);
```

Why did a crashed standalone BigMemory node not come up with all data intact?

Persistence was not configured or not configured correctly on the node.

I added data Client 1, but I can't see it on Client 2. Why not?

BigMemory Go does not distribute data. See BigMemory Max.

I have a small data set, and yet latency seems to be high.

There are a few ways to try to solve this, in order of preference:

1. Try pinning the cache. If the data set fits comfortably in heap and is not expected to grow, this will speed up gets by a noticeable factor. Pinning certain elements and/or tuning ARC settings might also be effective for certain use cases.
2. Increase the size of the off-heap store to allow data sets that cannot fit in heap—but can fit in memory—to remain very close to your application.

I am using Java 6 and getting a java.lang.VerifyError on the Backport Concurrent classes. Why?

The backport-concurrent library is used in BigMemory Go to provide java.util.concurrency facilities for Java 4 - Java 6. Use either the Java 4 version which is compatible with Java 4-6, or use the version for your JDK.

I have created a new cache and its status is STATUS_UNINITIALISED. How do I initialise it?

I get a javax.servlet.ServletException: Could not initialise servlet filter when using SimplePageCachingFilter. Why?

If you use this default implementation, the cache name is called "SimplePageCachingFilter". You need to define a cache with that name in ehcache.xml. If you override CachingFilter, you are required to set your own cache name.

Why is there a warning in my application's log that a new CacheManager is using a resource already in use by another CacheManager?

```
WARN  CacheManager ...  Creating a new instance of CacheManager using the diskStorePath
"C:\temp\tempcache" which is already used by an existing CacheManager.
```

This means that, for some reason, your application is trying to create one or more additional instances of CacheManager with the same configuration. Depending upon your persistence strategy, BigMemory Go will automatically resolve the disk-path conflict, or it will let you know that you must explicitly configure the diskStorePath.

To eliminate the warning:

- Use a separate configuration per instance.
- If you only want one instance, use the singleton creation methods, i.e., `CacheManager.getInstance()`. In Hibernate, there is a special provider for this called `net.sf.ehcache.hibernate.SingletonEhCacheProvider`. See Hibernate.

What does the following error mean? "Caches cannot be added by name when default cache config is not specified in the config. Please add a default cache config in the configuration."

The `defaultCache` is optional. When you try to programmatically add a cache by name, `CacheManager.add(String name)`, a default cache is expected to exist in the CacheManager configuration. To fix this error, add a defaultCache to the CacheManager's configuration.

Do I have to restart BigMemory Go after redeploying in a container?

Errors could occur if BigMemory Go runs with a web application that has been redeployed, causing BigMemory Go to not start properly or at all. If the web application is redeployed, be sure to restart BigMemory Go.

The Terracotta Management Console

Introduction

The Terracotta Management Console (TMC) is a web-based administration and monitoring application providing a wealth of advantages, including:

- Multi-level security architecture, with end-to-end SSL secure connections available
- Feature-rich and easy-to-use in-browser interface
- Remote management capabilities requiring only a web browser and network connection
- Cross-platform deployment
- Role-based authentication and authorization
- Support for LDAP directories and Microsoft Active Directory
- Aggregate statistics from multiple nodes
- Flexible deployment model plugs into both development environments and secure production architectures

The TMC can monitor BigMemory nodes and clusters through the Terracotta Management Server (TMS). The TMS acts as an aggregator and also provides a connection and security context for the TMC. The TMS must be available and accessible for the TMC to provide management services.

The TMS is included with your BigMemory kit under the `tools/management-console` directory.

For more information on using the TMC, see the following:

- Enabling Management in BigMemory Nodes
- Using the TMC UI
- Setting Up Security
- Directly Accessing the REST API
- Integrating With Nagios

Installing and Configuring the TMS

The TMS files are stored in the BigMemory kit's `management-console` directory. You can copy this directory with all of its contents to the location where the TMS will run. Be sure to place a copy of the license file inside the `management-console` directory.

Running With a Different Container

The TMS can be run directly with the provided container. To run it with an application server of your choice, use the file `management-console/webapps/tmc.war`. Follow the specifications and requirements of your chosen application server for deploying a WAR-based application.

Configuration

A Terracotta BigMemory Max cluster can be managed directly by connecting the TMC to any one of the servers in the cluster. All other servers and clients become visible to the TMC through that initial connection. No special setup or configuration is required. Simply create a new connection and enter the URI to a server in

the form

```
<scheme>://<host-address>:<tsa-port>.
```

To manage a client or standalone node (Terracotta Ehcache client or BigMemory Go) directly using the TMC, you must enable the REST management service on that node. To enable the REST management service on a BigMemory Go or Ehcache node, set the following element in the ehcache.xml configuration:

```
<ehcache ... >
...
  <managementRESTService enabled="true" bind="<ip_address>:<port>"/>
...
</ehcache>
```

where `<ip_address>` is the local network interface's IP address and `<port>` is the port number used to manage the node. The following defaults are in effect for `<managementRESTService>`:

- enabled="false" (This must be set to "true" for standalone caches)
- bind="0.0.0.0:9888" (This IP address binds the specified port all network interfaces on the local node)

The REST management service can also be enabled programmatically:

```
ManagementRESTServiceConfiguration rest = new ManagementRESTServiceConfiguration();
rest.setBind("0.0.0.0:9888");
rest.setEnabled(true);
config.addManagementRESTService(rest);
```

Displaying Update Statistics

By default, distributed caches (caches distribute in BigMemory Max) generate put events whenever elements are put or updated. To have the TMC track and display updates separately from puts, set the Terracotta property `ehcache.clusteredStore.checkContainsKeyOnPut` at the top of the Terracotta configuration file (`tc-config.xml` by default) before starting the Terracotta Server Array:

```
<tc-properties>
  <property name="ehcache.clusteredStore.checkContainsKeyOnPut" value="true" />
</tc-properties>
```

Enabling this property can substantially degrade performance. Before using in production, the effect of enabling this property should be thoroughly tested.

Using Multiple Instances of BigMemory Go CacheManagers With the TMC

When loading multiple instances of BigMemory Go CacheManagers with the TMC rest agent enabled in the same JVM, CacheManagers must be loaded by distinct classloaders. Two different web applications (two different WARs), for example, are loaded by different classloaders.

The TMC Update Checker

The Update Checker automatically checks to see if you have the latest updates, and collects diagnostic information on TMC usage. The Update Checker is on by default. To disable the update checker, use the

following system property:

```
-Dcom.terracotta.management.skipUpdateCheck=true
```

Starting and Connecting to the TMC

Start the TMC by running the following script:

```
management-console/bin/start-tmc.sh
```

To stop the TMC, use the following script:

```
management-console/bin/stop-tmc.sh
```

For Microsoft Windows, use start.bat and stop.bat, available in the same directory.

Connect to the TMC using the following URI with a standard web browser:

```
http://localhost:9889/tmc
```

If you are connecting remotely, substitute the appropriate hostname. If you have set up secure browser connections, use "https:" instead of "http:".

When you first connect to the TMC, the security setup page appears, where you can choose to run the TMC with authentication or without. Authentication can also be enabled/disabled in the TMC Settings panel once the TMC is running. For more information, refer to the Security Setup page.

For more information on using the TMC, click the **Help** links available on certain pages within the UI, or choose **Help** from the tool bar to access the TMC online help. You can also view information about using the TMC on this page.

Updating the TMS

Installing a new version of a Terracotta kit also installs an updated version of the TMS. When this new version is started, it will check for existing configuration files under `<user.home>/.tc/mgmt`, backing up any incompatible files (extension .bak). In this case, previously configured connections will not appear in the TMC and must be re-added.

Uninstalling the TMC

If you want to remove the Terracotta Managment Console, delete the `~/.tc/mgmt/` directory.

Terracotta Management Console Security Setup

Introduction

The Terracotta Management Server (TMS) includes a flexible, multi-level security architecture to easily integrate into a variety of environments.

The following levels of security are available:

- No authentication, and no or limited secured connections. Note that if the TMS cannot locate a Terracotta license at startup, or locates a Terracotta license that does not specifically enable it, no security setup page appears and the TMS runs without security of any type.
- Default role-based user authentication only. This is built in and set up when you first connect to TMS, and is intended to control access to the TMS. Standard LDAP and Microsoft Active Directory integration is also available.
- Basic security offering authentication and authorization of BigMemroy Go and BigMemory Max nodes (referred to as *agents* or *managed agents* in this context), as well as message hashing and other protective measures.
- Secured connections based on Secure Sockets Layer (SSL) technology can be used in conjunction with basic security.
- Certificate-based client authentication to enhance SSL-based security. In this case, basic security is disabled.

With the noted exceptions, these security layers can be used together to provide the overall level of security required by your environment. Except as noted below, security features are available only with a properly licensed version of the TMS.

This document discusses security from the perspective of the TMS. However, the TMS and the Terracotta Management Console (TMC) function in the same security context.

No Security

Upon initial connection to a properly licensed TMC, the authentication setup page appears, where you can choose to run the TMC with authentication or without.

Authentication can also be enabled/disabled in the TMC Settings panel once the TMC is running. If you enable authentication, all of the security features described in this document are available.

If you do not enable authentication, you will be directly connected to the TMC without being prompted for a login/password.

Even with no security enabled, however, you can still force SSL connections between browsers and the TMC.

Default Security

Default security consists of the built-in role-based accounts that are used to log into the TMC. This level of security controls access to the TMC only, and is appropriate for environments where all components, including the TMC, managed agents, and any custom RIAs, are on protected networks. An internal network

behind a firewall, where all access is trusted, is one example of such an environment. Note that connections between the TMC and managed agents remain unsecured.

Optionally, integration with an LDAP or Microsft Active Directory is also available. For more information, see the TMC help.

When TMS/TMC authentication is configured (whether with the .ini file, or LDAP or Active Directory), if a non-Administrator user logs into the TMS/TMC, he will not be able to see the Administration panel in the TMC, nor will he be able to perform administrative tasks such as shutting down a server. However, if a cluster is not secured, a non-Administrator user will be able to use the TMS Rest API to perform administrative tasks on the cluster.

In other words, if you secure the TMS/TMC but do not secure your TSA cluster, any user can perform administrative tasks on the cluster through the Rest API. To prevent this, you must secure both the TMS/TMC and your cluster.

If you are unsure whether your cluster is secured, go to the Connections tab in the Settings window, and look for the locked padlock icon next to your connection.

For more information about TSA security, refer to Securing Terracotta Clusters.

Basic Connection Security

You can secure the connections between the TMS and managed agents using a built-in hash-based message authentication scheme and digital certificates, also known as "identity assertion" (IA). Use this level of security in environments where the TMS may be exposed to unwanted connection attempts from rogue agents, or managed agents may come under attack from a rogue TMS.

NOTE: To fully secure connections between the TMS and managed agents, it is recommended that SSL be used for encryption.

To set up IA, complete the following steps:

- Set up a truststore for the TMS
- Configure IA on managed agents
- Create the shared secret for the TMS and managed agents

Setting Up a Truststore

The TMS must have a truststore containing the public-key certificate of every agent that connects to it. If you are not using a Certificate Authority (which provides the public keys), you must export public keys from the self-signed certificates in the keystore of each agent using a command similar to the following:

```
keytool -export -alias myAgent -keystore keystore-file.jks \
    -file myAgentCert.cert
```

Then import the keys into the TMS truststore, creating it as shown if it does not exist:

```
keytool -import -alias myAgent -file myAgentCert.cert \
    -keystore truststore.jks
```

Note that if a managed agent does not have a keystore, you must set one up. See the cluster security documentation for examples.

Once you create a truststore for storing these public keys, it must be available to the TMS in one of the follwoing ways:

- `${user.home}/.tc/mgmt/tms-truststore`
- a location configured with the system property `javax.net.ssl.trustStore`

Alternatively, you can import these public keys into the default truststore for the JVM (typically the cacerts file).

NOTE: If a different default location for TMS-related files is required, set it using the system property `com.tc.management.config.directory`.

Configuring IA

To configure IA for the TSA, see the TSA security setup page.

To configure IA on a Terracotta client, enable security (authentication via IA) on the REST service by adding the "securityServiceLocation" attribute to the managementRESTService element in the managed agent's configuration. The following example is for Ehcache:

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

If securityServiceLocation is not set, the authentication feature is disabled. To enable, set its value to the URI used to connect to the TMC, with `/tmc/api/assertIdentity` appended. In the example example above, "http://localhost:9889" is the TMC URI.

For BigMemory Go, use the same procedure as for a Terracotta client.

Creating a Shared Secret

You must create a password (or secret) that is shared between the TMS and managed agents, storing it in a Terracotta keychain file.

The scripts required in the following procedures are found in `${BIGMEMORY_GO_HOME}/management-console/bin` or `${BIGMEMORY_MAX_HOME}/tools/security/bin`. Use the equivalent .bat scripts for Microsoft Windows.

Shared Secret on the TMS

1. Create a shared secret for the assertion of trust between the TMS and managed agents by running the following script:

   ```
   ./add-tc-agent.sh <agent-url>
   ```

where `<agent-url>` is the URI of the agent. This value should correspond exactly to the URI you use in the TMC to connect to the given agent. For example:

```
./add-tc-agent.sh http://localhost:9888
```

Use `add-tc-agent.bat` with Microsoft Windows.

The script will automatically create the Terracotta keychain file `<user_home>/.tc/mgmt/keychain` if it does not already exist. Do not move or delete this keychain file—it must remain accessible to the TMS at that location.

2. When prompted, enter a shared secret of your choice.
   Be sure to note the secret that you enter, as you may need to enter it again in a later step.
3. Run the add-tc-agent script once for each agent, using that agent's URI.
   The script saves these entries to the same keychain file.

Shared Secret on Managed Agents

1. Each agent with a keychain entry must also have access to the same shared secret via a Terracotta keychain file:

   ```
   ./keychain.sh -c <user_home>/.tc/mgmt/agentKeychainFile \
       http://myHost:9889/tc-management-api
   ```

   where `<tmc-url>` is the URI used to connect to the TMC, with with `/tc-management-api` appended. If the named keychain file already exists on the node, omit the `-c` flag. Agents running on the same node can share a keychain file.
2. Enter the master key for the keychain file:

   ```
   Terracotta Management Console - Keychain Client
   KeyChain file successfully created in /path/to/agentKeychainFile
   Open the keychain by entering its master key:
   ```
3. Enter the shared secret associated with the TMS:

   ```
   Enter the password you wish to associate with this URL:
   Password for http://myHost:9889/ successfully stored
   ```

   The secret you enter must match the one entered for the TMS. Note that the script's success acknowledgment does *not* confirm that the secret matches the one stored on the TMS.

Adding SSL

In an environment where connections may be intercepted, or a higher level of authentication is required, adding SSL provides encryption. SSL should be used to enhance basic security.

To add SSL to BigMemory Max, see the TSA security setup page.

To add SSL to BigMemory Go, follow these steps for each node:

1. Enable SSL on the REST service by setting the managementRESTService element's "sslEnabled" attribute to "true" in the managed agent's configuration:

   ```
   <ehcache ...>
   ```

```
...
  <managementRESTService enabled="true"
  securityServiceLocation="https://localhost:9889/tmc/api/assertIdentity"
  sslEnabled="true" />
...
</ehcache>
```

2. Provide an identity store for the managed agent either at the default location, `${user.home}/.tc/mgmt/keystore`, or by setting the store's location with the system property `javax.net.ssl.keyStore`.

   The identity store is where the server-authentication certificate is stored. If the identity store cannot be found, the managed agent fails at startup.
3. Add a password for the managed agent's identity store to its keychain.

   The password must be keyed with the identity-store file's URI. Or set the password with the system property `javax.net.ssl.keyStorePassword`. If no password is found, the managed agent fails at startup.
4. The JVM running the TMS must have the same server-authentication certificate in one of the following locations:

   - the default truststore for the JVM (typically the cacerts file)
   - `${user.home}/.tc/mgmt/tms-truststore`
   - a location configured with the system property `javax.net.ssl.trustStore`

   If a truststore was already set up for the TMS and it contains the required public key, then skip this step.
5. If a custom truststore (not cacerts) is designated for the TMS, the truststore password must be included in the TMS keychain.

   The password must be keyed with the truststore file's URI. Or set the password with the system property `javax.net.ssl.trustStorePassword`.

Certificate-Based Client Authentication

As an alternative to the hash-based message authentication scheme of basic security, you can use certificate-based client authentication with BigMemory Go nodes. This form of authentication is not available for use with the Terracotta Server Array.

Setting up client authentication automatically turns off hash-based authentication. Note that you must configure SSL to use this security option.

You must set up keystores for all managed agents and a truststore for the TMS as described in the basic security and SSL sections. In addition, you must also set up truststores for all managed agents and a keystore for the TMS, as described in the following procedure.

To enable certificate-based client authentication, follow these steps:

1. Enable client authentication on the REST service by setting the managementRESTService element's "needClientAuth" attribute to "true" in the managed agent's configuration:

   ```
   <ehcache ...>
   ...
     <managementRESTService enabled="true"
   ```

```
    securityServiceLocation="http://localhost:9889/tmc/api/assertIdentity"
    sslEnabled="true" needClientAuth="true" />
...
</ehcache>
```

2. Provide a truststore for the managed agent at the default location, `${user.home}/.tc/mgmt/truststore`, or by setting the truststore location with the system property `javax.net.ssl.trustStore`.
3. The password for the truststore must be included in the managed agent's keychain.

   The password must be keyed with the truststore file's URI. Or set the password with the system property `javax.net.ssl.trustStorePassword`.
4. Provide an identity store for the TMS at the default location, `${user.home}/.tc/mgmt/tms-keystore`, or by setting the identity-store location with the system property `javax.net.ssl.keyStore`.

   The managed agent will be rejected by the TMS unless a valid certificate is found.
5. The password for the TMS identity store must be included in the TMS keychain.

   The password must be keyed with the identity-store file's URI. Or set the password with the system property `javax.net.ssl.keyStorePassword`.
6. To allow an SSL connection from the managed agent, an SSL connector must be configured. If the TMS is deployed with the provided Jetty web server, add the following to `/management-console/etc/jetty.xml` (in the BigMemory kit) as shown:

```
<Call name="addConnector">
 <Arg>
   <New class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
     <Arg>
       <New class="org.eclipse.jetty.http.ssl.SslContextFactory">
         <Set name="keyStore">/home/.tc/mgmt/tms-keystore</Set>
         <Set name="keyStorePassword">
     OBF:1v9u1w1c1ym51xmq1rwd1rwh1xmk1ym91w261v8s</Set>
         <Set name="keyManagerPassword">
     OBF:1v9u1w1c1ym51xmq1rwd1rwh1xmk1ym91w261v8s</Set>
         <Set name="TrustStore">/home/.tc/mgmt/tms-truststore</Set>
         <Set name="keyStorePassword">
     OBF:1v9u1w1c1ym51xmq1rwd1rwh1xmk1ym91w261v8s</Set>
         <Set name="needClientAuth">true</Set>
       </New>
     </Arg>
     <Set name="port">9999</Set>
     <Set name="maxIdleTime">30000</Set>
   </New>
 </Arg>
</Call>
```

Note the following about the configuration shown:

- If the TMS WAR is deployed with a different container, make the equivalent changes appropriate to that container.
- The SSL port must be free (unused by any another process) to avoid collisions.
- maxIdletime can be set to a value that suits your environment.
- If the default keystore or truststore are not being used, enter the correct paths to the keystore and truststore being used.
- Passwords have been obfuscated using a built-in Jetty tool:

```
java -cp lib/jetty-runner.jar
org.eclipse.jetty.util.security.Password myPassword
```

This command, which must be run from the TMC root directory, returns an obfuscated version of `myPassword`.

Forcing SSL connections For TMC Clients

If the TMC is deployed with the provided Jetty web server, web browsers connecting to the TMC can use an unsecured connection (via HTTP port 9889). A secure SSL-based connection is also available using HTTPS port 9443.

To force all web browsers to connect using SSL, disable the non-secure connector by commenting it out in `/management-console/etc/jetty.xml` (located in the BigMemory kit):

```
<!-- DISABLED non-secure connector
<Call name="addConnector">
  <Arg>
    <New class="org.eclipse.jetty.server.nio.SelectChannelConnector">
      <Set name="host"><Property name="jetty.host" /></Set>
      <Set name="port"><Property name="jetty.port" default="9889"/></Set>
      <Set name="forwarded">true</Set>
      <Set name="maxIdleTime">300000</Set>
      <Set name="Acceptors">2</Set>
      <Set name="statsOn">false</Set>
      <Set name="confidentialPort">8443</Set>
      <Set name="lowResourcesConnections">20000</Set>
      <Set name="lowResourcesMaxIdleTime">5000</Set>
    </New>
  </Arg>
</Call>
-->
```

If the TMC WAR is deployed with a different container, make the equivalent changes appropriate to that container.

About the Default Keystore

By default, the built-in Jetty container's configuration file (`management-console/etc/jetty.xml`) uses a JKS identity store, located in the same directory. This keystore contains a self-signed certificate (not signed by a certificate authority). If you intend to use this "untrusted" certificate, all SSL browser connections must recognize this certificate and register it as an exception for future connections. This is usually done at the time the browser first connects to the TMS.

Terracotta REST API

Introduction

The Terracotta Management Server (TMS) includes an embedded management REST interface. You can write custom scripts against this interface, or create a custom Rich Internet Application in place of the Terracotta Management Console (TMC).

This document defines and describes the management REST API used to interact with the REST interface agent.

NOTE: For simplicity, many of the examples given in this document assume a TMS that is running locally, and therefore "localhost" is used for the host address.

Connecting to the Management Service REST API

The REST API is available by connecting to the REST management service running on the TMS or a node running a REST agent. Use the URLs shown below.

Connecting to a TMS

```
http://<host>:<port>/tmc/api
```

where `<port>` is 9889 if running the TMS with the default container. If using your own container, substitute the port configured for that container.

Connecting Directly to a Standalone Node

```
http://<host>:<port>/tc-management-api
```

where `<port>` is configured in the <managementRESTService> element's bind attribute, in the Ehcache configuration file (`ehcache.xml` by default).

Connecting to a TSA

```
http://<host>:<port>/tc-management-api
```

where `<port>` is the TSA group port. This value is configured in a server's <tsa-group-port> element in the Terracotta configuration file (`tc-config.xml` by default). The default value for the group port is 9530.

Constructing URIs for HTTP Operations

The typical URI used to connect to the Terracotta management service has the following format:

```
<scheme>://<host>[:<port>]/<path>?<query>
```

These URIs use the standard scheme and domain, with "http" assumed as the scheme. HTTP operations access the REST API through URIs. The URI allows query strings under certain circumstances.

The URI Path

The `<path>` portion of the URI specifies resource locations using the following hierarchy:

1. Agent IDs – List of the desired clients using unique identifiers. If the connection is to a TMS and no IDs are given, all known clients are accessed. If the connection is made directly to a Terracotta client or the TSA (to a standalone REST service), then no ID is used since these are identified by a host:port address. All standalone (including TSA) REST interfaces will return the agent ID "embedded".
2. CacheManager names – List of the CacheManagers using their configured names. If "cacheManagers" is specified in the URI but no names are given, all CacheManagers for the specified clients are accessed.
3. Cache names – List of the caches using their configured names. If "caches" is specified in the URI but no names are given, all caches belonging to the accessed CacheManagers are accessed. In the case where access is broad, a substantial amount of data may be returned by a GET operation.

The structure of the path takes the following form:

```
/agents[;ids={comma_sep_agent_ids}]/cacheManagers[;names={comma_sep_cache_manager_names}]
    /caches[;names={comma_sep_cache_names}
```

Agent IDs are not required.

Special Resource Locations

Certain resource locations provide specific monitoring and administration services.

Discovery

A "discovery" URI format uses the path `/agents/info`. Used with a TMS, this URI returns metadata on all agents known (through configuration) to that TMS. Used with an embedded web service, metadata on that agent is returned (or a 404 if that agent is not reachable). Examples of other specific uses of discovery URIs are given elsewhere in this document.

Viewing Configuration

A URI format for viewing the configuration of CacheManagers and caches uses the path `agents/cacheManagers` or `agents/cacheManagers/caches`. Agents, cacheManagers, and caches can be specified using IDs and names. The data is returned in its native XML format.

To get the configuration of one or more CacheManagers, use the following format:

```
/agents[;ids={comma_sep_agent_ids}]/cacheManagers[;names={comma_sep_cache_manager_names}]
    /configs
```

Note: If no client IDs are specified in the request, then all of the clients' cacheManagers are returned. However, if no client IDs are specified in the request and the number of clients is more than the default maximum of 64, then an error is returned in the JSON response. The VM argument `com.terracotta.agent.defaultMaxClientsToDisplay` can be used to change the maximum number of clients to display.

To get the configuration of one or more caches, use the following format:

```
/agents[;ids={comma_sep_agent_ids}]/cacheManagers[;names={comma_sep_cache_manager_names}]
     /caches[;names={comma_sep_cache_names}/configs
```

Setting Configuration

Cache resource locations can also be specified for setting specific cache-configuration attributes using resource representations. The following is a comprehensive list of the attributes that can be set:

- enabled – A boolean for enabling (true, DEFAULT) or disabling (false) cache operations. For example, to disable a cache's operation: PUT {"enabled":true}.
- statsEnabled – A boolean for enabling (true) or disabling (false, DEFAULT) the gathering of cache statistics.
- sampledStatsEnabled – A boolean for enabling (true) or disabling (false, DEFAULT) the sampling of cache statistics.

Probing a New Connection URI

To probe the existence of an agent at a given location, use an URL with the following format:

```
   http://127.0.0.1:9889/tmc/api/agents/probeUrl/$urlToProbe
```

For example, the following should return information about the REST agent running at the given address (localhost:4343):

```
http://127.0.0.1:9889/tmc/api/agents/probeUrl/http%253A%252F%252Flocalhost%253A4343
```

If the agent is available, a (status code 200 AgentMetadataEntity) response similar to the following is returned:

```
   {"agentId":"embedded","agencyOf":"Ehcache","available":true,"secured":false,
"sslEnabled":false,"needClientAuth":false,"licensed":false,"sampleHistorySize":8640,
"sampleIntervalSeconds":10,"enabled":false,"restAPIVersion":"2.7.0"}
```

If not available, the status code should be "204 No Content".

Setting the authentication mode

The authentication status is saved in the properties file `${user.home}/.tc/mgmt/settings.ini`. There is a REST resource to interact with it (setting it on and off); setting it forces a manual restart of the TMS.

To GET the status, use the following URI:

```
http://localhost:9889/tmc/api/config/settings/authentication
```

Either "true" or "false" is returned.

To change (PUT) the status, call the same url with a request body containing "true" or "false". A "200 OK" status code is returned upon successful update.

See the examples of URIs, below, for how to use these resource locations.

Specifications for HTTP Operations

If no agent IDs are specified in a URI, all known agents are included.

Response Headers

For a typical HTTP request, the response header appears similar to the following:

```
  -- response --
200 OK
Content-Type:  application/vnd.sun.wadl+xml
Allow:  OPTIONS,GET,HEAD
Content-Length:  602
Server:  Jetty(7.5.4.v20111024)
```

Examples of URIs

The flexibility of the management-service REST API in turn makes available a flexible URI syntax. The following list of examples illustrate HTTP responses to specific URIs. These examples of the data returned by the listed HTTP operations are shown below without response headers.

{toc-zonel5:5}

DELETE

Removes a specified resource, such as a cache.

Examples

The following DELETE examples are organized by task and URI.

Remove a Cache

```
/agents;id=client01/cacheManagers;names=foo/caches;names=bar
```

Remove the cache "bar" from CacheManager "foo" on the Ehcache node "client01".

Clear Cache Statistics

```
/agents;id=client01/cacheManagers;names=foo/caches;names=bar/configs
```

Clears all cache statistics for cache "foo" and resets counters to zero.

Possible HTTP Status Codes for DELETE

400 – URI does not specify a single resource. 404 – Resource specified in the URI cannot be found.

GET and HEAD

Returns a JSON array representing the details of all specified resources, or an XML representation of data whose native format is XML.

NOTE: HEAD operations return the same metadata as GET operations, but no body.

Examples

The following GET examples are organized by task and URI.

Discover All Known Agents

```
/agents/info
```

Used with a TMS, this URI returns metadata on all agents known (through configuration) to that TMS. Used with an embedded web service, metadata on that agent is returned.

The following is a response from a TMS that has agents "foo" and "goo" configured, and both are responding:

```
[{"restAPIVersion":"1.0.0","available":true,"agentId":"foo","agencyOf":"Ehcache"},
{"restAPIVersion":"1.0.0","available":true,"agentId":"goo","agencyOf":"Ehcache"}]
```

The following is a response from a TMS that has agents "foo" and "goo" configured, but with only "foo" responding:

```
[{"restAPIVersion":"1.0.0","available":true,"agentId":"foo","agencyOf":"Ehcache"},
{"restAPIVersion":null,"available":false,"agentId":"goo","agencyOf":null}]
```

Note that the metadata returned includes the API version running on the agent, as well as the type of client ("agencyOf") the API is serving.

Get Details on Specific Agents

```
/agents;ids=client01,client02
```

JSON representing an array all available agent detail. If no agent IDs are included, all agents available are returned.

Get Details on Specific Caches

```
/agents;ids=client01/cacheManagers;names=foo/caches;names=bar
```

Get Configuration of Specific CacheManager

```
/agents;ids=client01/cacheManagers;names=foo/configs
```

Returns an XML representation of the CacheManager "foo". For example, the following is an XML representation returned from a standalone Ehcache node:

```
<configurations agentId="embedded" version="1.0.0-SNAPSHOT">
  <configuration cacheManagerName="foo">
    <ehcache maxBytesLocalDisk="300M" maxBytesLocalHeap="100M" maxBytesLocalOffHeap="200M"
    monitoring="on" name="CM1">
    <diskStore path="/var/folders/nn/lxsg77756534qfn7z14y5gtm0000gp/T/"/>
      <managementRESTService bind="0.0.0.0:9889" enabled="false"/>
      <cache name="Cache11">
    <persistence strategy="localTempSwap"/>
        <elementValueComparator class="net.sf.ehcache.store.DefaultElementValueComparator"/>
        <terracotta clustered="false">
```

```
          <nonstop/>
        </terracotta>
      </cache>
    </ehcache>
  </configuration>
</configurations>
```

Certain operations can only be executed against specific targets. Specifying multiple agents, CacheManagers, or caches will generate an error response (code 400).

Get Configuration of Specific Caches

```
/agents;ids=client01/cacheManagers;names=foo/caches;names=baz/configs
```

Get All CacheManager Details

```
/agents/cacheManagers
```

The following example shows a JSON object returned by this URI when the GET is executed against a standalone Ehcache node with two CacheManagers, each with one cache:


```
   [{"name":"CM2","attributes":{"ClusterUUID":"03e505092b6a4b1a9af5d1b035a7d5ed","
Enabled":true,"HasWriteBehindWriter":false,"MaxBytesLocalDiskAsString":"300M","
CacheAverageSearchTime":0,"CachePutRate":84,"CacheOnDiskHitRate":0,
"CacheMetrics":{"Cache12":[2,84,84]},
"CacheRemoveRate":0,"CacheOffHeapHitRate":0,"Searchable"
:false,"CacheOnDiskMissRate":84,"CacheNames":["Cache12"],"
TransactionRolledBackCount":0,"CacheInMemoryHitRate":2,"WriterQueueLength":0,"
CacheOffHeapMissRate":0,"Transactional":false,"CacheHitRate":2,"
TransactionCommitRate":0,"CacheExpirationRate":0,"CacheUpdateRate":0,"
MaxBytesLocalHeap":104857600,"CacheAverageGetTime":0.027891714,"
TransactionRollbackRate":0,"CacheEvictionRate":0,"CacheInMemoryMissRate":84,"
MaxBytesLocalDisk":314572800,"MaxBytesLocalOffHeapAsString":"200M","
CacheSearchRate":0,"TransactionCommittedCount":0,"TransactionTimedOutCount":0,"
Status":"STATUS_ALIVE","MaxBytesLocalOffHeap":209715200,"WriterMaxQueueSize":0,"
StatisticsEnabled":true,"MaxBytesLocalHeapAsString":"100M","CacheMissRate":84},"
agentId":"embedded","version":"1.0.0-SNAPSHOT"},{"name":"CM1","attributes":
{"ClusterUUID":"03e505092b6a4b1a9af5d1b035a7d5ed","Enabled":true,"
HasWriteBehindWriter":false,"MaxBytesLocalDiskAsString":"300M","
CacheAverageSearchTime":0,"CachePutRate":166,"CacheOnDiskHitRate":8,"
CacheMetrics":{"Cache11":[7,83,83],"Cache12":[6,83,83]},"CacheRemoveRate":0,"
CacheOffHeapHitRate":0,"Searchable":false,"CacheOnDiskMissRate":166,"CacheNames"
:["Cache11","Cache12"],"TransactionRolledBackCount":0,"CacheInMemoryHitRate":5,"
WriterQueueLength":0,"CacheOffHeapMissRate":0,"Transactional":false,"
CacheHitRate":13,"TransactionCommitRate":0,"CacheExpirationRate":0,"
CacheUpdateRate":0,"MaxBytesLocalHeap":104857600,"CacheAverageGetTime":0.061820637,
"TransactionRollbackRate":0,"CacheEvictionRate":0,"CacheInMemoryMissRate":174,"
MaxBytesLocalDisk":314572800,"MaxBytesLocalOffHeapAsString":"200M","
CacheSearchRate":0,"TransactionCommittedCount":0,"TransactionTimedOutCount":0,"
Status":"STATUS_ALIVE","MaxBytesLocalOffHeap":209715200,"WriterMaxQueueSize":0,"
StatisticsEnabled":true,"MaxBytesLocalHeapAsString":"100M","CacheMissRate":166},
"agentId":"embedded","version":"1.0.0-SNAPSHOT"}]
```


Note: When no client IDs are specified in the request, all of the clients' cacheManagers are returned. However, if the number of clients is more than the default maximum of 64, then an error is returned in the JSON response. The VM argument `com.terracotta.agent.defaultMaxClientsToDisplay` can be used to change the maximum number of clients to display.

Get Specific CacheManager Details

```
/agents/cacheManagers?show=CacheInMemoryHitRate&show=CacheHitRate&show=CacheAverageGetTime
```

This URI returns a JSON array with only the specified statistics:

```
[{"name":"CM1","attributes":{"CacheAverageGetTime":0.26357448,"CacheHitRate":47,
"CacheInMemoryHitRate":3},"agentId":"embedded","version":"1.0.0-SNAPSHOT"}]
```

Configuration attributes (for example, `MaxBytesLocalHeap`) can also be specified with the `show` query parameter.

Possible HTTP Status Codes for GET or HEAD

404 – Specified resource is not found.

OPTIONS

Retrieves the WADL describing all of the operations available on the specified resources.

Examples

The following OPTIONS examples are organized by task and URI. Examples executed against standalone nodes show a base URI ending in "/tc-management-api/", while those executed against a TMS have a base URI ending in "/tmc/api/".

Return WADL With Available Agent Operations

```
/agents;ids=client01,client02
```

The following is an example of a WADL returned by an embedded agent:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
<doc xmlns:jersey="http://jersey.java.net/"
    jersey:generatedBy="Jersey: 1.9.1
    09/14/2011 02:05 PM"/>
<grammars/>
<resources base="http://localhost:9888/tc-management-api/">
    <resource path="agents">
        <method name="GET" id="getAgents">
            <response>
                <representation mediaType="application/json"/>
            </response>
        </method>
        <resource path="/info">
            <method name="GET" id="getAgentsMetadata">
                <response>
                    <representation mediaType="application/json"/>
                </response>
            </method>
        </resource>
    </resource>
</resources>
</application>
```

OPTIONS

Return WADL With Available CacheManager "Config" Operations

```
/agents/cacheManagers/configs
```

OPTIONS using /configs:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
<doc xmlns:jersey="http://jersey.java.net/"
    jersey:generatedBy="Jersey: 1.9.1 09/14/2011 02:05 PM"/>
<grammars/>
<resources base="http://localhost:9888/tc-management-api/">
  <resource path="agents/cacheManagers/configs">
      <method name="GET" id="getCacheManagerConfig">
        <response>
          <representation mediaType="application/xml"/>
        </response>
      </method>
    </resource>
  </resources>
</application>
```

Return a WADL With Available Operations on a Specific Cache

```
/agents/cacheManagers;names=foo,goo/caches;names=bar
```

Returns information on the cache "bar" from all CacheManagers "foo" and "goo" on any agent reachable by the TMS:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
  <doc xmlns:jersey="http://jersey.java.net/"
      jersey:generatedBy="Jersey: 1.9.1 09/14/2011 02:05 PM"/>
  <grammars/>
  <resources base="http://localhost:9889/api/">
    <resource path="agents/cacheManagers;names=foo,goo/caches;names=bar">
      <method name="GET" id="getCaches">
        <response>
          <representation mediaType="application/json"/>
        </response>
      </method>
      <method name="PUT" id="createCache"/>
      <method name="DELETE" id="deleteCache"/>
      <resource path="/statistics">
        <method name="DELETE" id="wipeStatistics"/>
      </resource>
    </resource>
  </resources>
</application>
```

For information from specific agents, specify the agent ID:

```
/agents;ids=client01,client02/CacheManagers=foo,bar/caches=baz,goo
```

Returns a WADL, as shown above, but with more detailed resource locations (and more caches).

Used with an embedded web service, this URI returns information for the specified caches found on that agent only.

PUT

Creates the specified resource or updates a resource representation.

Allowed Resource Updates

Updating a resource representation means editing the value of one of the following boolean cache attributes:

- Enabled – Enable (true, DEFAULT) or disable (false) the cache.
- StatisticsEnabled – Enable (true, DEFAULT) or disable (false) statistics gathering for the cache. Disabling statistics can improve a cache's performance, but limits monitoring capabilities. Note that if statistics are disabled, then sampled statistics are automatically disabled.
- SampledStatiscsEnabled – Enable (true, DEFAULT) or disable (false) sampled statistics. Sampled statistics are used for providing averages and other aggregate values. If sampled statistics are enabled, statistics gathering is automatically enabled.

Examples

Create a New Cache

```
/agents;ids=MyConnectionGroup_MyEhcache/cacheManagers;names=foo/caches;names=baz
```

This URI creates the cache "baz" under the CacheManager "foo" on the Ehcache node with agent ID "MyConnectionGroup_MyEhcache". baz has the defaultCache configuration.

Update a Cache Attribute

```
/agents;ids=MyConnectionGroup_MyEhcache/cacheManagers;names=foo/caches;names=baz
```

This URI can update the cache attributes as specified in the content. For example, to turn off statistics gathering for the cache "baz", use the following content:

```
{"attributes":{"StatisticsEnabled":false}}
```

See Allowed Resource Updates for more information on updating cache configuration.

Possible HTTP Status Codes for PUT

201 – The operation was successful. 204 – The cache was successfully updated. 400 – The URI does not specify a single resource. 409 – The resource with the given name already exists.

Using Query Parameters in URIs

GET and HEAD HTTP operations can execute queries on specific resources. Query parameters are executed using the `show` parameter:

```
/agents[;ids={comma_sep_agent_ids}]/cacheManagers[;names={comma_sep_cache_manager_names}]
    /caches[;names={comma_sep_cache_names}?show=[parameter]&show=[parameter]
```

For example, to retrieve the values for the parameters `HasWriteBehindWriter` and `MaxBytesLocalDiskAsString` for the CacheManager CM1 on an Ehcache with the ID "foo", use the

following:

```
/agents;ids=foo/cachemanagers;names=CM1?show=HasWriteBehindWriter?
    show=MaxBytesLocalDiskAsString
```

This query returns a JSON object similar to the following:

```
[{"name":  "CM1","attributes":  {"HasWriteBehindWriter":true,
    "MaxBytesLocalDiskAsString":"300M"},"guid":"95d40b093c9f44389f3cc122fbe1c30b",
"agentId":"embedded","version":"1.0.0"}]
```

{/toc-zone}

API Version

All REST APIs should have the same version to avoid potential issues due to differing features and capabilities (see the section on version mismatches. Note that the REST API version is unrelated to the version of Terracotta products or any other Terracotta APIs.

You can discover the API version of connected REST agents using a GET operation with an /agents/info URI.

Version Mismatches

Differences in REST API versions can affect the features and functionality offered by the monitoring tools you create. Over time, version mismatches can arise between the TMS and TSA (when using an external TMS), and between the TMS and standalone nodes.

The TMS may be able to compensate agents with API versions older than its own version by exposing only their available capilities. Newer agent API versions can cause inconsistent behavior or malfunction if the TMS is unable to handle unfamiliar schema, functionality, or other differences in APIs.

JSON Schema

Use the schema as a guide to parsing the JSON objects returned by the REST API, and to validate the structure of data your scripts or RIA sends to agents.

Note that the schema is subject to change between API versions. You can use the REST API URIs to get examples of the JSON schema for the following:

- cacheManager
- cache
- cacheConfig
- cacheStatisticsSample

REST API for TSA

You can use the REST API to query the TMS regarding any connected TSA.

Statistics

Use the following URI extensions with the base extension /agents/statistics to return statistical information.

DGC Runs

Get statistics on the last 1000 DGC runs:

```
/dgc
```

Server Statistics

Get statistics k, l, and m for servers a, b, c:

```
/servers;names=a,b,c?show=k,l,m
```

If no "names" are specified, statistics for all servers are requested. If "show" is omitted, all statistics are requested.

Client Statistics

Get statistics k, l, and m for clients x, y, and z:

```
/clients;ids=x,y,z?show=k,l,m
```

If no "ids" are specified, statistics for all clients are requested. If "show" is omitted, all statistics are requested.

Topology Views

Use the following URI extensions with the base extension /agents/topologies to return topological information.

To get a complete cluster topology (all servers and clients), end the base extension with a forward slash ("/"):

```
/agents/topologies/
```

To get only servers a, b, and c:

```
/servers;names=a,b,c
```

If no "names" are specified, all servers are included.

To get only clients x, y, and z:

```
/clients;ids=x,y,z
```

If no "ids" are specified, all clients are included.

Configuration

Use the following URI extensions with the base extension /agents/configurations to return configuration information.

To get the configuration settings for all servers and clients, end the base extension with a forward slash ("/"):

```
/agents/configurations/
```

To get only servers a, b, and c:

```
/servers;names=a,b,c
```

If no "names" are specified, all servers are included.

To get only clients x, y, and z:

```
/clients;ids=x,y,z
```

If no "ids" are specified, all servers are included.

Diagnostics

Use the following URI extensions with the base extension /agents/disagnostics to return information useful in diagnosing trouble or initiate a DGC cycle.

Thread Dumps

Get a full thread dump from all servers and clients:

```
/threadDump
```

Get a thread dump from servers a, b, and c:

```
/threadDump/servers;names=a,b,c
```

If no "names" are specified, all servers are included.

Get a thread dump from clients x, y, and z:

```
/threadDump/clients;ids=x,y,z
```

If no "ids" are specified, all clients are included.

Thread dumps are written to the logs of their respective nodes. To have all generated thread dumps saved to a zip file, use `threadDumpArchive` instead of `threadDump`.

To write cluster state information (including, for example, on locks) in addition to thread dumps for each node, use `dumpClusterState` instead of `threadDump`. This action generates substantially more information than getting only thread dumps.

DGC Cycles

To initiate a DGC cycle, post:

```
/dgc
```

Backups

You can initiate backups of the cluster data by posting with the following URI extension:

```
/agents/backups/
```

To get the status of a backup ("true" for a backup in progress), use a GET operation with the same URI extension:

```
/agents/backups/
```

Note that backup operations involve the entire TSA and cannot be delegated to specific servers.

Operator Events

You can return operator events using the URI extension /agents/operatorEvents. To limit the size of the returned data, use a `sinceWhen` query.

To get operator events for the last ten minutes:

```
/agents/operatorEvents?sinceWhen=10m
```

To limit events to certain types, add `eventTypes`:

```
/agents/operatorEvents?eventTypes=ERROR,WARN
```

These parameters can be combined:

```
/agents/operatorEvents?sinceWhen=10m&eventTypes=ERROR,WARN
```

Logs

You can return logs using the URI extension /agents/logs. To limit the size of the returned data, use a `sinceWhen` query.

To get logs for the last ten minutes:

```
/agents/logs?sinceWhen=10m
```

Integrating with Nagios XI

You can monitor Terracotta nodes using Nagios XI by creating a Nagios plugin. A Nagios plugin can query the Terracotta Management Server (TMS) for information through the TMS REST interface, or directly through a node's REST interface.

Plugins can be written in a variety of languages, and should follow published guidelines.

This document provides an example using a shell script.

Monitoring the NODE_LEFT Event

When a node leaves a Terracotta cluster, it generates a `node.left` event. The following shell script can report this type of event in Nagios XI.

```
#!/bin/bash

# Parameters
# ----------
SERVER=$1    # The IP address or resolvable hostname of a Terracotta server.
PORT=$2      # The Terracotta server's tsa-group-port (9530 by default).
INTERVAL=$3  # How far back in time, in minutes, to search for the event.


RESTURL="http://${SERVER}:${PORT}/tc-management-api/agents/operatorEvents?sinceWhen=${INTERVAL}m"

GET_INFO=`curl "$RESTURL" -s | grep left`
NB_LINES=`echo $GET_INFO | wc -l`
if [[ $NB_LINES -gt 0 ]]; then
       SERVER_LIST=''
       for i in `echo $GET_INFO | sed 's/.*Node\(.*\)left the cluster.*/\1/g'`; do SERVER_LIST="$
       echo $SERVER_LIST
       CHECK="NODE_LEFT"
else
       CHECK="NO_EVENT"
fi

if [[ "$CHECK" == "NODE_LEFT" ]]; then
   echo "NODE LEFT EVENT: $SERVER_LIST"
   exit 2
elif [[ "$CHECK" == "NO_EVENT" ]]; then
   echo "No NODE LEFT Event: ${SERVER}"
   exit 0
else
   echo "Check failed"
   exit 3
fi
```

Note that the script's exit codes follow the standard required for Nagios plugins:

Value	Status	Description
0	OK	The plugin was able to check the service and it appeared to be functioning properly.
1	Warning	The plugin was able to check the service, but it appeared to be above some "warning" threshold or did not working properly.

2	Critical	The plugin detected that either the service was not running or it was above some "critical" threshold.
3	Unknown	Invalid command line arguments were supplied to the plugin or low-level failures internal to the plugin occurred (such as unable to fork or open a tcp socket) that prevent it from performing the specified operation. Higher-level errors (such as name resolution errors or socket timeouts) are outside of the control of plugins and should generally NOT be reported as UNKNOWN states.

Once you create the script, you must install it in Nagios. A number of tutorials on installing Nagios XI plugins are available on the Internet.

You can generalize the script to find other events by editing the REGEX pattern. Or edit the RESTURL to return other types of information.

Troubleshooting the Terracotta Management Server

The Terracotta Management Server (TMS) is easy to set up and operate. In cases where an error occurs, consult this page for more information.

Each section below addresses an error or set of errors that you may encounter.

Setup Errors

500 Problem Accessing the Keychain File

Problem: After configuring and attempting to use the LDAP or AD URL, a message similar to the following is seen:

```
Problem accessing /tmc/setupAuth. Reason:

impossible to initialize the keychain
...
~/.tc/mgmt/keychain doesn't point to a valid file
```

Cause: The keychain file does not exist in the expected location.

Solution: Create the file `keychain` in `$(user.home)/.tc/mgmt` while adding the first entry:

```
bin/keychain.sh -c -O -S ~/.tc/mgmt/keychain ldap://admin@localhost:1389
```

Cannot Retrieve Entry for LDAP or Active Directory User

Problem: After configuring and attempting to use the LDAP or AD URL, a message similar to "Impossible to retrieve systemUsername password from the keychain : ldap://admin@localhost:1389" appears.

Cause: The keychain does not contain an entry matching the system user specified.

Solution: Create a correct entry for the specified user. For the example above, the password in the keychain file should be keyed with "ldap://admin@localhost:1389".

Connections Errors

Connection Refused

Problem: An attempt to add a connection to a managed agent is rejected.

Cause: The agent is unreachable or not running.

Solution: Check the following:

- The URI in the connection setup is correct.

- The network connection to the node running the agent is working.
- The agent process is running on the expected node.

404 Connection Not Found

Problem: An attempt to add a connection returns a 404 status code.

Cause: The URI used in the connection setup is incorrect or malformed.

Solution: Check the following:

- The URI in the connection setup is correct.
- The port used in the URI is correct (by default: 9888 for BigMemory Go, 9530 for BigMemory Max).

"A message body reader for Java class ... was not found"

Problem: An attempt to add a connection causes the exception "A message body reader for Java class java.util.Collection, and Java type java.util.Collection, and MIME media type unknown/unknown was not found"

Cause: The URI used in the connection setup is incorrect or malformed.

Solution: Check the following:

- The URI in the connection setup is correct.
- The port used in the URI is correct (by default: 9888 for BigMemory Go, 9530 for BigMemory Max).

Connection Timed Out

Problem: An attempt to add a connection fails because the TMS has failed to reach the agent within the timeout limit.

Cause: The agent is unreachable or not running.

Solution: Check the following:

- The URI in the connection setup is correct.
- The network connection to the node running the agent is working.
- The agent process is running on the expected node.

Unexpected End of File From Server

Problem: An attempt to add a connection fails with an EOF error.

Cause: An unsecure connection is being attempted but the agent is set up to use SSL.

Solution: Ensure that the URI is using "https://" not "http://".

"Unrecognized SSL Message, plaintext connection?"

Problem: An attempt to add a connection fails with the error "Unrecognized SSL Message, plaintext connection?".

Cause: A secure connection is being attempted but the agent is not set up to use SSL.

Solution: Ensure that the URI is using "http://" not "https://", or set up SSL on the agent.

Missing Keychain Entry

Problem: An attempt to add a connection fails with the error "Missing keychain entry for URL <agent-url>".

Cause: A connection is being attempted to an agent with identity assertion, but the TMS keychain cannot find that agent's entry.

Solution: Add an entry for the agent using the add-tc-agent script.

401 Unauthorized

Problem 1: An attempt to add a connection to an agent configured with identity assertion returns a 401 status code.

Cause: The agent's public key cannot be found in the the TMS truststore.

Solution: Import the agent's public key into the TMS truststore.

Problem 2: An attempt to add a connection to an agent configured with identity assertion over SSL returns a 401 status code. Errors containing `unknown_certificate` (in the agent log) and

```
PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException:
unable to find valid certification path to requested target
```

(in the TMS log) appear in the logs (see the list of SSL-related logged errors).

Cause: The TMS public key cannot be found in the the agent's truststore.

Solution: Import the TMS public key into the agent's truststore.

Logged SSL Connection Errors

Certain issues will cause exceptions or other errors to occur in the logs when an SSL-enabled connection is attempted. The following list shows parts of log messages that indicate specific errors:

- `keyMaterial=null`
  The connection URI has not been added to the keychain.
- `unknown_certificate` (in the agent log) and
  `PKIX path building failed:`
  `sun.security.provider.certpath.SunCertPathBuilderException:`
  `unable to find valid certification path to requested target`

(in the TMS log)
The agent is not using (or cannot find) its keystore.

- `unknown_certificate` (in the agent log) and
  `the counterpart is not ssl compliant` (in the tms log)
  The agent is not configured to use SSL (or not configured correctly). Confirm that SSL is set up as shown above.
- `unknown_certificate` (in the TMS log)
  Identity assertion (basic TMS security, or IA) is being used over SSL, but the IA URI has not been added to the keychain file. For example:

```
bin/keychain.sh ~/.tc/mgmt/keychain https://localhost:9443/tmc/api/assertIdentity
```

In addition, ensure that the TMS container is configured to use tms-keystore and tms-truststore.

Runtime Errors

If CacheException is being thrown as a result of an attempt to perform certain operations in the TMC, see this issue.

Display Errors

Bad Cache or CacheManager Names

Using the following characters in the names of caches or CacheManagers causes display and runtime errors: `% | ; , / # & * " < ?`

Issues caused can include statistics not appearing correctly in the **Overview** panel, pop-up TMC error messages in response to an attempt to view cache configuration, and runtime CacheException errors.

Sizing Errors

The TMC displays cache sizing information on certain panels. If there appear to be errors in the way sizing information is displayed for nonstop caches, the sizing operation may be timing out (it uses the nonstop timeout value). You can tune the sizing operation's timeout value by setting the bulk-loading timeout multiplier.

The Terracotta Server Array

For BigMemory Max 4.0, Quartz Scheduler, and Terracotta Web Sessions

Introduction

The Terracotta Server Array (TSA) provides the platform for Terracotta products and the backbone for Terracotta clusters. A Terracotta Server Array can vary from a basic two-node tandem to a multi-node array providing configurable scale, high performance, and deep failover coverage.

The main features of the Terracotta Server Array include:

- **Distributed In-memory Data Management** – Manages 10-100x more data in memory than data grids
- **Scalability Without Complexity** – Simple configuration to add server instances to meet growing demand and facilitate capacity planning
- **High Availability** – Instant failover for continuous uptime and services
- **Configurable Health Monitoring** – Terracotta HealthChecker for inter-node monitoring
- **Persistent Application State** – Automatic permanent storage of all current shared in-memory data
- **Automatic Node Reconnection** – Temporarily disconnected server instances and clients rejoin the cluster without operator intervention

New for BigMemory Max 4.0

The 4.0 TSA is an in-memory data platform, where all data is kept in memory, providing faster, more consistent, and more predictable access to data. With resource management, if you have more data than memory available, the TSA protects itself from going over its limit through data eviction and throttling. In most cases, it will recover and come back to its normal working state automatically. In addition, three systems are available to protect data: the Fast Restart feature, active-mirror server groups, and backups.

Fast Restartability for Data Persistence

BigMemory's Fast Restart feature is now integrated into the TSA, providing crash resilience with quick recovery, plus a consistent record of the entire in-memory data set, no matter how large. For more information, refer to Fast Restartability.

No More Temporary Disk Storage

The new implementation has no option for temporary disk storage. All data handled by the TSA is in-memory only. With no overflow or swapping to server disks, the TSA-managed data set is always exactly what is in memory. (Note that localTempSwap continues to be an option for unclustered BigMemory Go.)

Resource Management

Resource management provides better control over the TSA's in-memory data through time, size, and count limitations. This enables automatic handling of, and recovery from, near-memory-full conditions. For more information, refer to Automatic Resource Management.

Predictable Eviction Strategy

Based upon user-configured time, size, and count limitations, the TSA's 3-pronged eviction strategy works automatically to ensure predictable behavior when memory becomes full. For more information, refer to Eviction.

Continuous Uptime

Improvements to provide continuous availability of data include flexibility in server startup sequencing, better utilization of extra mirrors in mirror groups, multi-stripe backup capability, optimizations to bulk load, and performance improvements for data access on rejoin. In addition, the TSA no longer uses Oracle Berkeley DB, enabling in-memory data to be ready for use much more quickly after any planned or unplanned restart.

Terracotta Management Console (TMC)

The expanded TMC replaces the Developer Console and Operations Center as the integrated platform for monitoring, managing, and administering all Terracotta deployments. There is also support for additional REST APIs for management and monitoring. For more information, start with Terracotta Management Console.

Additional Security Features

Active Directory (AD) and Lightweight Directory Access Protocol (LDAP) support on Terracotta servers, and custom SecretProvider on Terracotta clients. For more information, refer to Securing Terracotta Clusters and Setting up LDAP-based Authentication.

No More DSO, plus Simplified Configuration

DSO configuration has been deprecated, and the tc-config has a new format. Most of the elements are the same, but the structure is revised. For more information, refer to Terracotta Configuration Reference.

Definitions and Functional Characteristics

The major components of a Terracotta installation are the following:

- **Cluster** – All of the Terracotta server instances and clients that work together to share application state or a data set.
- **Terracotta client** – Terracotta clients run on application servers along with the applications being clustered by Terracotta. Clients manage live shared-object graphs.
- **Terracotta server instance** – A single Terracotta server. An *active* server instance manages Terracotta clients, coordinates shared objects, and persists data. Server instances have no awareness of the clustered applications running on Terracotta clients. A mirror (sometimes called "hot standby") is a live backup server instance which continuously replicates the shared data of an active server instance, instantaneously replacing the active if the active fails. **Mirror servers add failover coverage within each mirror group**.
- **Terracotta mirror group** – A unit in the Terracotta Server Array. Sometimes also called a "stripe," a mirror group is composed of exactly one active Terracotta server instance and at least one mirror Terracotta server instance. The active server instance manages and persists the fraction of shared data allotted to its mirror group, while each mirror server in the mirror group replicates (or mirrors) the shared data managed by the active server. **Mirror groups add capacity to the cluster**. The mirror servers are optional but highly recommended for providing failover.

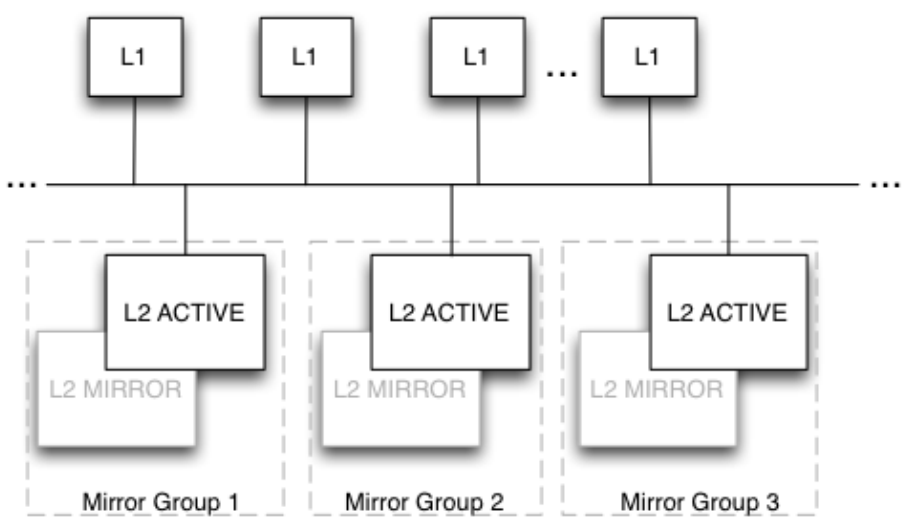
- **Terracotta Server Array** – The platform, consisting of all of the Terracotta server instances in a single cluster. Clustered data, also called in-memory data, or shared data, is partitioned equally among active Terracotta server instances for management and persistence purposes.

TIP: Nomenclature

This documentation may refer to a Terracotta server instance as L2, and a Terracotta client (the node running your application) as L1. These are the shorthand references used in Terracotta configuration files.

Figure 1 illustrates a Terracotta cluster with three mirror groups. Each mirror group has an active server and a mirror, and manages one third of the shared data in the cluster.

Fig. 1: A Server Array With 3 Mirror Groups



A Terracotta cluster has the following functional characteristics:

- Each mirror group automatically elects one active Terracotta server instance. There can never be more than one active server instance per mirror group, but there can be any number of mirrors. However, a performance overhead may become evident when adding more mirror servers due to the load placed on the active server by having to synchronize with each mirror.
- Every mirror group in the cluster must have a Terracotta server instance in active mode before the cluster is ready to do work.
- The shared data in the cluster is automatically partitioned and distributed to the mirror groups. The number of partitions equals the number of mirror groups. In Fig. 1, each mirror group has one third of the shared data in the cluster.
- Mirror groups **cannot** provide failover for each other. Failover is provided *within* each mirror group, not across mirror groups. This is because mirror groups provide scale by managing discrete portions of the shared data in the cluster -- they do not replicate each other. In Fig. 1, if Mirror Group 1 goes down, the cluster must pause (stop work) until Mirror Group 1 is back up with its portion of the shared data intact.
- Active servers are self-coordinating among themselves. No additional configuration is required to coordinate active server instances.
- Only mirror server instances can be hot-swapped in an array. In Fig. 1, the L2 MIRROR servers can be shut down and replaced with no affect on cluster functions. However, to add or remove an entire mirror group, the cluster must be brought down. Note also that in this case the original Terracotta configuration file is still in effect and no new servers can be added. Replaced mirror servers must

have the same address (hostname or IP address). If you must swap in a mirror with a different configuration, see Changing Cluster Topology in a Live Cluster.

Where To Go Next

For more about	Go to
Architecture	Terracotta Server Array Architecture
High Availability	Configuring Terracotta Clusters for High Availability
Configuration	Working with Terracotta Configuration Files
Resource Management	TSA Operations – Automatic Resource Management
Live Data Backup	TSA Operations – Distributed In-memory Data Backup

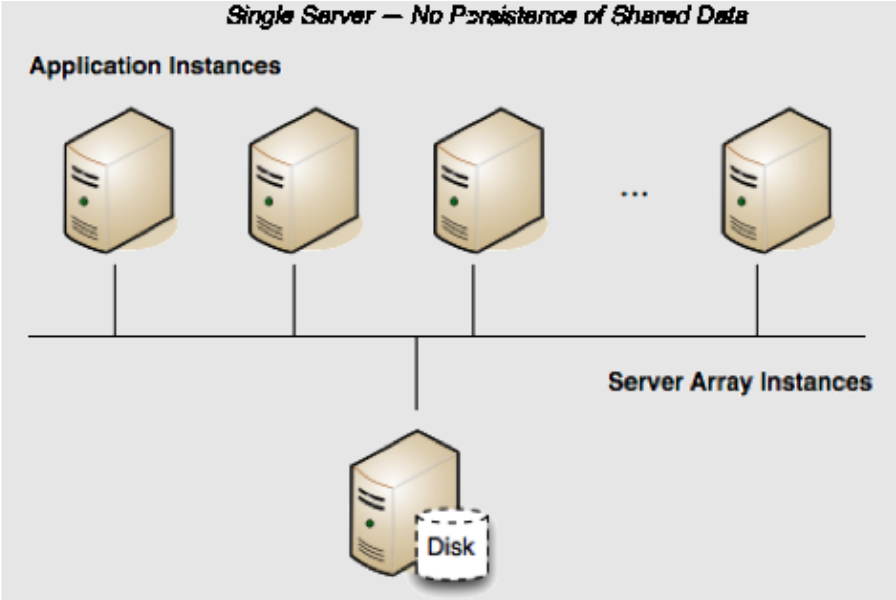
Terracotta Server Array Architecture

A Terracotta Server Array (TSA) can vary from a basic two-node tandem to a multi-node array. The Terracotta cluster can be configured into a number of different setups to serve both deployment stage and production needs. This page shows you how to add cluster reliability, availability, and scalability.

Terracotta Cluster in Development

Persistence: No | Failover: No | Scale: No

In a development environment, persisting shared data is often unnecessary and even inconvenient. Running a single-server Terracotta cluster without persistence is a good solution for creating an efficient development environment.



By default, a Terracotta server has Fast Restartability disabled, which means it will not persist data after a restart. Its configuration could look like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-8.xsd">
 <servers>
   <server name="Server1">
     <data>/opt/terracotta/server1-data</data>
     <tsa-port>9510</tsa-port>
     <offheap>
       <enabled>true</enabled>
       <maxDataSize>2g</maxDataSize>
     </offheap>
   </server>
  </servers>
 ...
</tc:tc-config>
```

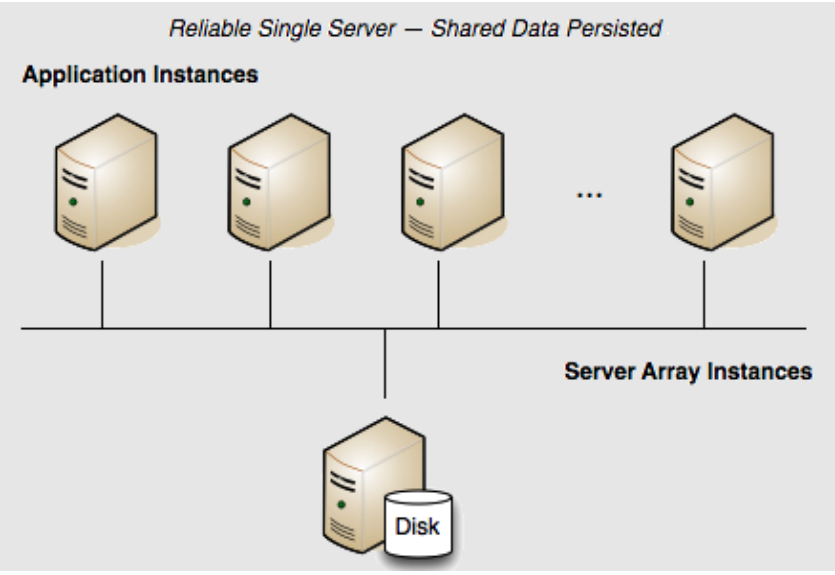
Persistence: No | Failover: No | Scale: No

If this server goes down, the application state (all clustered data) in the shared memory is lost. In addition, when the server is up again, all clients must be restarted to rejoin the cluster. Note that servers are required to run with off-heap, and that shared data is also lost.

Terracotta Cluster with Reliability

Persistence: Yes | Failover: No | Scale: No

The configuration above may be advantageous in development, but if shared in-memory data must be persisted, the server should be configured to use its local disk. Terracotta servers achieve data persistence with the Fast Restart feature.



Fast Restartability

The Fast Restart feature provides enterprise-ready crash resilience by keeping a fully consistent, real-time record of your in-memory data. After any kind of shutdown — planned or unplanned — the next time your application starts up, all of your BigMemory Max data is still available and very quickly accessible.

The Fast Restart feature persists the real-time record of the in-memory data in a Fast Restart store on the server's local disk. After any restart, the data that was last in memory (both heap and off-heap stores) automatically loads from the Fast Restart store back into memory. In addition, previously connected clients are allowed to rejoin the cluster within a window set by the `<client-reconnect-window>` element.

To configure the Terracotta server for Fast Restartability, add and enable the `<restartable>` and the `<offheap>` elements in the tc-config.xml.

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-8.xsd">
 <servers>
   <server name="Server1">
     <data>/opt/terracotta/server1-data</data>
     <tsa-port>9510</tsa-port>
     <offheap>
```

```
        <enabled>true</enabled>
        <maxDataSize>2g</maxDataSize>
      </offheap>
   </server>
   <!-- Fast Restartability must be added explicitly. -->
   <restartable enabled="true"/>
   <!-- By default the window is 120 seconds. -->
   <client-reconnect-window>120</client-reconnect-window>
 </servers>
 ...
</tc:tc-config>
```

Terracotta server memory allocation

Fast Restartability requires that `<offheap>` be enabled with a minimum setting for `maxDataSize` of 512MB. Refer to the following table for store size guidelines for servers in the TSA.

When Off-heap is set between	Configure at least this much Heap
4 - 10 GB	1 GB (Note: The default heap size is 2 GB.)
10 - 100 GB	2 GB
100 GB - 1 TB +	3 GB +
(Off-heap is configured in the tc-config.xml)	(Heap is configured using the -Xmx Java option)

Disk usage

Fast Restartability requires a unique and explicitly specified path. The default path is the Terracotta server's home directory. You can customize the path using the `<data>` element in the server's `tc-config.xml` configuration file.

The Terracotta Server Array can be configured to be restartable in addition to including searchable caches, but both of these features require disk storage. When both are enabled, be sure that enough disk space is available. Depending upon the number of searchable attributes, the amount of disk storage required might be 3 times the amount of in-memory data.

It is highly recommended to store the search index (`<index>`) and the Fast Restart data (`<data>`) on separate disks.

Client Reconnect Window

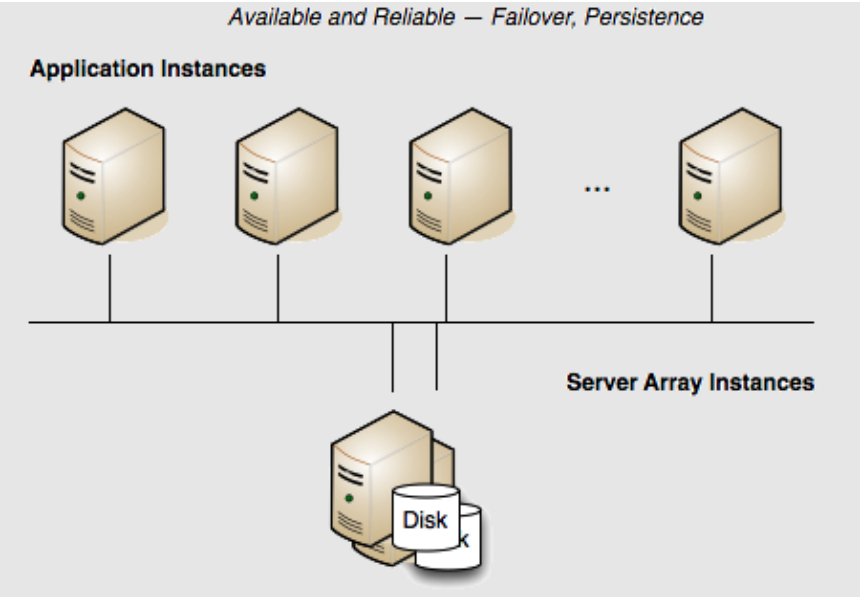
The `<client-reconnect-window>` does not have to be explicitly set if the default value is acceptable. However, in a single-server cluster, `<client-reconnect-window>` is in effect only if restartable mode is enabled.

Terracotta Server Array with High Availability

Persistence: Yes | Failover: Yes | Scale: No

The example above presents a reliable but *not* highly available cluster. If the server fails, the cluster fails. There is no redundancy to provide failover. Adding a mirror server adds availability because the mirror serves as a "hot standby" ready to take over for the active server in case of a failure.

Persistence: Yes | Failover: Yes | Scale: No



In this array, if the active Terracotta server instance fails, then the mirror instantly takes over and the cluster continues functioning. No data is lost.

The following Terracotta configuration file demonstrates how to configure this two-server array:

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-8.xsd">
 <servers>
   <server name="Server1">
     <data>/opt/terracotta/server1-data</data>
     <tsa-port>9510</tsa-port>
     <tsa-group-port>9530</tsa-group-port>
     <offheap>
       <enabled>true</enabled>
       <maxDataSize>2g</maxDataSize>
     </offheap>
   </server>
   <server name="Server2">
     <data>/opt/terracotta/server2-data</data>
     <tsa-port>9510</tsa-port>
     <tsa-group-port>9530</tsa-group-port>
     <offheap>
       <enabled>true</enabled>
       <maxDataSize>2g</maxDataSize>
     </offheap>
   </server>
   <restartable enabled="true"/>
   <client-reconnect-window>120</client-reconnect-window>
 </servers>
 ...
</tc:tc-config>
```

You can add more mirror servers to this configuration by adding more `<server>` sections. However, a performance overhead may become evident when adding more mirror servers due to the load placed on the active server by having to synchronize with each mirror.

Note: Terracotta server instances must not share data directories. Each server's `<data>` element should point to a different and preferably local data directory.

Starting the Servers

How server instances behave at startup depends on when in the life of the cluster they are started.

In a single-server configuration, when the server is started it performs a startup routine and then is ready to run the cluster (ACTIVE status). If multiple server instances are started at the same time, one is elected the active server (ACTIVE-COORDINATOR status) while the others serve as mirrors (PASSIVE-STANDBY status). The election is recorded in the servers' logs.

If a server instance is started while an active server instance is already running, it syncs up state from the active server instance before becoming a mirror. The active and mirror servers must always be synchronized, allowing the mirror server to mirror the state of the active. The mirror server goes through the following states:

1. **PASSIVE-UNINITIALIZED** – The mirror is beginning its startup sequence and is *not* ready to perform failover should the active fail or be shut down. The server's status light in the Terracotta Management Console (TMC) switches from red to orange.
2. **INITIALIZING** – The mirror is synchronizing state with the active and is *not* ready to perform failover should the active fail or be shut down. The server's status light in the TMC is orange.
3. **PASSIVE-STANDBY** – The mirror is synchronized and is ready to perform failover should the active server fail or be shut down. The server's status light in the TMC switches from orange to cyan.

The active server instance carries the load of sending state to the mirror during the synchronization process. The time taken to synchronize is dependent on the amount of clustered data and on the current load on the cluster. The active server instance and mirrors should be run on similarly configured machines for better throughput, and should be started together to avoid unnecessary sync ups.

The sequence in which servers startup does not affect data. Even if a former mirror server is initialized before the former active server, the mirror server's data is not erased. In the event that a mirror server went offline while the active server was still up, then when the mirror server returns, it remembers that it was in the mirror role. Even if the active server is offline at that point, the mirror server does not try to become the active. It waits until the active server returns, and clients are blocked from updating their data. When the active returns, it will restart the mirror. The mirror's data objects and indices are then moved to the dirty-objectdb-backup directory, and the active syncs its data with the mirror.

Failover

If the active server instance fails and two or more mirror server instances are available, an election determines the new active. Successful failover to a new active takes place only if at least one mirror server is fully synchronized with the failed active server; successful client failover (migration to the new active) can happen only if the server failover is successful. Shutting down the active server before a fully-synchronized mirror is available can result in a cluster-wide failure.

If the `maxDataSize` on the mirror server is smaller than on the active server, then the mirror server will fail to start and the user will be alerted that the configuration is invalid. If there are multiple mirrors with differing amounts of off-heap configured, then the passive with the smallest `maxDataSize` (that is still greater than or equal to the active's `maxDataSize`) will be elected to be the new active.

TIP: Hot-Swapping Mirror

A mirror can be hot-swapped if the replacement matches the original mirror's <server> block in the Terracotta configuration. For example, the new mirror should use the same host name or IP address configured for the original mirror. For information about swapping in a mirror with a different configuration, refer to Changing Cluster Topology in a Live Cluster.

Terracotta server instances acting as mirrors can run either in restartable mode or non-persistent mode. If a server instance running in restartable mode goes down, and a mirror takes over, the crashed server's data directory is cleared before it is restarted and allowed to rejoin the cluster. Removing the data is necessary because the cluster state could have changed since the crash. During startup, the restarted server's new state is synchronized from the new active server instance.

If both servers are down, and clustered data is persisted, the last server to be active will automatically be started first to avoid errors and data loss.

In setups where data is not persisted, meaning that restartable mode is not enabled, then no data is saved and either server can be started first.

A Safe Failover Procedure

To safely migrate clients to a mirror server without stopping the cluster, follow these steps:

1. If it is not already running, start the mirror server using the start-tc-server script. The mirror server must already be configured in the Terracotta configuration file.
2. Ensure that the mirror server is ready for failover (PASSIVE-STANDBY status). In the TMC, the status light will be cyan.
3. Shut down the active server using the stop-tc-server script.

   NOTE: If the script detects that the mirror server in STANDBY state isn't reachable, it issues a warning and fails to shut down the active server. If failover is not a concern, you can override this behavior with the `--force` flag.

   Clients will connect to the new active server.
4. Restart any clients that fail to reconnect to the new active server within the configured reconnection window.

The previously active server can now rejoin the cluster as a mirror server. If restartable mode had been enabled, its data is first removed and then the current data is read in from the now active server.

A Safe Cluster Shutdown Procedure

A safe cluster shutdown should follow these steps:

1. Shut down the mirror servers using the stop-tc-server script.
2. Shut down the clients. The Terracotta client will shut down when you shut down your application.
3. Shut down the active server using the stop-tc-server script.

To restart the cluster, first start the server that was last active. If clustered data is not persisted, any of the servers could be started first as no data conflicts can take place.

Split Brain Scenario

In a Terracotta cluster, "split brain" refers to a scenario where two servers assume the role of active server (ACTIVE-COORDINATOR status). This can occur during a network problem that disconnects the active and mirror servers, causing the mirror to both become an active server and open a reconnection window for clients (<client-reconnect-window>).

If the connection between the two servers is never restored, then two independent clusters are in operation. This is not a split-brain situation. However, if the connection is restored, one of the following scenarios results:

- No clients connect to the new active server – The original active server "zaps" the new active server, causing it to restart, wipe its database, and synchronize again as a mirror.
- A minority of clients connect to the new active server – The original active server starts a reconnect timeout for the clients that it loses, while zapping the new active server. The new active restarts, wipes its database, and synchronizes again as a mirror. Clients that defected to the new active attempt to reconnect to the original active, but if they do not succeed within the parameters set by that server, they must be restarted.
- A majority of clients connects to the new active server – The new active server "zaps" the original active server. The original active restarts, wipes its database, and synchronizes again as a mirror. Clients that do not connect to the new active within its configured reconnection window must be restarted.
- An equal number of clients connect to the new active server – In this unlikely event, exactly one half of the original active server's clients connect to the new active server. The servers must now attempt to determine which of them holds the latest transactions (or has the freshest data). The winner zaps the loser, and clients behave as noted above, depending on which server remains active. Manual shutdown of one of the servers may become necessary if a timely resolution does not occur.

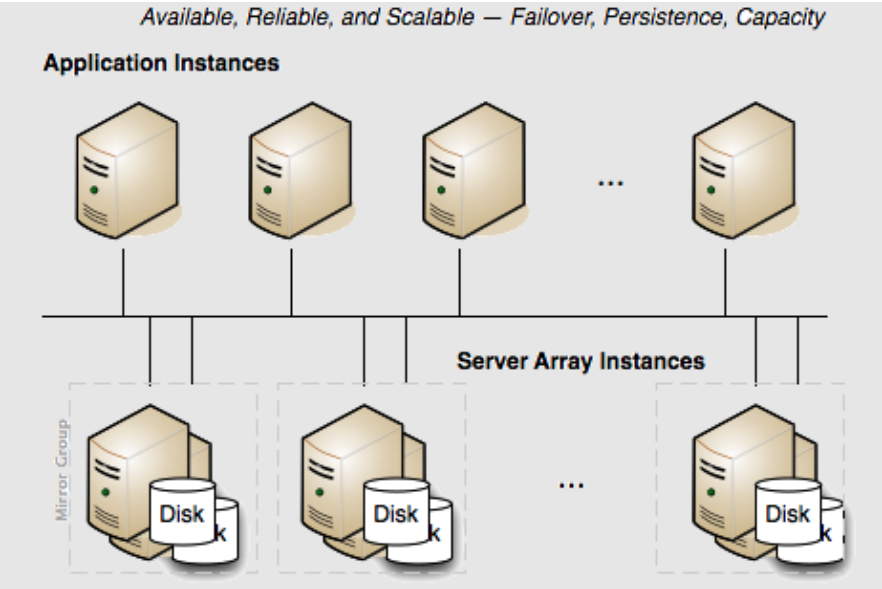
The Terracotta cluster can solve almost all split-brain occurrences without loss or corruption of shared data. However, it is highly recommended that you confirm the integrity of shared data after such an occurrence.

Scaling the Terracotta Server Array

Persistence: Yes | Failover: Yes | Scale: Yes

For capacity requirements that exceed the capabilities of a two-server active-mirror setup, expand the Terracotta cluster using a mirror-groups configuration. Using mirror groups with multiple coordinated active Terracotta server instances adds scalability to the Terracotta Server Array.

Persistence: Yes | Failover: Yes | Scale: Yes



Mirror groups are specified in the `<servers>` section of the Terracotta configuration file. Mirror groups work by assigning group memberships to Terracotta server instances. The following snippet from a Terracotta configuration file shows a mirror-group configuration with four servers:

```
...
<servers>
  <mirror-group election-time="10" group-name="groupA">
    <server name="server1">
    ...
    </server>
    <server name="server2">
    ...
    </server>
  </mirror-group>
  <mirror-group election-time="15" group-name="groupB">
    <server name="server3">
    ...
    </server>
    <server name="server4">
    ...
    </server>
  </mirror-group>
  <restartable enabled="true"/>
</servers>
...
```

In this example, the cluster is configured to have two active servers, each with its own mirror. If server1 is elected active in groupA, server2 becomes its mirror. If server3 is elected active in groupB, server4 becomes its mirror. server1 and server3 automatically coordinate their work managing Terracotta clients and shared data across the cluster.

In a Terracotta cluster designed for multiple active Terracotta server instances, the server instances in each mirror group participate in an election to choose the active. Once every mirror group has elected an active server instance, all the active server instances in the cluster begin cooperatively managing the cluster. The rest of the server instances become mirrors for the active server instance in their mirror group. If the active in a mirror group fails, a new election takes place to determine that mirror group's new active. Clients continue work without regard to the failure.

NOTE: Server vs. Mirror Group

Under <servers>, you may use either <server> or <mirror-group> configurations, but not both. All <server> configurations directly under <servers> work together as one mirror group, with one active server and the rest mirrors. To create more than one stripe, use <mirror-group> configurations directly under <servers>. The mirror group configurations then include one or more <server> configurations.

In a Terracotta cluster with mirror groups, each group, or "stripe," behaves in a similar way to an active-mirror setup (see Terracotta Server Array with High Availability). For example, when a server instance is started in a stripe while an active server instance is present, it synchronizes state from the active server instance before becoming a mirror. A mirror cannot become an active server instance during a failure until it is fully synchronized. If an active server instance running in restartable mode goes down, and a mirror takes over, the data directory is cleared before bringing back the crashed server.

Election Time

The `<mirror-group>` configuration allows you to declare the election time window. An active server is elected from the servers that cast a vote within this window. The value is specified in seconds and the default is 5 seconds. Network latency and the work load of the servers should be taken into consideration when choosing an appropriate window.

In the above example, the servers in groupA can take up to 10 seconds to elect an active server, and the servers in groupB can take up to 15 seconds.

Stripe and Cluster Failure

If the active server in a mirror group fails or is taken down, the cluster stops until a mirror takes over and becomes active (ACTIVE-COORDINATOR status).

However, the cluster cannot survive the loss of an entire stripe. If an entire stripe fails and no server in the failed mirror-group becomes active within the allowed window (based on the election-time setting), the entire cluster must be restarted.

Working with Terracotta Configuration Files

Introduction

Terracotta XML configuration files set the characteristics and behavior of Terracotta server instances and Terracotta clients. The easiest way to create your own Terracotta configuration file is by editing a copy of one of the sample configuration files available with the Terracotta BigMemory Max kit.

Where you locate the Terracotta configuration file, or how your Terracotta server and client configurations are loaded, depends on the stage your project is at and on its architecture. This document covers the following cases:

- Development stage, 1 Terracotta server
- Development stage, 2 Terracotta servers
- Deployment stage

This document discusses cluster configuration in the Terracotta Server Array. To learn more about the Terracotta server instances, see Terracotta Server Array Architecture.

For a comprehensive and fully annotated configuration file, see `config-samples/tc-config-reference.xml` in the Terracotta kit.

Quick Start Configuration

To configure the Terracotta server, create a `tc-config.xml` configuration file, or update the one that is provided in the config-samples/ directory of the BigMemory Max kit. For example:

```
<?xml version="1.0" encoding="UTF-8" ?>
 <tc:tc-config xmlns:tc="http://www.terracotta.org/config"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-8.xsd">
   <servers>
     <server host="localhost" name="My Server Name1">
       <!-- Specify the path where the server should store its data. -->
       <data>/local/disk/path/to/terracotta/server1-data</data>
        <!-- Specify the port where the server should listen for client
        traffic. -->
        <tsa-port>9510</tsa-port>
        <jmx-port>9520</jmx-port>
        <tsa-group-port>9530</tsa-group-port>
        <!-- Enable BigMemory on the server. -->
        <offheap>
          <enabled>true</enabled>
          <maxDataSize>4g</maxDataSize>
        </offheap>
      </server>
     <server host="localhost" name="My Server Name2">
       <data>/local/disk/path/to/terracotta/server2-data</data>
        <tsa-port>9510</tsa-port>
        <jmx-port>9520</jmx-port>
        <tsa-group-port>9530</tsa-group-port>
        <offheap>
          <enabled>true</enabled>
          <maxDataSize>4g</maxDataSize>
```

```
          </offheap>
        </server>
        <!-- Add the restartable element for Fast Restartability (optional). -->
        <restartable enabled="true"/>
      </servers>
      <clients>
        <logs>logs-%i</logs>
      </clients>
    </tc:tc-config>
```

To successfully configure a Terracotta Server Array using the Terracotta configuration file, note the following:

- Two or more servers should be defined in the <servers> section of the Terracotta configuration file.
- <tsa-group-port> is the port used by the Terracotta server to communicate with other Terracotta servers.
- Under <servers>, use either <server> or <mirror-group> configurations, but not a mixture. You may configure multiple servers or multiple mirror groups. <server> instances under <servers> work together as a mirror group. To create more than one stripe, use <mirror-group> instances.
- Terracotta server instances must not share data directories. Each server's <data> element should point to a different and preferably local data directory.
- For data persistence, configure fast restartability. Enabling <restartable> means that the shared in-memory data is backed up and, in case of failure, it is automatically restored. Setting <restartable> to "false" or omitting the <restartable> element are two ways to configure no persistence.
- Each server requires an off-heap store, which allows all data to be stored in-memory, limited only by the amount of memory in your server. Enable <offheap> and set <maxDataSize> to the amount of memory available in your server for data.
- All servers and clients should be running the same version of Terracotta and Java.

How Terracotta Servers Get Configured

At startup, Terracotta servers load their configuration from one of the following sources:

- A default configuration included with the Terracotta kit
- A local or remote XML file

These sources are explored below.

Default Configuration

If no configuration file is specified *and* no `tc-config.xml` exists in the directory in which the Terracotta instance is started, then default configuration values are used.

Local XML File (Default)

The file `tc-config.xml` is used by default if it is located in the directory in which a Terracotta instance is started *and* no configuration file is explicitly specified.

Local or Remote Configuration File

You can explicitly specify a configuration file by passing the -f option to the script used to start a Terracotta server. For example, to start a Terracotta server on UNIX/Linux using the provided script, enter:

```
start-tc-server.sh -f <path_to_configuration_file>
```

where <path_to_configuration_file> can be a URL or a relative directory path. In Microsoft Windows, use `start-tc-server.bat`.

Note: Cygwin (on Windows) is not supported for this feature.

How Terracotta Clients Get Configured

At startup, Terracotta clients load their configuration from one of the following sources:

- Local or Remote XML File
- Terracotta Server
- An Ehcache configuration file (using the `<terracottaConfig>` element) used with BigMemory Max and BigMemory Go.
- A Quartz properties file (using the `org.quartz.jobStore.tcConfigUrl` property) used with Quartz Scheduler.
- A Filter (in `web.xml`) element used with containers and Terracotta Web Sessions.
- The client constructor (`TerracottaClient()`) used when a client is instantiated programmatically using the Terracotta Toolkit.

Terracotta clients can load customized configuration files to specify <client> and <application> configuration. However, the <servers> block of every client in a cluster must match the <servers> block of the servers in the cluster. If there is a mismatch, the client will emit an error and fail to complete its startup.

NOTE: Error with Matching Configuration Files

On startup, a Terracotta client may emit a configuration-mismatch error if its <servers> block does not match that of the server it connects to. However, under certain circumstances, this error may occur even if the <servers> blocks appear to match. The following suggestions may help prevent this error: - Use `-Djava.net.preferIPv4Stack` consistently. If it is explicitly set on the client, be sure to explicitly set it on the server. - Ensure `etc/hosts` file does not contain multiple entries for hosts running Terracotta servers. - Ensure that DNS always returns the same address for hosts running Terracotta servers.

Local or Remote XML File

See the discussion for local XML file (default) in How Terracotta Servers Get Configured.

To specify a configuration file for a Terracotta client, see Clients in Development.

NOTE: Fetching Configuration from the Server

On startup, Terracotta clients must fetch certain configuration properties from a Terracotta server. A client loading its own configuration will attempt to connect to the Terracotta servers named in that configuration. If none of the servers named in that configuration are available, the client cannot complete its startup.

Terracotta Server

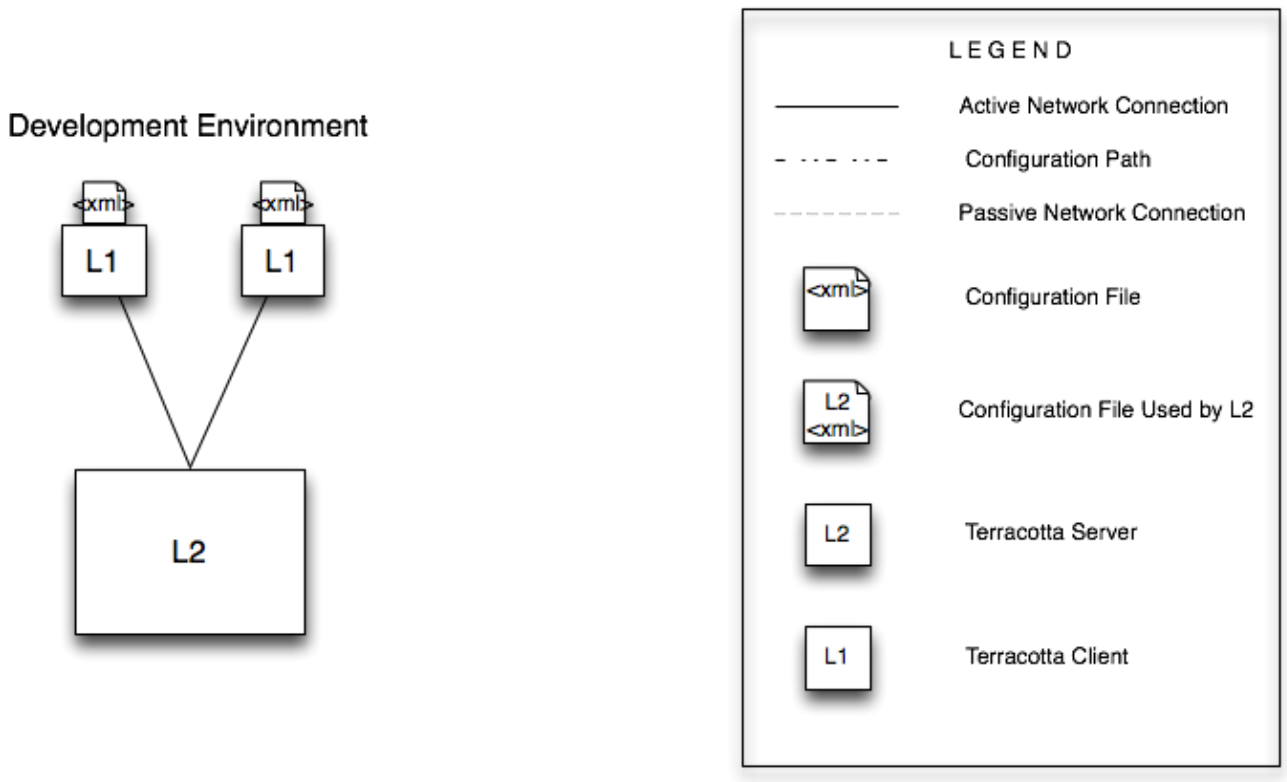
Terracotta clients can load configuration from an active Terracotta server by specifying its hostname and TSA port (see Clients in Production).

Configuration in a Development Environment

In a development environment, using a different configuration file for each Terracotta client facilitates the testing and tuning of configuration options. This is an efficient and effective way to gain valuable insight on best practices for clustering your application with Terracotta.

One-Server Setup in Development

For one Terracotta server, the default configuration is adequate.



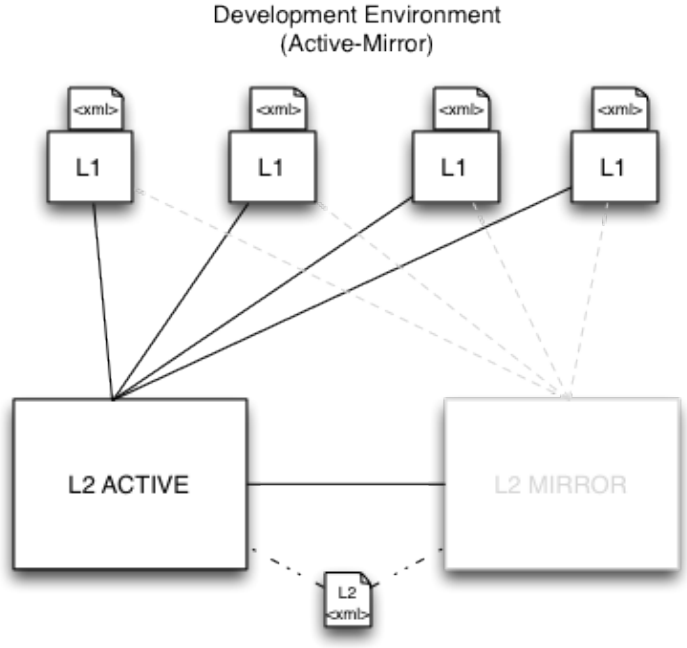
To use the default configuration settings, start your Terracotta server using the `start-tc-server.sh` (or `start-tc-server.bat)` script in a directory that does *not* contain the file `tc-config.xml`:

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.sh
```

To specify a configuration file, use one of the approaches discussed in How Terracotta Servers Get Configured.

Two-Server Setup in Development

A two-server setup, sometimes referred to as an active-mirror setup, has one active server instance and one "hot standby" (the mirror) that should load the same configuration file.



The configuration file loaded by the Terracotta servers must define each server separately using <server> elements. For example:

```
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-8.xsd"
xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...
<!-- Use an IP address or a resolvable host name for the host attribute. -->
  <server host="123.456.7.890" name="Server1">
...
  <server host="myResolvableHostName" name="Server2">
...
</tc:tc-config>
```

Assuming Server1 is the active server, using the same configuration allows Server2 to be the mirror and maintain the environment in case of failover. If you are running both Terracotta servers on the same host, the only port that has to be specified in configuration is the <tsa-port>; the values for <jmx-port> and <l2-group-port> are filled in automatically.

NOTE: Running Two Servers on the Same Host

If you are running the servers on the same machine, some elements in the <server> section, such as <tsa-port> and <server-logs>, must have different values for each server.

Server Names for Startup

With multiple <server> elements, the name attribute may be required to avoid ambiguity when starting a server:

```
start-tc-server.sh -n Server1 -f <path_to_configuration_file>
```

In Microsoft Windows, use `start-tc-server.bat`.

For example, if you are running Terracotta server instances on the same host, you must specify the name attribute to set an unambiguous target for the script.

However, if you are starting Terracotta server instances in an unambiguous setup, specifying the server name is optional. For example, if the Terracotta configuration file specifies different IP addresses for each server, the script assumes that the server with the IP address corresponding to the local IP address is the target.

Clients in Development

You can explicitly specify a client's Terracotta configuration file by passing `-Dtc.config=path/to/my-tc-config.xml` when you start your application with the Terracotta client.

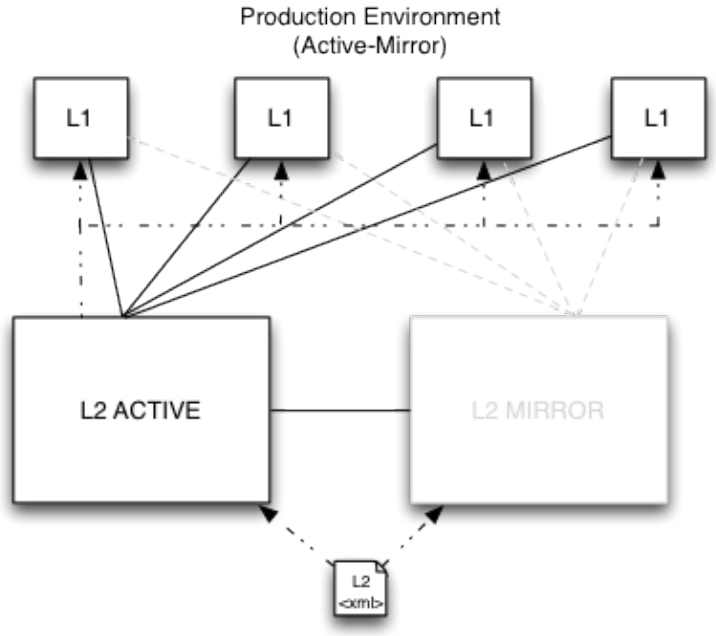
```
-Dtc.config=path/to/my-tc-config.xml -cp classes myApp.class.Main
```

where `myApp.class.Main` is the class used to launch the application you want to cluster with Terracotta.

If `tc-config.xml` exists in the directory in which you run Java, it can be loaded without `-Dtc.config`.

Configuration in a Production Environment

For an efficient production environment, it's recommended that you maintain one Terracotta configuration file. That file can be loaded by the Terracotta server (or servers) and pushed out to clients. While this is an optional approach, it's an effective way to centralize and decrease maintenance.



If your Terracotta configuration file uses "%i" for the hostname attribute in its server element, change it to the actual hostname in production. For example, if in development you used the following:

```
<server host="%i" name="Server1">
```

and the production host's hostname is myHostName, then change the host attribute to the myHostName:

```
<server host="myHostName" name="Server1">
```

Clients in Production

For clients in production, you can set up the Terracotta environment before launching your application.

Setting Up the Terracotta Environment

To start your application with the Terracotta client using your own scripts, first set the following environment variables:

```
TC_INSTALL_DIR=<path_to_local_Terracotta_home>
TC_CONFIG_PATH=<path/to/tc-config.xml>
```

or

```
TC_CONFIG_PATH=<server_host>:<tsa-port>
```

where `<server_host>:<tsa-port>` points to the running Terracotta server. The specified Terracotta server will push its configuration to the Terracotta client.

Alternatively, a client can specify that its configuration come from a server by setting the *tc.config* system propery:

```
-Dtc.config=serverHost:tsaPort
```

If more than one Terracotta server is available, enter them in a comma-separated list:

```
TC_CONFIG_PATH=<server_host1>:<tsa-port>,<server_host2>:<tsa-port>
```

If <server_host1> is unavailable, <server_host2> is used.

Terracotta Products

Terracotta products can set a configuration path using their own configuration files.

For BigMemory Max and BigMemory Go, use the `<terracottaConfig>` element in the Ehcache configuration file (`ehcache.xml` by default):

```
<terracottaConfig url="localhost:9510" />
```

For Quartz, use the `org.quartz.jobStore.tcConfigUrl` property in the Quartz properties file (`quartz.properties` by default):

```
org.quartz.jobStore.tcConfigUrl = /myPath/to/tc-config.xml
```

For Terracotta Web Sessions, use the appropriate elements in `web.xml` or `context.xml` (see Web Sessions Installation).

Binding Ports to Interfaces

Normally, the ports you specify for a server in the Terracotta configuration are bound to the interface associated with the host specified for that server. For example, if the server is configured with the IP address "12.345.678.8" (or a hostname with that address), the server's ports are bound to that same interface:

```
<server host="12.345.678.8" name="Server1">
 ...
 <tsa-port>9510</tsa-port>
 <jmx-port>9520</jmx-port>
 <tsa-group-port>9530</tsa-group-port>
</server>
```

However, in certain situations it may be necessary to specify a different interface for one or more of a server's ports. This is done using the `bind` attribute, which allows you bind a port to a different interface. For example, a JMX client may only be able connect to a certain interface on a host. The following configuration shows a JMX port bound to an interface different than the host's:

```
<server host="12.345.678.8" name="Server1">
 ...
 <tsa-port>9510</tsa-port>
 <jmx-port bind="12.345.678.9">9520</jmx-port>
 <tsa-group-port>9530</tsa-group-port>
</server>
```

Which Configuration?

Each server and client must maintain separate log directories. By default, server logs are written to `%(user.home)/terracotta/server-logs` and client logs to `%(user.home)/terracotta/client-logs`.

To find out which configuration a server or client is using, search its logs for an INFO message containing the text "Configuration loaded from".

Configuring Terracotta Clusters For High Availability

Introduction

High Availability (HA) is an implementation designed to maintain uptime and access to services even during component overloads and failures. Terracotta clusters offer simple and scalable HA implementations based on the Terracotta Server Array (see Terracotta Server Array Architecture for more information).

The main features of a Terracotta HA architecture include:

- Instant failover using a hot standby or multiple active servers – provides continuous uptime and services
- Configurable automatic internode monitoring – Terracotta HealthChecker
- Automatic permanent storage of all current shared (in-memory) data – available to all server instances (no loss of application state)
- Automatic reconnection of temporarily disconnected server instances and clients – restores hot standbys without operator intervention, allows "lost" clients to reconnect

  Client reconnection refers to reconnecting clients that have not yet been disconnected from the cluster by the Terracotta Server Array. To learn about reconnecting BigMemory Max clients that have been disconnected from their cluster, see Using Rejoin to Automatically Reconnect Terracotta Clients.

TIP: Nomenclature

This document may refer to a Terracotta server instance as L2, and a Terracotta client (the node running your application) as L1. These are the shorthand references used in Terracotta configuration files.
It is important to thoroughly test any High Availability setup before going to production. Suggestions for testing High Availability configurations are provided in this section.

Basic High-Availability Configuration

A basic high-availability configuration has the following components:

- Two or More Terracotta Server Instances

  You may set up High Availability using either `<server>` or `<mirror-group>` configurations. Note that `<server>` instances do work together as a mirror group, but to create more than one stripe, or to configure the election-time, use `<mirror-group>` instances. See Terracotta Server Arrays Architecture on how to set up a cluster with multiple Terracotta server instances.
- Server-Server Reconnection

  A reconnection mechanism can be enabled to restore lost connections between active and mirror Terracotta server instances. See Automatic Server Instance Reconnect for more information.
- Server-Client Reconnection

  A reconnection mechanism can be enabled to restore lost connections between Terracotta clients and server instances. See Automatic Client Reconnect for more information.

High-Availability Features

The following high-availability features can be used to extend the reliability of a Terracotta cluster. These features are controlled using properties set with the <tc-properties> section at the *beginning* of the Terracotta configuration file:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-8.xsd">

 <tc-properties>
   <property name="some.property.name" value="true"/>
   <property name="some.other.property.name" value="true"/>
   <property name="still.another.property.name" value="1024"/>
 </tc-properties>

<!-- The rest of the Terracotta configuration goes here. -->

</tc:tc-config>
```

See the section *Overriding tc.properties* in Configuration Guide and Reference for more information.

HealthChecker

HealthChecker is a connection monitor similar to TCP keep-alive. HealthChecker functions between Terracotta server instances (in High Availability environments), and between Terracotta sever instances and clients. Using HealthChecker, Terracotta nodes can determine if peer nodes are reachable, up, or in a GC operation. If a peer node is unreachable or down, a Terracotta node using HealthChecker can take corrective action. HealthChecker is on by default.

You configure HealthChecker using certain Terracotta properties, which are grouped into three different categories:

- Terracotta server instance -> Terracotta client
- Terracotta Server -> Terracotta Server (HA setup only)
- Terracotta Client -> Terracotta Server

Property category is indicated by the prefix:

- l2.healthcheck.l1 indicates L2 -> L1
- l2.healthcheck.l2 indicates L2 -> L2
- l1.healthcheck.l2 indicates L1 -> L2

For example, the `l2.healthcheck.l2.ping.enabled` property applies to L2 -> L2.

The following HealthChecker properties can be set in the <tc-properties> section of the Terracotta configuration file:

Property	Definition

`l2.healthcheck.l1.ping.enabled`
`l2.healthcheck.l2.ping.enabled`
`l1.healthcheck.l2.ping.enabled`
	Enables (True) or disables (False) ping probes (tests). Ping probes are high-level attempts to gauge the ability of a remote

	node to respond to requests and is useful for determining if temporary inactivity or problems are responsible for the node's silence. Ping probes may fail due to long GC cycles on the remote node.

`l2.healthcheck.l1.ping.idletime`
`l2.healthcheck.l2.ping.idletime`
`l1.healthcheck.l2.ping.idletime`
	The maximum time (in milliseconds) that a node can be silent (have no network traffic) before HealthChecker begins a ping probe to determine if the node is alive.

`l2.healthcheck.l1.ping.interval`
`l2.healthcheck.l2.ping.interval`
`l1.healthcheck.l2.ping.interval`
	If no response is received to a ping probe, the time (in milliseconds) that HealthChecker waits between retries.

`l2.healthcheck.l1.ping.probes`
`l2.healthcheck.l2.ping.probes`
`l1.healthcheck.l2.ping.probes`
	If no response is received to a ping probe, the maximum number (integer) of retries HealthChecker can attempt.

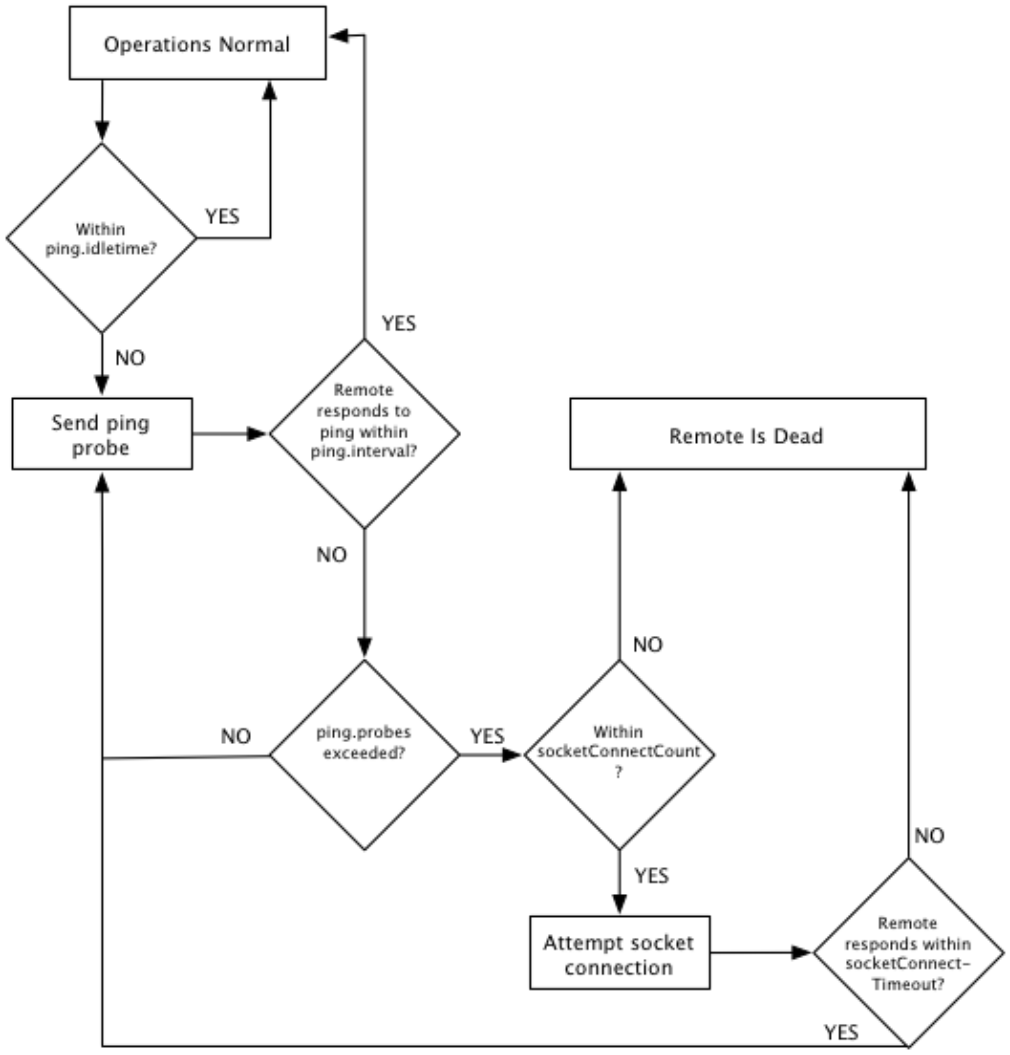
`l2.healthcheck.l1.socketConnect`
`l2.healthcheck.l2.socketConnect`
`l1.healthcheck.l2.socketConnect`
	Enables (True) or disables (False) socket-connection tests. This is a low-level connection that determines if the remote node is reachable and can access the network. Socket connections are not affected by GC cycles.

`l2.healthcheck.l1.socketConnectTimeout`
`l2.healthcheck.l2.socketConnectTimeout`
`l1.healthcheck.l2.socketConnectTimeout`
	A multiplier (integer) to determine the maximum amount of time that a remote node has to respond before HealthChecker concludes that the node is dead (regardless of previous successful socket connections). The time is determined by multiplying the value in ping.interval by this value.

`l2.healthcheck.l1.socketConnectCount`
`l2.healthcheck.l2.socketConnectCount`
`l1.healthcheck.l2.socketConnectCount`
	The maximum number (integer) of successful socket connections that can be made without a successful ping probe. If this limit is exceeded, HealthChecker concludes that the target node is dead.

`l1.healthcheck.l2.bindAddress`	Binds the client to the configured IP address. This is useful where a host has more than one IP address available for a client to use. The default value of "0.0.0.0" allows the system to assign an IP address.
`l1.healthcheck.l2.bindPort`	Set the client's callback port. Terracotta configuration does not assign clients a port for listening to cluster communications such as that required by HealthChecker. The default value of "0" allows the system to assign a port. A value of "-1" disables a client's callback port.

The following diagram illustrates how HealthChecker functions.



Calculating HealthChecker Maximum

The following formula can help you compute the maximum time it will take HealthChecker to discover failed or disconnected remote nodes:

```
Max Time = (ping.idletime) + socketConnectCount * [(ping.interval * ping.probes)
           + (socketConnectTimeout * ping.interval)]
```

Note the following about the formula:

- The response time to a socket-connection attempt is less than or equal to `(socketConnectTimeout * ping.interval)`. For calculating the worst-case scenario (absolute maximum time), the equality is used. In most real-world situations the socket-connect response time is likely to be close to 0 and the formula can be simplified to the following:

  ```
  Max Time = (ping.idletime) + [socketConnectCount * (ping.interval * ping.probes)]
  ```
- `ping.idletime`, the trigger for the full HealthChecker process, is counted once since it is in effect only once each time the process is triggered.

- `socketConnectCount` is a multiplier that is in incremented as long as a positive response is received for each socket connection attempt.
- The formula yields an ideal value, since slight variations in actual times can occur.
- To prevent clients from disconnecting too quickly in a situation where an active server is temporarily disconnected from both the backup server and those clients, ensure that the Max Time for L1->L2 is approximately 8-12 seconds longer than for L2->L2. If the values are too close together, then in certain situations the active server may kill the backup and refuse to allow clients to reconnect.

Configuration Examples

The configuration examples in this section show settings for L1 -> L2 HealthChecker. However, they apply in the similarly to L2 -> L2 and L2 -> L1, which means that the server is using HealthChecker on the client.

Aggressive

The following settings create an aggressive HealthChecker with low tolerance for short network outages or long GC cycles:

```
<property name="l1.healthcheck.l2.ping.enabled" value="true" />
<property name="l1.healthcheck.l2.ping.idletime" value="2000" />
<property name="l1.healthcheck.l2.ping.interval" value="1000" />
<property name="l1.healthcheck.l2.ping.probes" value="3" />
<property name="l1.healthcheck.l2.socketConnect" value="true" />
<property name="l1.healthcheck.l2.socketConnectTimeout" value="2" />
<property name="l1.healthcheck.l2.socketConnectCount" value="5" />
```

According to the HealthChecker "Max Time" formula, the maximum time before a remote node is considered to be lost is computed in the following way:

```
2000 + 5 [( 3 * 1000 ) + ( 2 * 1000)] = 27000
```

In this case, after the initial idletime of 2 seconds, the remote failed to respond to ping probes but responded to every socket connection attempt, indicating that the node is reachable but not functional (within the allowed time frame) or in a long GC cycle. This aggressive HealthChecker configuration declares a node dead in no more than 27 seconds.

If at some point the node had been completely unreachable (a socket connection attempt failed), HealthChecker would have declared it dead sooner. Where, for example, the problem is a disconnected network cable, the "Max Time" is likely to be even shorter:

```
2000 + 1[3 * 1000) + ( 2 * 1000 ) = 7000
```

In this case, HealthChecker declares a node dead in no more than 7 seconds.

Tolerant

The following settings create a HealthChecker with a higher tolerance for interruptions in network communications and long GC cycles:

```
<property name="l1.healthcheck.l2.ping.enabled" value="true" />
<property name="l1.healthcheck.l2.ping.idletime" value="5000" />
<property name="l1.healthcheck.l2.ping.interval" value="1000" />
<property name="l1.healthcheck.l2.ping.probes" value="3" />
<property name="l1.healthcheck.l2.socketConnect" value="true" />
```

```
<property name="l1.healthcheck.l2.socketConnectTimeout" value="5" />
<property name="l1.healthcheck.l2.socketConnectCount" value="10" />
```

According to the HealthChecker "Max Time" formula, the maximum time before a remote node is considered to be lost is computed in the following way:

```
5000 + 10 [( 3 x 1000 ) + ( 5 x 1000)] = 85000
```

In this case, after the initial idletime of 5 seconds, the remote failed to respond to ping probes but responded to every socket connection attempt, indicating that the node is reachable but not functional (within the allowed time frame) or excessively long GC cycle. This tolerant HealthChecker configuration declares a node dead in no more than 85 seconds.

If at some point the node had been completely unreachable (a socket connection attempt failed), HealthChecker would have declared it dead sooner. Where, for example, the problem is a disconnected network cable, the "Max Time" is likely to be even shorter:

```
5000 + 1[3 * 1000) + ( 5 * 1000 )] = 13000
```

In this case, HealthChecker declares a node dead in no more than 13 seconds.

Tuning HealthChecker to Allow for GC or Network Interruptions

GC cycles do not affect a node's ability to respond to socket-connection requests, while network interruptions do. This difference can be used to tune HealthChecker to work more efficiently in a cluster where one or the other of these issues is more likely to occur:

- To favor detection of network interruptions, adjust the socketConnectCount down (since socket connections will fail). This will allow HealthChecker to disconnect a client sooner due to network issues.
- To favor detection of GC pauses, adjust the socketConnectCount up (since socket connections will succeed). This will allow clients to remain in the cluster longer when no network disconnection has occurred.

The ping interval increases the time before socket-connection attempts kick in to verify health of a remote node. The ping interval can be adjusted up or down to add more or less tolerance in either of the situations listed above.

Automatic Server Instance Reconnect

An automatic reconnect mechanism can prevent short network disruptions from forcing a restart for any Terracotta server instances in a server array with hot standbys. If not disabled, this mechanism is by default in effect in clusters set to networked-based HA mode.

NOTE: Increased Time-to-Failover

This feature increases time-to-failover by the timeout value set for the automatic reconnect mechanism. This event-based reconnection mechanism works independently and exclusively of HealthChecker. If HealthChecker has already been triggered, this mechanism cannot be triggered for the same node. If this mechanism is triggered first by an internal Terracotta event, HealthChecker is prevented from being triggered for the same node. The events that can trigger this mechanism are not exposed by API but are logged.

Configure the following properties for the reconnect mechanism:

- `l2.nha.tcgroupcomm.reconnect.enabled` – (DEFAULT: false) When set to "true" enables a server instance to attempt reconnection with its peer server instance after a disconnection is detected. Most use cases should benefit from enabling this setting.
- `l2.nha.tcgroupcomm.reconnect.timeout` – Enabled if `l2.nha.tcgroupcomm.reconnect.enabled` is set to true. Specifies the timeout (in milliseconds) for reconnection. Default: 2000. This parameter can be tuned to handle longer network disruptions.

Automatic Client Reconnect

Clients disconnected from a Terracotta cluster normally require a restart to reconnect to the cluster. An automatic reconnect mechanism can prevent short network disruptions from forcing a restart for Terracotta clients disconnected from a Terracotta cluster.

NOTE: Performance Impact of Using Automatic Client Reconnect

With this feature, clients waiting to reconnect continue to hold locks. Some application threads may block while waiting to for the client to reconnect.
This event-based reconnection mechanism works independently and exclusively of HealthChecker. If HealthChecker has already been triggered, this mechanism cannot be triggered for the same node. If this mechanism is triggered first by an internal Terracotta event, HealthChecker is prevented from being triggered for the same node. The events that can trigger this mechanism are not exposed by API but are logged.

Configure the following properties for the reconnect mechanism:

- `l2.l1reconnect.enabled` – (DEFAULT: false) When set to "true" enables a client to reconnect to a cluster after a disconnection is detected. This property controls a server instance's reaction to such an attempt. It is set on the server instance and is passed to clients by the server instance. A client cannot override the server instance's setting. If a mismatch exists between the client setting and a server instance's setting, and the client attempts to reconnect to the cluster, the client emits a mismatch error and exits. Most use cases should benefit from enabling this setting.
- `l2.l1reconnect.timeout.millis` – Enabled if `l2.l1reconnect.enabled` is set to true. Specifies the timeout (in milliseconds) for reconnection. This property controls a server instance's timeout during such an attempt. It is set on the server instance and is passed to clients by the server instance. A client cannot override the server instance's setting. Default: 2000. This parameter can be tuned to handle longer network disruptions.

Special Client Connection Properties

Client connections can also be tuned for the following special cases:

- Client failover after server failure
- First-time client connection

The connection properties associated with these cases are already optimized for most typical environments. If you attempt to tune these properties, be sure to thoroughly test the new settings.

Client Failover After Server Failure

When an active Terracotta server instance fails, and a mirror Terracotta server is available, the mirror server becomes active. Terracotta clients connected to the previous active server automatically switch to the new active server. However, these clients have a limited window of time to complete the failover.

TIP: Clusters with a Single Server

This reconnection window also applies in a cluster with a single Terracotta server that is restarted. However, a single-server cluster must have <restartable> enabled for the reconnection window to take effect.
This window is configured in the Terracotta configuration file using the <client-reconnect-window> element:

```
<servers>
  ...
  <client-reconnect-window>120</client-reconnect-window>
  <!-- The reconnect window is configured in seconds, with a default value of 120.
       The default value is "built in," so the element does not have to be explicitly
       added unless a different value is required. -->
  ...
</servers>
```

Clients that fail to connect to the new active server must be restarted if they are to successfully reconnect to the cluster.

First-Time Client Connection

When a Terracotta client is first started (or restarted), it attempts to connect to a Terracotta server instance based on the following properties:

```
# -1 == retry all configured servers eternally.
l1.max.connect.retries = -1
# Must the client and server be running the same version of Terracotta?
l1.connect.versionMatchCheck.enabled = true
# Time (in milliseconds) before a socket connection attempt is timed out.
l1.socket.connect.timeout=10000
# Time (in milliseconds; minimum 10) between attempts to connect to a server.
l1.socket.reconnect.waitInterval=1000
```

A client with `l1.max.connect.retries` set to a positive integer is given a limited number of attempts (equal to that integer) to connect. If the client fails to connect after the configured number of attempts, it exits.

The property `l1.max.connect.retries` controls connection attempts only *after* the client has obtained and resolved its configuration from the server. To control connection attempts *before* configuration is resolved, set the following property on the client:

```
-Dcom.tc.tc.config.total.timeout=5000
```

This property limits the time (in milliseconds) that a client spends attempting to make an initial connection.

Using Rejoin to Automatically Reconnect Terracotta Clients

A Terracotta client may disconnect and be timed out (ejected) from the cluster. Typically, this occurs because of network communication interruptions lasting longer than the configured HA settings for the cluster. Other

causes include long GC pauses and slowdowns introduced by other processes running on the client hardware.

You can configure clients to automatically rejoin a cluster after they are ejected. If the ejected client continues to run under nonstop cache settings, and then senses that it has reconnected to the cluster (receives a clusterOnline event), it can begin the rejoin process.

Note the following about using the rejoin feature:

- Disconnected clients can only rejoin clusters to which they were previously connected.
- Clients rejoin as new members and will wipe all cached data to ensure that no pauses or inconsistencies are introduced into the cluster.
- Clients cannot rejoin a new cluster; if the TSA has been restarted and its data has not been persisted, clients can never rejoin and must be restarted.
- If the TSA has been restarted and its data has been persisted, clients are allowed to rejoin.
- Any nonstop-related operations that begin (and do not complete) before the rejoin operation completes may be unsuccessful and may generate a NonStopCacheException.
- If a Terracotta client with rejoin enabled is running in a JVM with clients that do not have rejoin, then only that client will rejoin after a disconnection. The remaining clients cannot rejoin and may cause the application to behave unpredictably.
- Once a client rejoins, the clusterRejoined event is fired on that client only.

Configuring Rejoin

The rejoin feature is disabled by default. To enable the rejoin feature in an Terracotta client, follow these steps:

1. Ensure that all of the caches in the Ehcache configuration file where rejoin is enabled have nonstop enabled.
2. Ensure that your application does not create caches on the client without nonstop enabled.
3. Enable the rejoin attribute in the client's `<terracottaConfig>` element:

```
<terracottaConfig url="myHost:9510" rejoin="true" />
```

For more options on configuring `<terracottaConfig>`, see the configuration reference.

Exception During Rejoin

Under certain circumstances, if a lock is being used by your application, an InvalidLockAfterRejoinException could be thrown during or after client rejoin. This exception occurs when an unlock operation takes place on a lock obtained *before* the rejoin attempt completed.

To ensure that locks are released properly, application code should encapsulate lock-unlock operations with try-finally blocks:

```
myLock.acquireLock();
try {
  // Do some work.
} finally {
  myLock.unlock();
}
```

Effective Client-Server Reconnection Settings: An Example

To prevent unwanted disconnections, it is important to understand the potentially complex interaction between HA settings and the environment in which your cluster runs. Settings that are not appropriate for a particular environment can lead to unwanted disconnections under certain circumstances.

In general, it is advisable to maintain an L1-L2 HealthChecker timeout that falls between the L2-L2 HealthChecker timeout as modified in the following ineqaulity:

```
L2-L2 HealthCheck + Election Time
        <  L1-L2 HealthCheck
                  <  L2-L2 HealthCheck + Election Time + Client Reconnect Window
```

This allows a cluster's L1s to avoid disconnecting before a client reconnection window is opened (a backup L2 takes over), or to not disconnect if that window is never opened (the original active L2 is still functional). The Election Time and Client Reconnect Window settings, which are found in the Terracotta configuration file, are respectively 5 seconds and 120 seconds by default.

For example, in a cluster where the L2-L2 HealthChecker triggers at 55 seconds, a backup L2 can take over the cluster after 180 seconds (55 + 5 + 120). If the L1-L2 HealthChecker triggers after a time that is greater than 180 seconds, clients may not attempt to reconnect until the reconnect window is closed and it's too late.

If the L1-L2 HealthChecker triggers after a time that is less than 60 seconds (L2-L2 HealthChecker + Election Time), then the clients may disconnect from the active L2 before its failure is determined. Should the active L2 win the election, the disconnected L1s would then be lost.

A check is performed at server startup to ensure that L1-L2 HealthChecker settings are within the effective range. If not, a warning with a prescription is printed.

Testing High-Availability Deployments

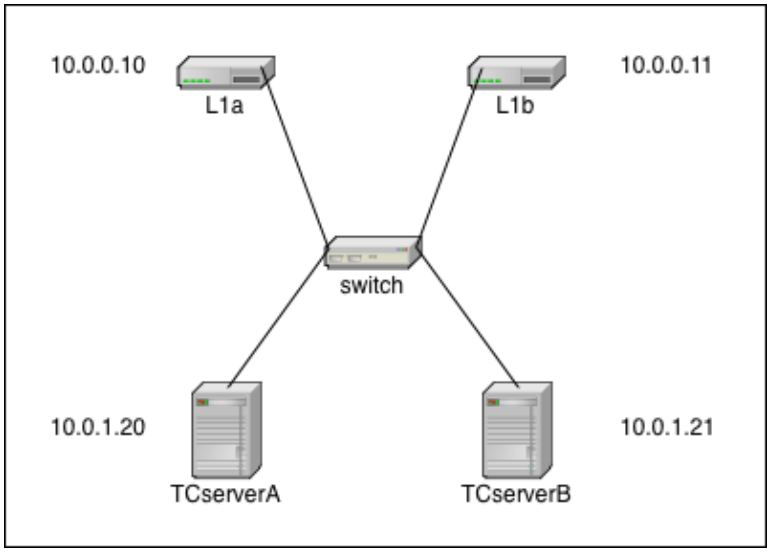
This section presents recommendations for designing and testing a robust cluster architecture. While these recommendations have been tested and shown to be effective under certain circumstances, in your custom environment additional testing is still necessary to ensure an optimal setup, and to meet specific demands related to factors such as load.

High-Availability Network Architecture And Testing

To take advantage of the Terracotta active-mirror server configuration, certain network configurations are necessary to prevent split-brain scenarios and ensure that Terracotta clients (L1s) and server instances (L2s) behave in a deterministic manner after a failure occurs. This is regardless of the nature of the failure, whether network, machine, or other type.

This section outlines two possible network configurations that are known to work with Terracotta failover. While it is possible for other network configurations to work reliably, the configurations listed in this document have been well tested and are fully supported.

Deployment Configuration: Simple (no network redundancy)



Description

This is the simplest network configuration. There is no network redundancy so when any failure occurs, there is a good chance that all or part of the cluster will stop functioning. All failover activity is up to the Terracotta software.

In this diagram, the IP addresses are merely examples to demonstrate that the L1s (L1a & L1b) and L2s (TCserverA & TCserverB) can live on different subnets. The actual addressing scheme is specific to your environment. The single switch is a single point of failure.

Additional configuration

There is no additional network or operating-system configuration necessary in this configuration. Each machine needs a proper network configuration (IP address, subnet mask, gateway, DNS, NTP, hostname) and must be plugged into the network.

Test Plan - Network Failures Non-Redundant Network

To determine that your configuration is correct, use the following tests to confirm all failure scenarios behave as expected.

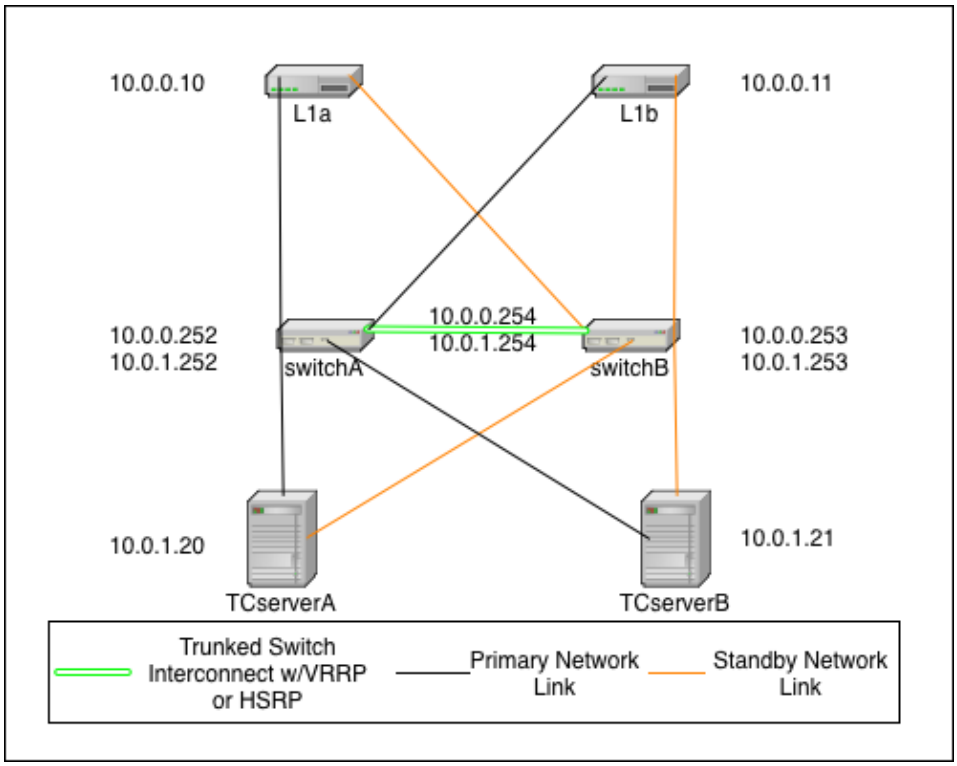
TestID	Failure	Expected Outcome
FS1	Loss of L1a (link or system)	Cluster continues as normal using only L1b
FS2	Loss of L1b (link or system)	Cluster continues as normal using only L1a
FS3	Loss of L1a & L1b	Non-functioning cluster
FS4	Loss of Switch	Non-functioning cluster
FS5	Loss of Active L2 (link or system)	mirror L2 becomes new Active L2, L1s fail over to new Active L2
FS6	Loss of mirror L2	Cluster continues as normal without TC redundancy
FS7	Loss of TCservers A & B	Non-functioning cluster

Test Plan - Network Tests Non-redundant Network

After the network has been configured, you can test your configuration with simple ping tests.

TestID	Host	Action	Expected Outcome
NT1	all	ping every other host	successful ping
NT2	all	pull network cable during continuous ping	ping failure until link restored
NT3	switch	reload	all pings cease until reload complete and links restored

Deployment Configuration: Fully Redundant



Description

This is the fully redundant network configuration. It relies on the failover capabilities of Terracotta, the switches, and the operating system. In this scenario it is even possible to sustain certain double failures and still maintain a fully functioning cluster.

In this diagram, the IP addressing scheme is merely to demonstrate that the L1s (L1a & L1b) can be on a different subnet than the L2s (TCserverA & TCserverB). The actual addressing scheme will be specific to your environment. If you choose to implement with a single subnet, then there will be no need for VRRP/HSRP but you will still need to configure a single VLAN (can be VLAN 1) for all TC cluster machines.

In this diagram, there are two switches that are connected with trunked links for redundancy and which implement Virtual Router Redundancy Protocol (VRRP) or HSRP to provide redundant network paths to the cluster servers in the event of a switch failure. Additionally, all servers are configured with both a primary and

secondary network link which is controlled by the operating system. In the event of a NIC or link failure on any single link, the operating system should fail over to the backup link without disturbing (e.g. restarting) the Java processes (L1 or L2) on the systems.

The Terracotta fail over is identical to that in the simple case above, however both NIC cards on a single host would need to fail in this scenario before the TC software initiates any fail over of its own.

Additional configuration

- Switch - Switches need to implement VRRP or HSRP to provide redundant gateways for each subnet. Switches also need to have a trunked connection of two or more lines in order to prevent any single link failure from splitting the virtual router in two.
- Operating System - Hosts need to be configured with bonded network interfaces connected to the two different switches. For Linux, choose mode 1. More information about Linux channel bonding can be found in the RedHat Linux Reference Guide. Pay special attention to the amount of time it takes for your VRRP or HSRP implementation to reconverge after a recovery. You don't want your NICs to change to a switch that is not ready to pass traffic. This should be tunable in your bonding configuration.

Test Plan - Network Failures Redundant Network

The following tests continue the tests listed in Network Failures (Pt. 1). Use these tests to confirm that your network is configured properly.

TestID	Failure	Expected Outcome
FS8	Loss of any primary network link	Failover to standby link
FS9	Loss of all primary links	All nodes fail to their secondary link
FS10	Loss of any switch	Remaining switch assumes VRRP address and switches fail over NICs if necessary
FS11	Loss of any L1 (both links or system)	Cluster continues as normal using only other L1
FS12	Loss of Active L2	mirror L2 becomes the new Active L2, All L1s fail over to the new Active L2
FS13	Loss of mirror L2	Cluster continues as normal without TC redundancy
FS14	Loss of both switches	non-functioning cluster
FS15	Loss of single link in switch trunk	Cluster continues as normal without trunk redundancy
FS16	Loss of both trunk links	possible non-functioning cluster depending on VRRP or HSRP implementation
FS17	Loss of both L1s	non-functioning cluster
FS18	Loss of both L2s	non-functioning cluster

Test Plan - Network Testing Redundant Network

After the network has been configured, you can test your configuration with simple ping tests and various failure scenarios.

The test plan for Network Testing consists of the following tests:

TestID	Host	Action	Expected Outcome
NT4	any	ping every other host	successful ping
NT5	any	pull primary link during continuous ping to any other host	failover to secondary link, no noticable network interruption
NT6	any	pull standby link during continuous ping to any other host	no effect
NT7	Active L2	pull both network links	mirror L2 becomes Active, L1s fail over to new Active L2
NT8	Mirror L2	pull both network links	no effect
NT9	switchA	reload	nodes detect link down and fail to standby link, brief network outage if VRRP transition occurs
NT10	switchB	reload	brief network outage if VRRP transition occurs
NT11	switch	pull single trunk link	no effect

Terracotta Cluster Tests

All tests in this section should be run after the Network Tests succeed.

Test Plan - Active L2 System Loss Tests - verify Mirror Takeover

The test plan for mirror takeover consists of the following tests:

TestID	Test	Setup	Steps	Expected Result
TAL1	Active L2 Loss - Kill	L2-A is active, L2-B is mirror. All systems are running and available to take traffic.	1. Run app 2. Kill -9 Terracotta PID on L2-A (Active)	L2-B(mirror) becomes active. Takes the load. No drop in TPS on Failover.
TAL2	Active L2 Loss - clean shutdown	L2-A is active, L2-B is mirror. All systems are running and available to take traffic.	1. Run app 2.Run ~/bin/stop-tc-server.sh on L2-A (Active)	L2-B(mirror) becomes active. Takes the load. No drop in TPS on Failover.
TAL3	Active L2 Loss - Power Down	L2-A is Active, L2-B is mirror. All systems are running and available to take traffic	1. Run app 2. Power down L2-A (Active)	L2-B(mirror) becomes active. Takes the load. No drop in TPS on Failover.
TAL4	Active L2 Loss - Reboot	L2-A is Active, L2-B is mirror. All systems are running and available to take traffic	1. Run app 2. Reboot L2-A (Active)	L2-B(mirror) becomes active. Takes the load. No drop in TPS on Failover.
TAL5	Active L2 Loss - Pull Plug	L2-A is Active, L2-B is mirror. All systems are running and available to take traffic	1. Run app 2. Pull the power cable on L2-A (Active)	L2-B(mirror) becomes active. Takes the load. No drop in TPS on Failover.

Test Plan - Mirror L2 System Loss Tests

System loss tests confirms High Availability in the event of loss of a single system. This section outlines tests for testing failure of the Terracotta mirror server.

The test plan for testing Terracotta mirror Failures consist of the following tests:

TestID	Test	Setup	Steps	Expected Result
TPL1	Mirror L2 loss - kill	L2-A is active, L2-B is mirror. All systems are running and available to take traffic.	1. Run app 2. Kill -9 L2-B (mirror)	data directory needs to be cleaned up, then when L2-B is restarted, it re-synchs state from Active Server.
TPL2	Mirror L2 loss -clean	L2-A is active, L2-B is mirror. All systems are running and available to take traffic	1. Run app 2. Run ~/bin/stop-tc-server.sh on L2-B (mirror)	data directory needs to be cleaned up, then when L2-B is restarted, it re-synchs state from Active Server.
TPL3	Mirror L2 loss -power down	L2-A is active, L2-B is mirror. All systems are running and available to take traffic	1. Run app 2. Power down L2-B (mirror)	data directory needs to be cleaned up, then when L2-B is restarted, it re-synchs state from Active Server.
TPL4	Mirror L2 loss -reboot	L2-A is active, L2-B is mirror. All systems are running and available to take traffic	1. Run app 2. Reboot L2-B (mirror)	data directory needs to be cleaned up, then when L2-B is restarted, it re-synchs state from Active Server.
TPL5	Mirror L2 loss -Pull Plug	L2-A is active, L2-B is mirror. All systems are running and available to take traffic	1. Run app 2. Pull plug on L2-B (mirror)	data directory needs to be cleaned up, then when L2-B is restarted, it re-synchs state from Active Server.

Test Plan - Failover/Failback Tests

This section outlines tests to confirm the cluster ability to fail-over to the mirror Terracotta server, and fail back.

The test plan for testing fail over and fail back consists of the following tests:

TestID	Test	Setup	Steps	Expected Result
TFO1	Failover/Failback	L2-A is active, L2-B is mirror. All systems are running and available to take traffic	1. Run application 2. Kill -9 (or run stop-tc-server) on L2-A (Active) 3. After L2-B takes over as Active, start-tc-server on L2-A. (L2-A is now mirror) 4. Kill -9 (or run stop-tc-server) on L2-B. (L2-A is now Active)	After first failover L2-A->L2-B, txns should continue. L2-A should come up cleanly in mirror mode when tc-server is run. When second failover occurs L2-B->L2-A, L2-A should process txns.

Test Plan - Loss of Switch Tests

{tip} This test can only be run on a redundant network {tip}

This section outlines testing the loss of a switch in a redundant network, and confirming that no interrupt of service occurs.

The test plan for testing failure of a single switch consists of the following tests:

TestID	Test	Setup	Steps	Expected Result

TSL1	Loss of 1 Switch	2 Switches in redundant configuration. L2-A is active, L2-B is mirror. All systems are running and available to take traffic.	1. Run application 2. Power down/pull plug on Switch	All traffic transparently moves to switch 2 with no interruptions

Test Plan - Loss of Network Connectivity

This section outlines testing the loss of network connectivity.

The test plan for testing failure of the network consists of the following tests:

TestID	Test	Setup	Steps	Expected Result
TNL1	Loss of NIC wiring (Active)	L2-A is active, L2-B is mirror. All systems are runnng and available to traffic	1. Run application 2. Remove Network Cable on L2-A	All traffic transparently moves to L2-B with no interruptions
TNL2	Loss of NIC wiring (mirror)	L2-A is active, L2-B is mirror. All systems are runnng and available to traffic	1. Run application 2. Remove Network Cable on L2-B	No user impact on cluster

Test Plan - Terracotta Cluster Failure

This section outlines the tests to confirm successful continued operations in the face Terracotta Cluster failures.

The test plan for testing Terracotta Cluster failures consists of the following tests:

TestID	Test	Setup	Steps	Expected Result
TF1	Process Failure Recovery	L2-A is active, L2-B is mirror. All systems are runnng and available to traffic	1. Run application 2. Bring down all L1s and L2s 3. Start L2s then L1s	Cluster should come up and begin taking txns again
TF2	Server Failure Recovery	L2-A is active, L2-B is mirror. All systems are runnng and available to traffic	1. Run application 2. Power down all machines 3. Start L2s and then L1s	Should be able to run application once all servers are up.

Client Failure Tests

This section outlines tests to confirm successful continued operations in the face of Terracotta client failures.

The test plan for testing Terracotta Client failures consists of the following tests:

TestID	Test	Setup	Steps	Expected Result
TCF1	L1 Failure -	L2-A is active, L2-B is mirror. 2 L1s L1-A and L1-B All systems are running and available to traffic	1. Run application 2. kill -9 L1-A.	L1-B should take all incoming traffic. Some timeouts may occur due to txns in process when L1 fails over.

Cluster Security

Introduction

The Enterprise Edition of the Terracotta kit provides standard authentication methods to control access to Terracotta servers. Enabling one of these methods causes a Terracotta server to require credentials before allowing a JMX connection to proceed.

You can choose one of the following to secure servers:

- SSL-based security – Authenticates all nodes (including clients) and secures the entire cluster with encrypted connections. Includes role-based authorization.
- LDAP-based authentication – Uses your organization's authentication database to secure access to Terracotta servers.
- JMX-based authentication – provides a simple authentication scheme to protect access to Terracotta servers.

Note that Terracotta scripts cannot be used with secured servers without passing credentials to the script.

Configure SSL-based Security

See the advanced security page to learn how to use Secure Sockets Layer (SSL) encryption and certificate-based authentication to secure enterprise versions of Terracotta clusters. Note that using SSL to a Terracotta cluster reduces performance due to the overhead introduced by encrypting inter-node communication.

Configure Security Using LDAP (via JAAS)

Lightweight Directory Access Protocol (LDAP) security is based on JAAS and requires Java 1.6. Using an earlier version of Java will not prevent Terracotta servers from running; however security will *not* be enabled.

To configure security using LDAP, follow these steps:

1. Save the following configuration to the file `.java.login.config`:

```
Terracotta {
com.sun.security.auth.module.LdapLoginModule REQUIRED
java.naming.security.authentication="simple"
userProvider="ldap://orgstage:389"
authIdentity="uid={USERNAME},ou=People,dc=terracotta,dc=org"
authzIdentity=controlRole
useSSL=false
bindDn="cn=Manager"
bindCredential="****"
bindAuthenticationType="simple"
debug=true;
};
```

   Edit the values for `userProvider` (LDAP server), `authIdentity` (user identity), and `bindCredential` (encrypted password) to match the values for your environment.

2. Save the file `.java.login.config` to the directory named in the Java property `user.home`.
3. Add the following configuration to each <server> block in the Terracotta configuration file:

```
<server host="myHost" name="myServer">
...
 <authentication>
   <mode>
     <login-config-name>Terracotta</login-config-name>
   </mode>
 </authentication>
...
</server>
```

4. Start the Terracotta server and look for a log message containing "INFO - Credentials: loginConfig[Terracotta]" to confirm that LDAP security is in effect.

   NOTE: Incorrect Setup

   If security is set up incorrectly, the Terracotta server can still be started. However, you may not be able to shut down the server using the shutdown script (**stop-tc-server**).

Configure Security Using JMX Authentication

Terracotta can use the standard Java security mechanisms for JMX authentication, which relies on the creation of `.access` and `.password` files with correct permissions set. The default location for these files for JDK 1.5 or higher is `$JAVA_HOME/jre/lib/management`.

To configure security using JMX authentication, follow these steps:

1. Ensure that the desired usernames and passwords for securing the target servers are in the JMX password file `jmxremote.password` and that the desired roles are in the JMX access file `jmxremote.access`.
2. If both `jmxremote.access` and `jmxremote.password` are in the default location (`$JAVA_HOME/jre/lib/management`), add the following configuration to each <server> block in the Terracotta configuration file:

```
<server host="myHost" name="myServer">
...
 <authentication />
...
</server>
```

3. If `jmxremote.password` is not in the default location, add the following configuration to each <server> block in the Terracotta configuration file:

```
<server host="myHost" name="myServer">
...
 <authentication>
   <mode>
     <password-file>/path/to/jmx.password</password-file>
   </mode>
 </authentication>
...
</server>
```

4. If `jmxremote.access` is not in the default location, add the following configuration to each <server> block in the Terracotta configuration file:

```
<server host="myHost" name="myServer">
...
 <authentication>
   <mode>
     <password-file>/path/to/jmxremote.password</password-file>
   </mode>
   <access-file>/path/to/jmxremote.access</access-file>
 </authentication>
...
</server>
```

File Not Found Error

If the JMX password file is not found when the server starts up, an error is logged stating that the password file does not exist.

User Roles

There are two roles available for Terracotta servers and clients:

- admin â— The user with the "admin" role is the initial user who sets up security. Thereafter, the "admin" user can perform system functions such as shutting down servers, clearing or deleting caches and cache managers, and reloading configurations.
- terracotta â— This is the operator role. The default username for the operator role is "terracotta". The "terracotta" user can connect to the TMC and access the read-only areas. In addition, the "terracotta" user can start a secure server. But a user must have the "admin" role in order to run the stop-tc-server script.

Using Scripts Against a Server with Authentication

A script that targets a secured Terracotta server must use the correct login credentials to access the server. If you run a Terracotta script such as **backup-data** or **server-stat** against a secured server, pass the credentials using the `-u` (followed by username) and `-w` (followed by password) flags.

For example, if Server1 is secured with username "user1" and password "password", you run the **server-stat** script by entering the following:

UNIX/LINUX

```
[PROMPT]${TERRACOTTA_HOME}/server/bin/server-stat.sh -s Server1 -u user1 -w password
```

MICROSOFT WINDOWS

```
[PROMPT]%TERRACOTTA_HOME%\server\bin\server-stat.bat -s Server1 -u user1 -w password
```

Extending Server Security

Since JMX messages are not encrypted, server authentication does not provide secure message transmission once valid credentials are provided by a listening client. To extend security beyond the login threshold, consider the following options:

- Place Terracotta servers in a secure location on a private network.

- Restrict remote queries to an encrypted tunnel such as provided by SSH or stunnel.
- If using public or outside networks, use a VPN for all communication in the cluster.
- If using Ehcache, add a cache decorator to the cache that implements your own encryption and decryption.

Securing Terracotta Clusters

Introduction

Terracotta clusters can be secured using authentication, authorization, and encryption. Built-in authentication and authorization is available, as is support for external directory services. In addition, certificate-based Secure Sockets Layer (SSL) encryption can be used to secure communications between nodes.

Security in a Terracotta cluster includes both server-server connections and client-server connections. Note that all connections in a secured cluster must be secure; no insecure connections are allowed, and therefore security must be enabled globally in the cluster, including in the Terracotta Management Server. It is not possible to enable security for certain nodes only.

Overview

Security is set up using Terracotta configuration, tools provided in the Terracotta kit, standard Java tools, and public key infrastructure (via standard digital X.509 digital certificates).

Security-Related Files

Each Terracotta server utilizes the following types of files to implement security:

- Java keystore – Contains the server's private key and public-key certificate. Protected by a keystore/certificate-entry password.
- Truststore – A keystore file containing only the certificates' public keys. Needed only if using self-signed certificates, not a Certificate Authority (CA).
- Keychain – Stores passwords, including to the server's keystore and to entries in these other files. The tools for creating and managing the Terracotta keychain file are provided with the Terracotta kit.
- Authorization – A .ini file with password-protected user accounts and their roles for servers and clients that connect to the server.
  Microsoft Active Directory and standard LDAP authentication/authorization are available options.

TIP: Secure cacerts File

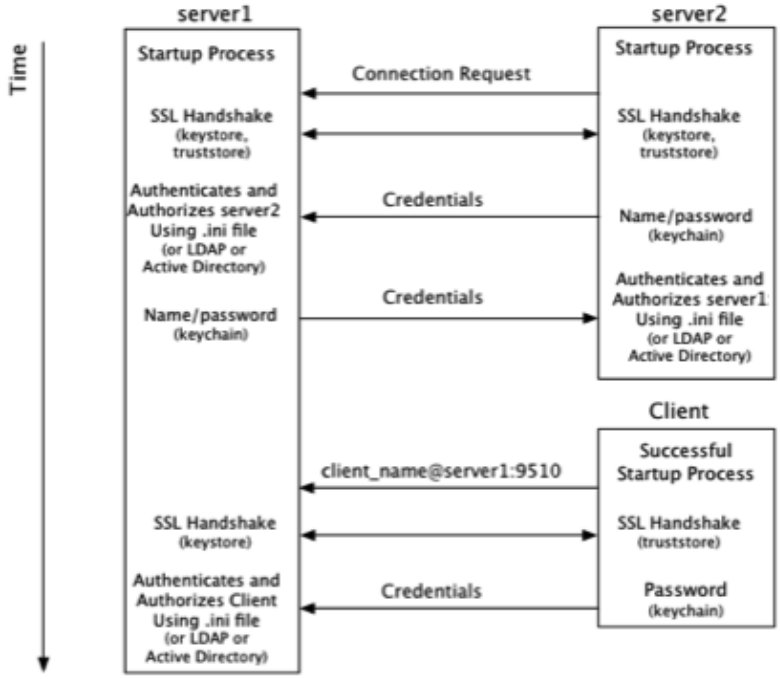
The standard Java cacerts file, located in `${JAVA_HOME}java.home/lib/security` by default, is a system-wide repository for CA root certificates included with the JDK. These certificates can play a part in certificate chains. Java documentation recommends that the cacerts file be protected by changing its default password and file permissions.
Each Terracotta client also has a keychain file that stores the password it uses to authenticate with the server.

All files are read on startup. Changes made to the files after startup cannot be read unless the cluster is restarted.

Process Diagram

The following diagram illustrates the flow of security information—and that information's origin in terms of the files described above—during initial cluster connections:

Process Diagram



From a Terracotta server's point of view, security checks take place at the time a connection is made with another node on the cluster:

1. After startup, servers can begin to make connection requests to servers named in configuration.
2. A connection request from server2 initiates the process of establishing a secure connection using SSL.
3. Server1 authenticates server2 using stored credentials. Credentials are also associated with a role that authorizes server2. The process is symmetrical: server2 authenticates and authorizes server1.
4. A connection request from a Terracotta client initiates the process of establishing a secure connection using SSL.
5. Server1 authenticates and authorizes the client using stored credentials and associated roles. Because clients may be communicating with any active server in the cluster during their lifetimes, the client must be able to authenticate with any active server. Clients should be able to authenticate against *all* servers in the cluster, since active servers may fail over to mirror servers.

From a Terracotta client's point of view, security checks occur at the time the client attempts to connect to an active server in the cluster:

1. The client uses a server URI that includes the client username.

   A typical (non-secure) URI is `<server-address>:<port>`. A URI that initiates a secure connection takes the form `<client-username>@<server-address>:<port>`.
2. A secure connection using SSL is established with the server.
3. The client sends a password fetched from a local keychain file. The password is associated with the client username.

Configuration Example

The following configuration snippet is an example of how security could be set up for the servers in the illustration above:

Configuration Example

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
...
  <servers secure="true">
    <server host="172.16.254.1" name="server1">
    ...
      <security>
        <ssl>
          <certificate>jks:server1alias@/the/path/keystore-file.jks</certificate>
        </ssl>
        <keychain>
          <url>file:///%(user.dir)/server1keychain.tkc</url>
        </keychain>
        <auth>
          <realm>com.tc.net.core.security.ShiroIniRealm</realm>
          <url>file:///%(user.dir)/my_auth.ini</url>
          <user>server1username</user>
        </auth>
      </security>
   ...
   </server>
    <server host="172.16.254.2" name="server2">
    ...
      <security>
        <ssl>
          <certificate>jks:server2alias@/the/path/keystore-file.jks</certificate>
        </ssl>
        <keychain>
          <url>file:///%(user.dir)/server2keychain.tkc</url>
        </keychain>
        <auth>
          <realm>com.tc.net.core.security.ShiroIniRealm</realm>
          <url>file:///%(user.dir)/my_auth.ini</url>
          <user>server2username</user>
        </auth>
      </security>
   ...
   </server>
   ...
 </servers>
...
</tc:tc-config>
```

See the configuration section below for more information on the configuration elements in the example.

Setting Up Server Security

To set up security on a Terracotta server, follow the steps in each of these procedures:

{toc-zonel3:3}

NOTE: Script names in the examples given below are for *NIX systems. Equivalent scripts are available for Microsoft Windows in the same locations. Simply substitute the .bat extension for the .sh extension shown and convert path delimiters as appropriate.

Create the Server Certificate and Add It to the Keystore

Each Terracotta server must have a keystore file containing a digital certificate and the associated private key.

This document assumes that you will use self-signed certificates. Self-signed certificates do not carry a digital signature from an official CA.

IMPORTANT SECURITY CONSIDERATION!: Self-signed certificates might be less safe than CA-signed certificates because they lack third-party identity verification and do not carry a digital signature from an official CA. Your organization might already have policies and procedures in place regarding the generation and use of digital certificates and certificate chains, including the use of certificates signed by a Certificate Authority (CA). To follow your organization's policies and procedures regarding using digital certificates, you might need to adjust the procedures outlined in this document.

When used for a Terracotta server, the following conditions must be met for certificates and their keystores:

- The keystore must be a Java keystore (JKS) compatible with JDK 1.6 or higher.
- The certificate must be keyed with the alias named in the value of the `<certificate>` element of the server's configuration.
- The Common Name (CN) field in the Distinguished Name must contain the hostname of the server, as configured in the server's configuration.
- The password securing the certificate must match the keystore's main password. In other words, **the store password and key passwords must be identical.**
- When using a self-signed certificate (not one signed by a trusted CA), create a custom truststore for storing public keys.
- If security is not the most important consideration, you can set the following Java system properties as shown:
  - `-Dtc.ssl.trustAllCerts=true` (use any certificates)
  - `-Dtc.ssl.disableHostnameVerifier=true` (bypass hostname verification)

If you have a keystore in place, but the server certificate is not already stored in the keystore, you must import it into the keystore. If the keystore does not already exist, you must create it.

Creating Self-Signed Certificates Using Java Keytool

For testing purposes, or if you intend to use self-signed certificates, use the Java keytool command to create the public-private key pair. You also use this command to create keystores and truststores, but note that keytool refers to truststores as "keystores" since there is only a logical difference.

Specifying a Custom Truststore

Note that if you are **not** using cacerts, the default Java truststore, the custom truststore must be specified with the `javax.net.ssl.trustStore` system property. In this case, you can choose to reset the custom truststore's default password with `javax.net.ssl.trustStorePassword`.
The following could be used to create both public-private keys (including a certificate) and a keystore file for the server called "server1" in the configuration example above:

```
keytool -genkey -keystore keystore-file.jks -dname "CN=172.16.254.1, OU=Terracotta, O=SAG, L=San
```

Note that the values passed to `-storepass` and `-keypass` match. Also, the field designating the Common Name (CN) must match the server's hostname, which matches the value entered in the server's configuration. This hostname can be an IP address or a resolvable domain name. If the `-dname` option is left out, a series of identity prompts (distinguished-name fields based on the X.500 standard) will appear before the server's entry is created in the keystore. The CN prompt appears as shown:

```
What is your first and last name?
   [Unknown]:
```

There are a number of other keytool options to consider, including `-keyalg` (cryptographic algorithm; default is DSA) and `-validity` (number of days until the certificate expires; default is 90). These and other options are dependent on your environment and security requirements. See the JDK documentation for more information on using the keytool.

Create a keystore and entry on each Terracotta server.

Exporting and Importing Certificates

Each server should have a copy of each other server's public-key certificate in its truststore.

The following could be used to export the certificate of the server called "server1" in the configuration example above:

```
keytool -export -alias server1alias -keystore keystore-file.jks \
    -file server1SelfSignedCert.cert
```

This "cert" file can now be used to import server1's certificate into the truststore of every other server. For example, to create a truststore and import server1's certificate on server2, copy the cert file to the working directory on server2 and use the following command:

```
keytool -import -alias server1alias -file server1SelfSignedCert.cert \
    -keystore truststore.jks
```

After the password prompt, information about the certificate appears, and you are prompted to trust it. You must repeat this process until every server has a truststore containing the public-key certificate of every other server in the cluster.

TIP: Use a Single Truststore

Instead of recreating the truststore on every server, create a single truststore containing every server's public key, then copy that to every server. This same truststore can also be used for clients..

Set Up the Server Keychain

The keystore and each certificate entry are protected by passwords stored in the server keychain file. The location of the keychain file is specified in the value of the `<url>` element under the `<keychain>` element of the server's configuration file.

For example, with this server configuration, when the server starts up, the keychain file would be searched for in the user's (process owner's) home directory. In the configuration example, a keychain file called `server1keychain.tkc` is searched for when server1 is started.

The keychain file should have the following entries:

- An entry for the local server's keystore entry.
- An entry for every server that the local server will connect to.

Entries are created using the keychain script found in the Terracotta kit's `tools/security/bin` directory.

Creating an Entry for the Local Server

Create an entry for the local server's keystore password:

```
tools/security/bin/keychain.sh -O <keychain-file> <certificate-URI>
```

where `<keychain-file>` is the file named in the server configuration's `<keychain>/<url>` element (including correct path), and `<certificate-URI>` is the URI value in the server configuration's `<ssl>/<certificate>` element. **Note**: The `<certificate-URI>` must match the server configuration's `<ssl>/<certificate>` element exactly, including the path to the keystore.

By default, the keychain file stores passwords using an obfuscation scheme, requiring the use of `-O` (hyphen capital letter O) with the keychain script for *any* operation on the file. To switch a more secure encryption-based scheme, see Using Encrypted Keychains.

If the keychain file does not exist, add the `-c` option to create it:

```
tools/security/bin/keychain.sh -O -c <keychain-file> <certificate-URI>
```

You will be prompted to enter a password to associate with the URI. **You must enter the same password used to secure the server's certificate in the keystore.**

For example, to create an entry for server1 from the configuration example above, enter:

```
tools/security/bin/keychain.sh -O server1keychain.tkc jks:server1alias@/the/path/keystore-file.jk

Terracotta Management Console - Keychain Client
Enter the password you wish to associate with this URL:   server1pass
Confirm the password to associate with this URL:   server1pass
Password for jks:server1alias@/the/path/keystore-file.jks successfully stored
```

Creating Entries for Remote Servers

Entries for remote servers have the format `tc://<user>@<host>:<group-port>`. Note that the value of `<user>` is specified in each server configuration's `<security>/<auth>/<user>` and is *not* related to the user running as process owner. If a value for `<security>/<auth>/<user>` is not specified, the username "terracotta" is used by default.

For example, to create an entry for server2 in server1's keychain, use:

```
tools/security/bin/keychain.sh -O server1keychain.tkc tc://server2username@172.16.254.2:9530
```

If the keychain file does not exist, add the `-c` option:

```
tools/security/bin/keychain.sh -O -c server1keychain.tkc tc://server2username@172.16.254.2:9530
```

You will be prompted to enter a password to associate with the entry `server2username@172.16.254.2:9530`.

An entry for server1 must also be added to server2's keychain:

```
tools/security/bin/keychain.sh -O server2keychain.tkc tc://server1@172.16.254.1:9530
```

Set Up Authentication/Authorization

Servers and clients that connect to a secured server must have credentials (usernames/passwords) and roles (authorization) defined. This section discusses the provided authentication/authorization mechanism, based on using a .ini file. To use LDAP or Microsoft Active Directory instead, see the LDAP/AD setup page.

Authentication and authorization are set up using the usermanagement script, located in the Terracotta kit's `tools/security/bin` directory. This script also creates the .ini file that contains the required usernames and roles (associated passwords are stored in the keychain file).

All nodes in a secured Terracotta cluster must have an entry in the server's .ini file:

- The local server itself
- All other servers
- All clients

Use the usermanagement script with the following format:

```
tools/security/bin/usermanagement.sh -c <file> <username> terracotta
```

where `-c` is required only if the file does not already exist. For servers, the `<username>` will be used as the value configured in `<security>/<auth>/<user>`. For clients, the username must match the one used to start the client.

Note: While the "terracotta" role is appropriate for Terracotta servers and clients, the "admin" role is necessary for performing system functions such as stopping servers. For more information about roles, refer to the User Roles section.

For example:

```
# Create the .ini file and add a server username and role.
tools/security/bin/usermanagement.sh -c my_auth.ini server1username terracotta

# Add another server.
tools/security/bin/usermanagement.sh my_auth.ini server2username terracotta

# Add a client.
tools/security/bin/usermanagement.sh my_auth.ini client1username terracotta

# Add a user with an "admin" (read/write) role.
tools/security/bin/usermanagement.sh my_auth.ini admin1username admin

# Add a user with a "terracotta" (read) role.
tools/security/bin/usermanagement.sh my_auth.ini console1username operator
```

The correct Shiro realm must be specified in the server configuration, along with the path to the .ini file:

```
...
<auth>
  <realm>com.tc.net.core.security.ShiroIniRealm</realm>
  <url>file:///%(user.dir)/my_auth.ini</url>
  <user>server1username</user>
</auth>
```

...

Configure Server Security

Security for the Terracotta Server Array is configured in the Terracotta configuration file (`tc-config.xml` by default). The following is an example security configuration:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
...
  <servers secure="true">
    <server host="172.16.254.1" name="server1">
    ...
      <security>
        <ssl>
          <certificate>jks:server1alias@/the/path/keystore-file.jks</certificate>
        </ssl>
        <keychain>
          <url>file:///%(user.dir)/server1keychain.tkc</url>
        </keychain>
        <auth>
          <realm>com.tc.net.core.security.ShiroIniRealm</realm>
          <url>file:///%(user.dir)/my_auth.ini</url>
          <user>server1username</user>
        </auth>
      </security>
   ...
   </server>
   ...
 </servers>
...
</tc:tc-config>
```

Every server participating in an SSL-based secured cluster must have a `<security>` block wherein security-related information is encapsulated and defined. The keystore, keychain, and .ini files named in the configuration must be available to every server in the cluster. LDAP or Microsoft Active Directory can be configured in place of file-based authentication/authorization.

The following table defines selected security-related elements and attributes shown in the configuration example.

Name	Definition	Notes
secure	Attribute in <servers> element. Enables SSL security for the cluster. DEFAULT: false.	Enables/disables SSL-based security globally.
certificate	Element specifying the location of the server's authentication certificate and its containing keystore file. The format for the certificate-keystore location is jks:alias@/path/to/keystore. "alias" must match the value used to key the certificate in the keystore file.	Only the JKS type of keystore is supported.
url	The URI for the keychain file (when under <keychain>) or for the authentication/authorization mechanism	These files are created and managed with the keychain and usermanagement scripts. If using Microsoft Active Directory or LDAP, an LDAP or LDAPS connection is

	(when under <auth>). These URIs are passed to the keychain or realm class to specify the keychain file or authentication/authorization source, respectively.	specified. The configured URL for locating the keychain file can be overridenn with the property `com.tc.security.keychain.url`.
realm	The Shiro security realm that determines the type of authentication/authorization scheme being used: file-based (.ini), Microsoft Active Directory, or standard LDAP.	This element's value is specified in the section covering the setup for the chosen authentication/authorization scheme.
user	The username that represents this server and is authenticated by other servers. This name is part of the server's credentials. Default username is "terracotta"	

{/toc-zone}

Enabling SSL on Terracotta Clients

Terracotta clients do not require any specific configuration to enable SSL connections to a Terracotta Server Array.

NOTE: Script names in the examples given below are for *NIX systems. Equivalent scripts are available for Microsoft Windows in the same locations. Simply substitute the .bat extension for the .sh extension shown and convert path delimiters as appropriate.

To enable SSL security on the client, ensure the following:

- Prepend the client username to the address used by the client to connect to the cluster.

  This should be the username that will be authenticated followed by an "at" sign ("@") and the address of an active server running in secure mode. The format is `<client-username>@<host>:<tsa-port>`. Prepending the username automatically causes the client to initiate an SSL connection.

  If the client has username `client1`, for example, and attempts to connect to the server in the configuration example, the address would be:

  ```
  client1@172.16.254.1:9510
  ```

  This URI replaces the address `<host>:<tsa-port>` used to start clients in non-SSL clusters.
- The client username and its corresponding password must match those in the server's .ini file or credentials in LDAP or Active Directory. The username is included in the URI, but the password must come from a local keychain entry that you create.

  The client credentials must be associated with the role "terracotta" or "admin".
- If Terracotta servers are using self-signed certificates (not certificates signed by a well-known CA), then you must specify a truststore for the client which contains the public key of every server in the cluster.

Create a Keychain Entry

The Terracotta client should have a keychain file with an entry for every Terracotta server in the cluster. The format for the entry uses the "tc" scheme:

```
tc://<client-username>@<host>:<tsa-port>
```

An entry for the server in the example configuration should look like the following:

```
tc://client1@172.16.254.1:9510
```

Use the keychain script in the Terracotta kit to add the entry:

```
tools/security/bin/keychain.sh -O clientKeychainFile tc://client1@172.16.254.1:9510
```

By default, the keychain file stores passwords using an obfuscation scheme, requiring the use of `-O` (hyphen capital letter O) with the keychain script for *any* operation on the file. To switch a more secure encryption-based scheme, see this page.

If the keychain file does not already exist, use the `-c` flag to create it:

```
tools/security/bin/keychain.sh -O -c clientKeychainFile tc://client1@172.16.254.1:9510
```

You will be prompted to enter a client password to associate with the URI.

This entry in the client's keychain file serves as the key for the client's password, and will be provided to the server along with the client username ("client1" in the example). **These credentials must match those in the server's .ini file or LDAP or Active Directory credentials.**

The Terracotta client searches for the keychain file in the following locations:

- %(user.home)/.tc/mgmt/keychain
- %(user.dir)/keychain.tkc
- The path specified by the system property `com.tc.security.keychain.url`

Example Using the Keychain Script

When you run the keychain script, the following prompt should appear:

```
Terracotta Management Console - Keychain Client
KeyChain file successfully created in clientKeychainFile
Enter the password you wish to associate with this URL:
Password for tc://client1@172.16.254.1:9510 successfully stored
```

Note that the script does not verify the credentials or the server address.

Using a Client Truststore

If Terracotta servers are using self-signed certificates (not certificates signed by a well-known CA), create a truststore on the client and import each server's public-key certificate into that truststore.

Using a Client Truststore

If you have already created a truststore for a server in the TSA, you can copy that file to each client after first importing that server's public-key certificate into the copy.

For the client to find the truststore, you must set the Java system property `javax.net.ssl.trustStore` to the location of the truststore file. In this case, note the existing secrets for opening the truststore and accessing each certificate.

TIP: Changing the Truststore Password

To change the existing truststore master password, use the Java system property `javax.net.ssl.trustStorePassword`.

Security With the Terracotta Management Server

Additional configuration is required for using a secured TSA with the Terracotta Management Server (TMS):

{toc-zone|3:3}

Configuring Identity Assertion

Add the following to each server's <security> block:

```
<security>
...
  <management>
    <ia> https://my-tms.mydomain.com:9443/tmc/api/assertIdentity</ia>
    <timeout>10000</timeout>
    <hostname>my-l2.mydomain.com</ hostname >
  </management>
</security>
```

where:

- `<timeout>` is the the timeout value (in milliseconds) for connections from the server to the TMS.
- `<ia>` is the HTTPS (or HTTP) URL with the domain of the TMS, followed by the port 9443 and the path `/tmc/api/assertIdentity`.

  If using HTTPS (recommended), export a public key from the TMS and import it into the server's truststore. You must also export a public key from the server and import it into the TMS's truststore, or copy the server's truststore (including the local server's public key) to the TMS.
- `<management><hostname>` is used only if the DNS hostname of the server does not match server hostname used in its certificate. If there is a mismatch, enter the DNS address of the server here.

You must export a public key from the TMS

JMX Authentication Using the Keychain

The following is **required** for server-to-client REST-agent authorization to succeed. Every node in the cluster must have the following entry in its keychain, all locked with the identical secret:

```
jmx:net.sf.ehcache:type=RepositoryService
```

In addition, server-server REST-agent communication must also be authorized using a keychain entry with the following format:

```
jmx://<user>@<host>:<group-port>
```

Note that the value of `<user>` is specified in each server configuration's `<security>/<auth>/<user>` and is *not* related to the user running as process owner.

For example, to create an entry for server2 in server1's keychain, use:

```
tools/security/bin/keychain.sh -O server1keychain.tkc jmx://server2username@172.16.254.2:9530
```

Each server must have an entry for itself and one for each other server in the TSA.

Setting Up Security on the TMS

An unsecured TMS cannot connect to a secured TSA. See this page to learn how to set up security on the TMS.

{/toc-zone}

Restricting Clients to Specified Servers (Optional)

By default, clients are not restricted to authenticate a specific set of servers when responding to REST requests. However, it is possible to explicitly list the servers that a client may respond to by using the <managementRESTService> element's `securityServiceLocation` attribute in the Ehcache configuration.

When this attribute is empty (or missing), no such restriction exists and the client will authenticate against any server in the cluster that meets the established security requirements. This is the recommended setting, as SSL connections and the authentication/authorization mechanism provide sufficient security.

In the case where an extra layer of security is required for the client's REST service, you can configure a list of allowed servers as follows:

```
<managementRESTService ...
  securityServiceLocation=" https://my-l2-node1/tmc/api/assertIdentity ,
     https://my-l2-node2/tmc/api/assertIdentity ">
```

where my-l2-node1 and my-l2-node2 are the servers' hostnames. Note, however, that any of the servers in a client's cluster can forward a REST request to that client at any time, and so all servers should be listed if this feature is used.

Running a Secured Server

Start a server in a secure Terracotta cluster using the start-tc-server script as usual. If you are using encrypted keychains, a master password must be entered at the command line during server startup (or set the server to automatically fetch the password).

Confirm Security Enabled

You can confirm that a server in a number of ways:

- Look for the startup message `Security enabled, turning on SSL`.
- Search for log messages containing "SSL keystore", "HTTPS Authentication enabled", and "Security enabled, turning on SSL".
- Attempt to make JMX connections to the server—these should fail.

Stopping a Secured Server

Stop a server in a secure Terracotta cluster using the stop-tc-server script with the following arguments:

- `-f <tc-config-file>` — A valid path to the self-signed certificate must have been specified in the server's configuration file.
- `-u <username>` — The user specified must have the "admin" role.
- `-w <password>`

For more information about the stop script, refer to Start and Stop Server Scripts.

Troubleshooting

You may encounter any of the following errors at startup:

TCRuntimeException: ... Wrong secret provided ?

The following error indicates that the keychain file uses the default obfuscation scheme, but that the `-O` flag was not used with the keychain script:

```
com.tc.exception.TCRuntimeException: com.terracotta.management.keychain.crypto.SecretMismatchExce
```

Be sure to use the `-O` flag whenever using the keychain script.

No Configured SSL certificate

The following error indicates that no SSL certificate was found for the server named "myServer":

```
Fatal Terracotta startup exception:


**********************************************************************
Security is enabled but server myServer has no configured SSL certificate.
**********************************************************************
```

Check that the expected SSL certificate was created for myServer and stored at the configured location.

IllegalStateException: Invalid cluster security configuration

These types of errors typically occur when the security section in the Terracotta configuration file is not set up properly. However, this type of error can also indicate problems elsewhere in the security setup. For example, an error similar to the following can occur:

```
java.lang.IllegalStateException: Invalid cluster security configuration. Unable to find connectio
```

This error indicates that credentials cannot be found for the server named "myOtherServer". These credentials may be missing from or do not exist in the configured authentication source.

RuntimeException: Couldn't access a Console instance to fetch the password from!

This results from using "nohup" during startup. The startup process requires a console for reading password entry. In fact, you cannot run the startup process in the background if it requires manual password entry. See this section for information on how to avoid having to manually enter the master keychain password.

TCRuntimeException: Couldn't create KeyChain instance ...

The keychain file specified in the Terracotta configuration cannot be found. Check for the existence of the file at the location specified in `<keychain>/<url>` or the property `com.tc.security.keychain.url`.

RuntimeException: Couldn't read from file ...

This error appears just after an incorrect password is entered for an encrypted keychain file.

RuntimeException: No password available in keyChain for ...

This error appears if no keychain password entry is found for the server's certificate. You must explicitly store the certificate password in the keychain file.

This error could also appear if the resolved hostname or IP address is different from the one in the keychain entry:

- `tc://terracotta@localhost:9530` is the entry, but when the server configuration is read then `localhost` is resolved to an IP address. The entry searched for becomes `tc://terracotta@<a.certain.ip.address>:9530`.
- `tc://terracotta@<a.certain.ip.address>:9530` is the entry, but when the server configuration is read then `<a.certain.ip.address>` is resolved to a host name. The entry searched for becomes `tc://terracotta@my.host.com:9530`.

Two Active Servers (Split Brain)

Instead of an active-mirror 2-server stripe, both servers assert active status after being started. This error can be caused by the failure of the SSL handshake. An entry similar to the following may appear in the server log:

```
2013-05-17 12:10:24,805 [L2_L2:TCWorkerComm # 1_W] ERROR com.tc.net.core.TCConnection - SSL hands
```

For each server, ensure that all keychain entries are accurate, and that the required certificates are available from the appropriate truststores.

No Messages Indicating Security Enabled

If servers start with no errors, but there are no messages indicating that security is enabled, ensure that the `<servers>` element contains `secure="true"`.

Setting Up LDAP-Based Authentication

Introduction

Instead of using the built-in user management system, you can set up authentication and authorization for the Terracotta Server Array (TSA) based on the Lightweight Directory Access Protocol (LDAP). This allows you to use your existing security infrastructure for controlling access to Terracotta clusters.

The two types of LDAP-based authentication supported are Microsoft Active Directory and standard LDAP. In addition, LDAPS (LDAP over SSL) is supported.

NOTE: Terracotta servers must be configured to use SSL before any Active Directory or standard LDAP can be used.

Configuration Overview

Active Directory and standard LDAP are configured in the <auth> section of each server's configuration block:

```
<servers secure="true">
  <server host="172.16.254.1" name="server1">
    ...
    <security>
      ...
      <auth>
        <realm>...</realm>
        <url>...</url>
        <user>...</user>
      </auth>
    </security>
...
</server>
```

Active Directory and standard LDAP are configured using the <realm> and <url> elements; the <user> element is used for connections between Terracotta servers and is not required for LDAP-related configuration.

For presentation, the URLs used in this document use line breaks. **Do not use line breaks** when creating URLs in your configuration.

Realms and Roles

The setup for LDAP-based authentication and authorization uses Shiro realms to map user groups to one of the following two roles:

- admin – This is the typical administrator role that can perform system functions such as shutting down servers and reloading configuration.
- terracotta – A typical operator's role, with no rights to perform system functions such as shutdowns or configuration reloads. Though they may overlap, note that the terracotta role is not completely a subset of the admin role.

URL Encoding

Certain characters used in the LDAP URL must be encoded, unless wrapped in a CDATA construct. Characters that may be required in an LDAP URL are described below:

- & (ampersand) – Encode as %26.
- { (left brace) – Encode as %7B.
- } (right brace) – Encode as %7D.
- Space – Encode as %20. *Spaces must always be encoded*, even if wrapped in CDATA.
- = (equals sign) – Does not require encoding.

Active Directory Configuration

Specify the realm and URL in the <security> section of the Terracotta configuration as follows:

```
<auth>
  <realm>com.tc.net.core.security.ShiroActiveDirectoryRealm</realm>
  <url>ldap://admin_user@server_address:server_port/searchBase=search_domain%26
       groupBindings=groups_to_roles</url>
  <user></user>
</auth>
```

Note the value of the <realm> element, which must specify the correct class (or Shiro security realm) for Active Directory. The components of the URL are defined in the following table.

Component	Description
ldap://	For the scheme, use either `ldap://` or `ldaps://`.
admin_user	The name of a user with sufficient rights in Active Directory to perform a search in the domain specified by searchBase. The password for this user password must be stored in the Terracotta keychain used by the Terracotta server, using as key the root of the LDAP URI, `ldap://admin_user@server_name:server_port`, with no trailing slash ("/").
server_address:server_port	The IP address or resolvable fully qualified domain name of the server, and the port for Active Directory.
searchBase	Specifies the Active Directory domain to be searched. For example, if the Active Directory domain is reggae.jamaica.org, then the format is `searchBase=dc=reggae,dc=jamaica,dc=org`.
groupBindings	Specifies the mappings between Active Directory groups and Terracotta roles. For example, `groupBindings=Domain%20Admins=admin,Users=terracotta` maps the Active Directory groups "Domain Admins" and "Users" to the "admin" and "terracotta" Terracotta roles, respectively. To be mapped, the named Active Directory groups must be part of the domain specified in searchBase; all other groups (including those with the specified names) in other domains are ignored.

For example:

```
<auth>
  <realm>com.tc.net.core.security.ShiroActiveDirectoryRealm</realm>
  <url>ldap://bmarley@172.16.254.1:389?searchBase=dc=reggae,dc=jamaica,dc=org%26
       groupBindings=Domain%20Admins=admin,Users=terracotta</url>
```

```
  <user></user>
</auth>
```

Standard LDAP Configuration

Specify the realm and URL in the <security> section of the Terracotta configuration as follows:

```
<auth>
  <realm>com.tc.net.core.security.ShiroLdapRealm</realm>
  <url>ldap://directory_manager@myLdapServer:636?
       userDnTemplate=cn=%7B0%7D,ou=users,dc=mycompany,dc=com%26
       groupDnTemplate=cn=%7B0%7D,ou=groups,dc=mycompany,dc=com%26
       groupAttribute=uniqueMember%26
       groupBindings=bandleaders=admin,bandmembers=terracotta</url>
  <user></user>
</auth>
```

Note the value of the <realm> element, which must specify the correct class (or Shiro security realm) for Active Directory. The components of the URL are defined in the following table.

Component	Description
ldap://	For the scheme, use either `ldap://` or `ldaps://`.
directory_manager	The name of a user with sufficient rights on the LDAP server to perform searches. No user is required if anonymous lookups are allowed. If a user is required, the user's password must be stored in the Terracotta keychain, using as key the root of the LDAP URL, `ldap://admin_user@server_name:server_port`, with no trailing slash ("/").
server_address:server_port	The IP address or resolvable fully qualified domain name of the server, and the LDAP server port.
userDnTemplate	Specifies user-template values. See the example below.
groupDnTemplate	Specifies group-template values. See the example below.
groupAttribute	Specifies the LDAP group attribute whose value uniquely identifies a user. By default, this is "uniqueMember". See the example below.
groupBindings	Specifies the mappings between LDAP groups and Terracotta roles. For example, `groupBindings=bandleaders=admin,bandmembers=terracotta` maps the LDAP groups "bandleaders" and "bandmembers" to the "admin" and "terracotta" Terracotta roles, respectively.

For example:

```
<auth>
  <realm>com.tc.net.core.security.ShiroLdapRealm</realm>
  <url>ldap://dizzy@172.16.254.1:636?
       userDnTemplate=cn=%7B0%7D,ou=users,dc=mycompany,dc=com%26
       groupDnTemplate=cn=%7B0%7D,ou=groups,dc=mycompany,dc=com%26
       groupAttribute=uniqueMember%26
       groupBindings=bandleaders=admin,bandmembers=terracotta</url>
  <user></user>
</auth>
```

This implies the LDAP directory structure is set up similar to the following:

```
+ dc=com
    + dc=mycompany
        + ou=groups
            + cn=bandleaders
                  | uniqueMember=dizzy
                  | uniqueMember=duke
            + cn=bandleaders
                  | uniqueMember=art
                  | uniqueMember=bird
```

If, however, the the LDAP directory structure is set up similar to the following:

```
+ dc=com
    + dc=mycompany
        + ou=groups
            + cn=bandleaders
                  | musician=dizzy
                  | musician=duke
            + cn=bandleaders
                  | musician=art
                  | musician=bird
```

then the value of groupAttribute should be "musician".

Using the CDATA Construct

To avoid encoding the URL, wrap it in a CDATA construct as shown:

```
<url><![CDATA[ldap://dizzy@172.16.254.1:636?
     userDnTemplate=cn={0},ou=users,dc=mycompany,dc=com&
     groupDnTemplate=cn={0},ou=groups,dc=mycompany,dc=com&
     groupAttribute=uniqueMember&
     groupBindings=bandleaders=admin,bandmembers=terracotta]]></url>
```

Using Encrypted Keychains

Introduction

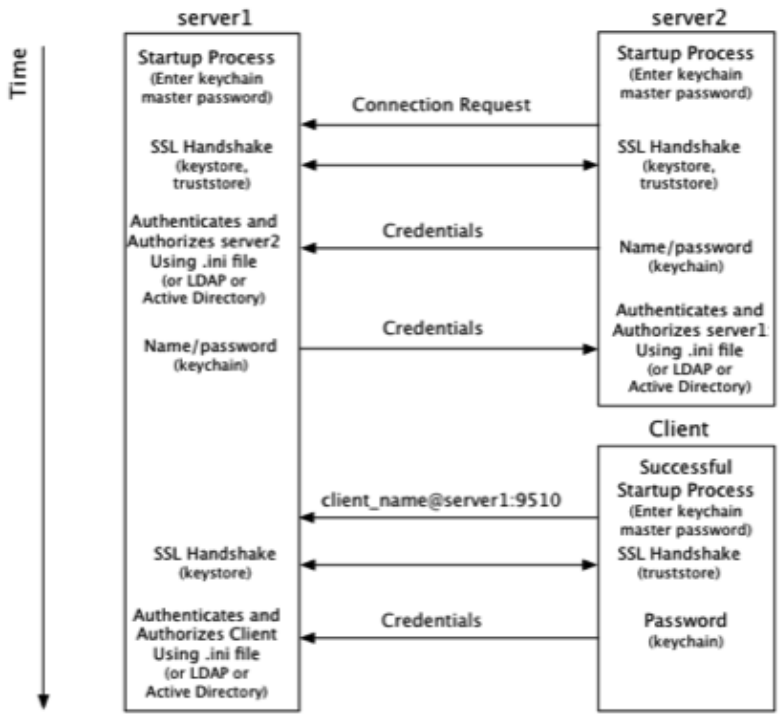
Security the Terracotta Server Array (TSA) is set up using Terracotta configuration, tools provided in the Terracotta kit, standard Java tools, and public key infrastructure (via standard digital X.509 digital certificates). This setup process is described in the TSA security setup document.

By default, the keychain script used to create Terracotta keychain files uses an obfuscation scheme to protect passwords. This scheme, while adequate for development environments or environments where keychain-file security cannot compromised, may not serve certain production environments.

If your environment requires stronger protection for keychain files, use the encryption scheme described in this document. The encryption scheme requires entry of a master password each time the keychain file is accessed.

Note: Except for the keychain setup, you must follow the setup instructions, including for authentication and SSL, as described in the TSA security setup document.

The following diagram shows where the master password is required in the startup process of a Terracotta cluster.



From a Terracotta server's point of view, security checks take place at startup and at the time a connection is made with another node on the cluster:

1. At startup, server1 requires a password to be entered directly from the console to complete its startup process. The password can also be read from a file to avoid manual entry.

2. A connection request from server2 initiates the process of establishing a secure connection using SSL.
3. Server1 authenticates server2 using stored credentials. Credentials are also associated with a role that authorizes server2. The process is symmetrical: server2 authenticates and authorizes server1.
4. A connection request from a Terracotta client initiates the process of establishing a secure connection using SSL.
5. Server1 authenticates and authorizes the client using stored credentials and associated roles. Because clients may be communicating with any active server in the cluster during their lifetimes, the client must be able to authenticate with any active server. Clients should be able to authenticate against *all* servers in the cluster, since active servers may fail over to mirror servers.

From a Terracotta client's point of view, security checks occur at the time the client attempts to connect to an active server in the cluster:

1. The client uses a server URI that includes the client username.

   A typical (non-secure) URI is `<server-address>:<port>`. A URI that initiates a secure connection takes the form `<client-username>@<server-address>:<port>`.
2. A secure connection using SSL is established with the server.
3. The client sends a password fetched from a local keychain file. The password is associated with the client username.

Note that the diagram and process shown above are similar to those found in the TSA security setup document. The main differences, described in this document, concern the use of the keychain file.

Configuration Example

The following configuration snippet is an example of how security could be set up for the servers in the illustration above:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
...
  <servers secure="true">
    <server host="172.16.254.1" name="server1">
    ...
      <security>
        <ssl>
          <certificate>jks:server1alias@/the/path/keystore-file.jks</certificate>
        </ssl>
        <keychain>
          <secret-provider>
            com.terracotta.management.security.ConsoleFetchingBackend
          </secret-provider>
          <url>file:///%(user.dir)/server1keychain.tkc</url>
        </keychain>
        <auth>
          <realm>com.tc.net.core.security.ShiroIniRealm</realm>
          <url>file:///%(user.dir)/myShiroFile.ini</url>
          <user>server1username</user>
        </auth>
      </security>
   ...
   </server>
    <server host="172.16.254.2" name="server2">
    ...
      <security>
```

```
        <ssl>
          <certificate>jks:server2alias@/the/path/keystore-file.jks</certificate>
        </ssl>
        <keychain>
          <url>file:///%(user.dir)/server2keychain.tkc</url>
        </keychain>
        <auth>
          <realm>com.tc.net.core.security.ShiroIniRealm</realm>
          <url>file:///%(user.dir)/myShiroFile.ini</url>
          <user>server2username</user>
        </auth>
      </security>
   ...
   </server>
   ...
 </servers>
...
</tc:tc-config>
```

See the configuration section for more information on the configuration elements in the example.

NOTE: Script names in the examples given below are for *NIX systems. Equivalent scripts are available for Microsoft Windows in the same locations. Simply substitute the .bat extension for the .sh extension shown and convert path delimiters as appropriate.

Configuring the Encrypted Server Keychain

By default, keychain files protect stored passwords using an obfuscation scheme. You can override this scheme by explicitly naming the secret provider for encryption:

```
<secret-provider>com.terracotta.management.security.ConsoleFetchingBackend</secret-provider>
```

This secret provider is also shown in the configuration example above.

TIP: Overriding the Configured Secret Provider

You can override the configured secret provider using the property `com.terracotta.SecretProvider`. For ex without changing configuration, use `com.terracotta.SecretProvider=com.terracotta.management.security.ObfuscatingSec`

Adding Entries to Encrypted Keychain Files

You must also add entries to the keychain file as described in the TSA security setup document, **but avoid using the `-o` flag when using the keychain script.**

For example, to create an entry for the local server's keystore password, use:

```
tools/security/bin/keychain.sh <keychain-file> <certificate-URI>
```

If the keychain file does not exist, add the `-c` option to create it:

```
tools/security/bin/keychain.sh -c <keychain-file> <certificate-URI>
```

You will be prompted for the keychain file's master password, then for a password to associate with the URI. **For the URI, you must enter the same password used to secure the server's certificate in the keystore.**

For example, to create an entry for server1 from the configuration example above, enter:

```
tools/security/bin/keychain.sh server1keychain.tkc jks:server1alias@keystore-file.jks

Terracotta Management Console - Keychain Client
Open the keychain by entering its master key:   xxxxxxx
Enter the password you wish to associate with this URL:   server1pass
Confirm the password to associate with this URL:   server1pass
Password for jks:server1alias@keystore-file.jks successfully stored
```

To create an entry for server2 in server1's keychain, use:

```
tools/security/bin/keychain.sh server1keychain.tkc tc://server2username@172.16.254.2:9530
```

Encrypted Client Keychain Files

For clients, set the secret provider with the following property:

```
com.terracotta.express.SecretProvider=net.sf.ehcache.terracotta.security.ConsoleFetchingSecretPro
```

Add entries to the keychain file as described in the TSA security setup document, **but avoid using the `-o` flag when using the keychain script.**

For example:

```
tools/security/bin/keychain.sh clientKeychainFile tc://client1@172.16.254.1:9510
```

When you run the keychain script, the following prompt should appear:

```
Terracotta Management Console - Keychain Client
KeyChain file successfully created in clientKeychainFile
Open the keychain by entering its master key:
```

Enter the master key, then answer the prompts for the secret to be associated with the server URI:

```
Enter the password you wish to associate with this URL:
Password for tc://client1@172.16.254.1:9510 successfully stored
```

Note that the script does not verify the credentials or the server address.

If the keychain file does not already exist, use the `-c` flag to create it:

```
tools/security/bin/keychain.sh -c clientKeychainFile tc://client1@172.16.254.1:9510
```

If creating the keychain file, you will be prompted for a master password. To automate the entry of the master password, see this section.

The Terracotta client searches for the keychain file in the following locations:

- %(user.home)/.tc/mgmt/keychain

- %(user.dir)/keychain.tkc
- The path specified by the system property `com.tc.security.keychain.url`

Security With the TMS

If you are using the Terracotta Management Server (TMS), you must set up JMX authentication. Every node in the cluster must have the following entry in its keychain, all locked with the identical secret:

```
jmx:net.sf.ehcache:type=RepositoryService
```

In addition, server-server REST-agent communication must also be authorized using a keychain entry using the format `jmx://<user>@<host>:<group-port>`.

Add entries to the keychain file as described in the TSA security setup document, **but avoid using the -o flag when using the keychain script.**

For example, to create an entry for server2 in server1's keychain, use:

```
tools/security/bin/keychain.sh server1keychain.tkc jmx://server2username@172.16.254.2:9530
```

Each server must have an entry for itself and one for each other server in the TSA.

Reading the Keychain Master Password From a File

Instead of manually entering the master keychain password at startup, you can set servers and clients to automatically read the password.

Note: Cygwin (on Windows) is not supported for this feature.

Servers Automatically Reading the Keychain Password

1. Implement the interface `com.terracotta.management.security.SecretProviderBackEnd` (located in the JAR `com.terracotta:security-keychain`) to fetch a password from a given file. For example:

```
package com.foo;

import com.terracotta.management.security.SecretProviderBackEnd;

import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class MySecretProvider implements SecretProviderBackEnd {
  private byte[] bytes;

  // This method reads the password into a byte array.
  @Override
  public void fetchSecret() {
    try {
      bytes = readPasswordFile("password.pw");
    } catch (IOException ioe) {
```

```
      throw new RuntimeException("Cannot read password from file", ioe);
    }
  }

  private byte[] readPasswordFile(String filename) throws IOException {
    FileInputStream fis = new FileInputStream(filename);
    try {
      byte[] buffer = new byte[64];
      ByteArrayOutputStream baos = new ByteArrayOutputStream();

      while (true) {
        int read = fis.read(buffer);
        if (read == -1) {
          break;
        }
        baos.write(buffer, 0, read);
      }

      return baos.toByteArray();
    } finally {
      fis.close();
    }
  }

  // This method returns the byte array containing the password.
  @Override
  public byte[] getSecret() {
    return bytes;
  }
}
```

2. Create a JAR containing your implementation (MySecretProvider), then copy it to the BigMemory Max `server/lib` directory.
3. Assuming the new JAR file is called `my-secret-provider.jar`, edit the start-tc-server script in the BigMemory Max `server/bin` as follows:

   UNIX/LINUX
   Change the line

   ```
   -cp "${TC_INSTALL_DIR}/lib/tc.jar" \
   ```

   to

   ```
   -cp "${TC_INSTALL_DIR}/lib/tc.jarr:${TC_INSTALL_DIR}/lib/my-secret-provider.jar" \
   ```

   MICROSOFT WINDOWS
   Change the line

   ```
   set CLASSPATH=%TC_INSTALL_DIR%\lib\tc.jar
   ```

   to

   ```
   set CLASSPATH=%TC_INSTALL_DIR%\lib\tc.jar;%TC_INSTALL_DIR%\lib\my-secret-provider.jar
   ```
4. Ensure that the server's configuration includes the <secret-provider> element specifying your implementation:

   ```
   <security>
     ...
     <keychain>
   ```

```
          <url>/path/to/my/keychain</url>
          <secret-provider>com.foo.MySecretProvider</secret-provider>
        </keychain>
        ...
      </security>
```

At startup, the server will read the keychain password from the file specified in your implementation.

For a simpler solution, you could instead hardcode the password:

```
package com.foo;

 import com.terracotta.management.security.SecretProviderBackEnd;

 public class MySecretProvider implements SecretProviderBackEnd {

   // This method returns the byte array containing the password.
   @Override
   public byte[] getSecret() {
     return new byte[] {'p', 'a', 's', 's', 'w', 'o', 'r', 'd'};
   }

   @Override
   public void fetchSecret() {
   }
 }
```

Clients Automatically Reading the Keychain Password

You can set up Terracotta clients to read their keychain's master password in a similar way as for servers. Import `org.terracotta.toolkit.SecretProvider` and override `fetchSecret()` and `getSecret()` as shown above.

Instead of packaging the implementation in a JAR, simply use the system property `com.terracotta.express.SecretProvider` to specify your implementing class.

Terracotta Server Array Operations

Automatic Resource Management

Terracotta Server Array resource management involves self-monitoring and polling to determine the real-time size of the data set and assess the amount of memory remaining according to user-configured limitations. In-memory data can be managed from three directions:

1. **Time** – TTI/TTL settings can be configured to expire entries that will then be evicted by the new TSA eviction implementation. You can also configure caches so that their entries are eternal, or you can pin entries or caches so that they are never evicted.
2. **Size** – The total amount of BigMemory managed by the TSA can be configured using the `maxDataSize` child element in the tc-config.xml file.
3. **Count** – The total number of entries per cache can be configured using the `maxEntriesInCache` attribute in the ehcache.xml file.

Eviction

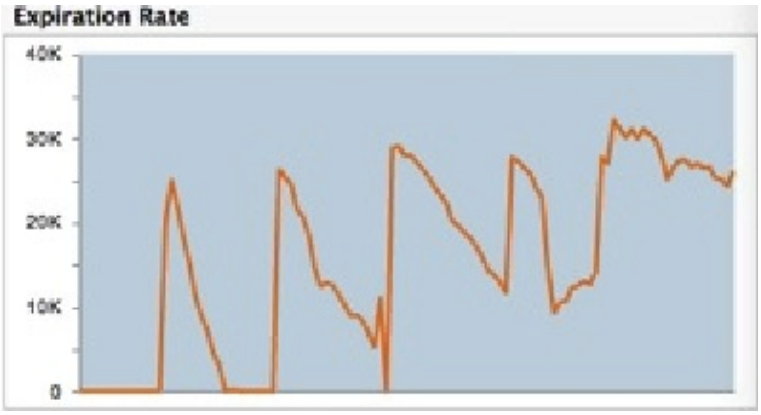
All data is kept in memory, and the TSA runs evictions in the background to keep the data set within its limitations. Eviction of entries from the data set reduces the amount of data before the memory becomes full. The criteria for an entry to be eligible for eviction are:

- It is not on a Terracotta client (L1).
- It is not pinned to a Terracotta server.
- It is held in a cache backed by a System of Record (SOR).

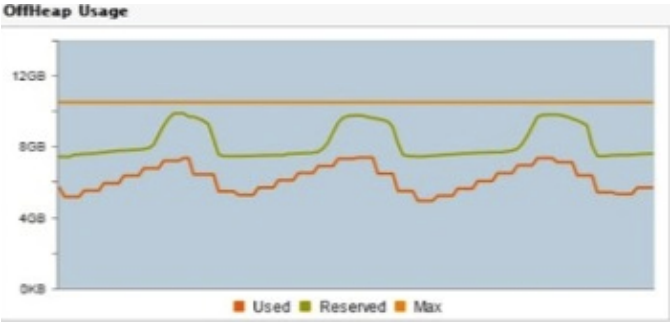
Store vs. Cache

BigMemory's in-memory data is treated as a "store" when BigMemory owns the data, and as a "cache" when the data also resides in a System of Record (SOR). Generally, data that is created by BigMemory and run-time data created by your application are examples of data that is treated as a store. The TSA does not evict data stores because they are the only or primary records. The TSA can evict cached data because that data is backed up in an SOR. (Distinctions in data structures are handled automatically, but you can also use the Terracotta Toolkit API to configure customized data structures.)
Eviction is done by the following evictors, which work together in the background:

1. The periodic evictor is activated on an as-needed basis. It removes expired entries based on TTI/TTL settings. The Server Expiration Rate graph in the TMC shows the activity of the periodic evictor.



2. The resource-based evictor is activated by the periodic TTI/TTL eviction scheduler, as well as by resource monitoring events. This evictor continuously polls heap and off-heap stores to check current resource usage. At approximately 10% usage of the `maxDataSize` (configured in the tc-config.xml file), it starts looking for TTI/TTL-expired elements to evict. At approximately 90% usage, it evicts live as well as expired elements. If a monitored resource goes over its critical threshold, this evictor will work continually until the monitored resource falls below the critical threshold.



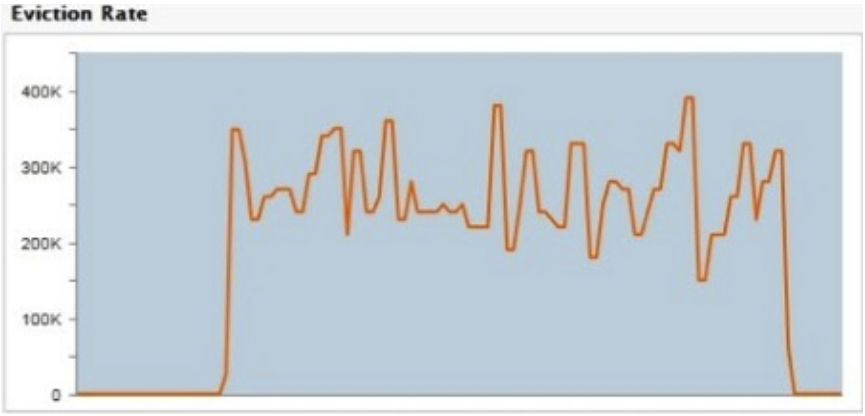
This evictor actually monitors two thresholds, used space and reserved space. Resource eviction is triggered if either the reserved or used is above its threshold. Once resource eviction has started, both used and reserve must fall below their respective thresholds before resource eviction ends.

The Offheap Usage graph in the TMC provides the following information:

- Off-heap Max is the configured maxDataSize
- Off-heap Reserved represents usage of the space that is reserved for the system
- Off-heap Used represents the amount BigMemory that is in use

3. The capacity-based evictor is activated when a cache goes over its maximum count (as configured with `maxEntriesInCache`), plus an overshoot count, and it attempts to bring the size of the cache to the max capacity. The `maxEntriesInCache` attribute must be present in the Ehcache configuration (do not include `maxEntriesInCache` in your configuration if you do not want the capacity evictor to run). If `maxEntriesInCache` is not set, it gets the default value 0, which means that the cache is unbounded and will not undergo capacity eviction (but periodic and resource evictions are still allowed).

The Server Eviction Rate graph in the TMC shows the activity of the resource and capacity evictors.



Customizing the Eviction Strategy

Based upon the three types of evictors, there are three strategies that you can employ for controlling the size of the TSA's data set:

1. Set the Time To Idle (TTI) or Time to Live (TTL) options for any entry in your data set. After the time has expired, the periodic evictor will clear the entry.
2. Set the `maxDataSize` element to control how much BigMemory should be used before the resource-based evictor is activated.
3. Set the `maxEntriesInCache` attribute to control when the capacity-based evictor is activated.

Near-Memory-Full Conditions

In a near-memory-full condition, where evictions are not happening fast enough to keep the data set within its BigMemory size limitations, the TSA will put a throttle on operations for a temporary period while it attempts to automatically recover in the background. If unable to recover, the TSA will move into "restricted mode" to prevent out-of-memory errors. The Terracotta Management Console (TMC) uses events to report when the TSA enters restricted mode and allows you to execute additional recovery measures.

Summary of TSA behavior in near-memory-full conditions:

- If usage reaches its critical threshold, T1, then it enters "throttle mode," where writes are slowed while the TSA attempts to evict eligible cache entries in order to bring memory usage within the configured range.
- If usage reaches its halt threshold, T2, then it enters "restricted mode," where writes are blocked, an exception is thrown, and operator intervention is needed to reduce memory usage.
- When usage falls below T1, then the TSA returns to normal operation.

	Throttle mode	**Restricted mode**
Entered when	Used or Reserved usage crosses its critical threshold	Used or Reserved usage crosses its halt threshold
Operator event	"TPS seems really low; marking us as being throttled'	"We're in restricted mode; waiting a while and retrying"
Data access	Modifications to in-memory data are slowed	Modifications to in-memory data are blocked
Allowed operations	All cache operations still allowed	Only gets, removes, and config changes are allowed

Actions	Evictions continue automatically in the background	Operator intervention required to make additional evictions
State of the data	Evictable data is still present	No more data present in memory that can be evicted by the evictor (all caches are pinned)
Recovery	Automatic	From the TMC or programmatically, clear caches and/or remove entries from a data set
Back to normal operation	As soon as the background evictions have time to catch up and reduce the data set to within its limitations	After user intervention clears space, the TSA will automatically continue with normal operation

Restricted Mode Operations

If the TSA is temporarily under restricted mode, any change to the data set which may result in increased resource utilization is not allowed, including all put and replace methods. Restricted mode does allow gets, removes, configuration changes, and other operations.

Recovery

Recovery from throttle mode is automatic, as soon as the background evictions have time to reduce the data set to within its limitations.

If the TSA enters restricted mode, operator events will be logged in the TMC, and user or programmatic intervention is necessary. In the TMC, you can initiate actions to manually reduce the data set. You can also anticipate operator events and use programmatic logic to respond appropriately.

The following actions are recommended for reducing the data set:

- Clear caches (from the TMC or programmatically)
- Remove entries from data sets programmatically

Note: Because eviction in restricted mode is resource-driven, changing TTI/TTL or maximum capacity will not move the TSA out of restricted mode.

To clear caches from the TMC, click the **Application Data** tab and the **Management** sub-tab. Each cache will have a clickable option to **Clear Cache**. Note that caution should be used when considering whether to clear a pinned cache.

Cluster Events

Cluster events report topology changes, performance issues, and errors in operations. These events are logged by both Terracotta server (L2) and client (L1), and can also be viewed in the TMC.

By default, the L2 stores a maximum of 100 events in memory, and this is the number pulled by the TMC. To edit that number, use the Terracotta property `l2.operator.events.store`. To set the property in the Terracotta configuration file, use:

```
<tc-properties>
  ...
  <property name="l2.operator.events.store" value="500" />
</tc-properties>
```

Event Types and Definitions

The following table describes the types of event that can be found in logs or viewed in the TMC.

Event Category Type	Level	Cause	Action	Notes
ClusterStateEvents old.db	INFO	Active L2 restarting with data recovered from disk.		
ClusterTopologyEvent config.reloaded	INFO	Cluster configuration was reloaded.		
DGCMessages dgc.started	INFO	Periodic DGC, which was explicitly enabled in configuration, has started a cleanup cycle.	If periodic DGC is unneeded, disable to improve overall cluster performance.	Periodic DGC, which is disabled by default, is mostly useful for toolkit, which lacks automatic handling of distributed garbage.
DGCMessages dgc.finished	INFO	Periodic DGC, which was explicitly enabled in configuration, ended a cleanup cycle.	If periodic DGC is unneeded, disable to improve overall cluster performance.	Periodic DGC, which is disabled by default, is mostly useful for toolkit, which lacks automatic handling of distributed garbage.
DGCMessages dgc.canceled	INFO	Periodic DGC, which was explicitly enabled in configuration, has been cancelled due to an interruption (for example, by a failover operation).	If periodic DGC is unneeded, disable to improve overall cluster performance.	Periodic DGC, which is disabled by default, is mostly useful for toolkit, which lacks automatic handling of distributed garbage.
DGCMessages inlineDgc.cleanup .started	INFO	L2 starting up as active with existing data, triggering inline DGC		Only seen as server starts up as active (from recovery) from restartable.
DGCMessages inlineDgc.cleanup .finished	INFO	Inline DGC operation completed		
DGCMessages inlineDgc.cleanup	INFO	Inline DGC operation	Investigate any unusual cluster	Possibly occurs during failover, but other events should indicate real cause.

.canceled		interrupted.	behavior or other events.	
HAMessages node.joined	INFO	Specified node has joined the cluster.		
HAMessages node.left	WARN	Specified node has left the cluster.	Check why the node has left (for example: long gc, network issues, or issues with local node resources).	
HAMessages clusterNode StateChanged	INFO	L2 changing state (for example, from initializing to active).	Check to see that state change is expected.	
HAMessages handshake.reject	ERROR	L1 unsuccessfully trying to reconnect to cluster, but has already be expelled.	L1 should go into a rejoin operation or must be restarted manually.	
MemoryManager high.memory .usage	WARN	Heap usage has crossed its configured threshold.	If combined with longgc, could imply need to relieve memory pressure on L1.	The default threshold can be reconfigured through the `high.memory.usage` tc-property (refer to tc.properties).
MemoryManager long.gc	WARN	A full GC longer than the configured threshold has occurred.	Reduce cache memory footprint in L1. Investigate issues with application logic and garbage creation.	The default threshold is 8 seconds, and can be reconfigured in tc.properties using `longgc.threshhold`. Occurrence of this event could help diagnose certain failures, such as in healthchecking.
ResourceManagement resource.nearcapacity	WARN	L2 entered throttled mode, which could be a temporary condition (e.g., caused by bulk-loading) or indicate insufficient	See the section on responses to memory pressure.	After emitting this, L2 can emit `resource.capacityrestored` (return to normal mode) or resource.fullcapacity (move to restricted mode), based on resource availability.

		allocation of memory to L1s		
ResourceManagement resource.fullcapacity	ERROR	L2 entered restricted mode, which could be a temporary condition (e.g., caused by bulk-loading) or indicate insufficient allocation of memory to L1s	See the section on responses to memory pressure.	After emitting this, L2 can emit `resource.capacityrestored` (return to normal mode), based on resource availability.
ResourceManagement resource .capacityrestored	INFO	L2 returned to normal from throttled or restricted mode.		
SystemSetupEvent time.different	WARN	System clocks are not aligned.	Synchronize system clocks.	The default tolerance is 30 seconds, and can be reconfigured in tc.properties using `time.sync.threshold`. Note that overly large tolerance can introduce unpredictable errors and behaviors.
ZapEvents zap.received	CRITICAL	Another L2 is trying to cause the specified one to restart ("zap").	Investigate a possible "split brain" situation (a mirror L2 behaves as an active) if the specified L2 does not obey the restart order.	A "zap" operation happens only within a mirror group.
ZapEvents zap.accepted	CRITICAL	The specified L2 is accepting the order to restart ("zap" order).	Check state of the target L2 to ensure that it restarts as mirror or manually restart the L2.	A "zap" order is issued only within a mirror group.
ZapEvents dirty.db	WARN	A mirror L2 is trying to join with data in place.	If the mirror does not automatically restart and wipe its data, its data may need to be manually wiped and	Restarted mirror L2s must wipe their data to resync with the active L2. This is normally an automatic operation that should not require action.

before it's restarted.

Live Backup of Distributed In-memory Data

Backups of the entire data set across all stripes (mirror groups) of the Terracotta Server Array can be made using the TMC Backup feature. This feature creates a time-stamped backup of each stripe's data, providing a snapshot of the TSA's in-memory data.

The Backup feature is available when fast restartability is enabled for the TSA (`<restartable enabled="true"/>` in the tc-config.xml).

Creating a Backup

From the TMC, select the **Administration** tab and the **Backups** sub-tab. Click the **Make Backup** button to perform a backup. The TMC sends a backup request to all stripes in the cluster.

In order to capture a consistent snapshot of the in-memory data, the backup function creates a pause in transactions, allowing any unfinished transactions to complete, and then the backup is written. This allows the backup to be a consistent record of the entries in-memory, as well as search and other indices.

Note that when backing up a cluster, each stripe is backed up independently and at a slightly different time than the other stripes.

When complete, a window appears that confirms the backup was taken and provides the time-stamped file name(s) of the backup.

Backup Directory

Backups are saved to the default directory `data-backup`, unless otherwise configured in the tc-config.xml. Terracotta automatically creates `data-backup` in the directory containing the Terracotta server's configuration file (tc-config.xml by default).

You can override the default directory by specifying a different backup directory in the server's configuration file using the `<data-backup>` property:

```
<servers>
   <server name="Server1">
      <data>/opt/terracotta/server1-data</data>
      <data-backup>path/to/my/backup/directory</data-backup>
      <offheap>
         <enabled>true</enabled>
         <maxDataSize>2g</maxDataSize>
      </offheap>
   </server>
   <restartable enabled="true"/>
</servers>
```

Restoring Data from a Backup

If the TSA fails, on restart it automatically restores data from its data directory, recreating the application state. If the current data files are corrupt or missing, or in other situations where an earlier snapshot of data is

required, you can restore them from backups:

1. Shut down the Terracotta cluster.
2. (Optional) Make copies of any existing data files.
3. Delete the existing data files from your Terracotta servers.
4. Copy the backup data files to the directory from which you deleted the original (existing) data files.
5. Restart the Terracotta cluster.

Server and Client Reconnections

A reconnection mechanism restores lost connections between active and mirror Terracotta server instances. See Automatic Server Instance Reconnect for more information.

A Terracotta Server Array handles perceived client disconnection (for example, a network failure, a long client GC, or node failure) based on the configuration of the HealthChecker or Automatic Client Reconnect mechanisms. A disconnected client also attempts to reconnect based on these mechanisms. The client tries to reconnect first to the initial server, then to any other servers set up in its Terracotta configuration. To preserve data integrity, clients resend any transactions for which they have not received server acks.

Changing Cluster Topology in a Live Cluster

Using the TMC, you can change the topology of a live cluster by reloading an edited Terracotta configuration file.

Note the following restrictions:

- Only the removal or addition of <server> blocks in the <servers> or <mirror-group> section of the Terracotta configuration file are allowed.
- All servers and clients must load the same configuration file to avoid topology conflicts.

  Servers that are part of the same server array but do not share the edited configuration file must have their configuration file edited and reloaded as shown below. Clients that do not load their configuration from the servers must have their configuration files edited to exactly match that of the servers.

Note: Changing the topology of a live cluster will not affect the distribution of data that is already loaded in the TSA. For example, if you added a stripe to a live cluster, the data in the server array would not be redistributed to utilize it. Instead, the new stripe could be used for adding new caches, while the original servers would continue to manage the original data.

Adding a New Server

To add a new server to a Terracotta cluster, follow these steps:

1. Add a new <server> block to the <servers> or <mirror-group> section in the Terracotta configuration file being used by the cluster. The new <server> block should contain the minimum information required to configure a new server. It should appear similar to the following, with your own values substituted:

   ```
   <server host="myHost" name="server2" >
   ```

```
    <data>%(user.home)/terracotta/server2/server-data</data>
    <logs>%(user.home)/terracotta/server2/server-logs</logs>
    <tsa-port>9513</tsa-port>
</server>
```

2. Make sure you are connected to the TMC, and that the TMC is connected to the target cluster. See the TMC documentation for more information on using the TMC.
3. With the target cluster selected in the TMC, click the **Administration** tab, then choose the **Change Topology** panel.
4. Click **Reload**.
   A message appears with the result of the reload operation. A successful operation logs a message similar to the following:

   ```
   2013-03-14 13:25:44,821 INFO - Successfully overridden server topology from file at
   ```

   '/bigmemory-max-4/tc-config.xml'.
5. Start the new server.

Removing an Existing Server

To remove a server from a Terracotta cluster configuration, follow these steps:

1. Shut down the server you want to remove from the cluster.
   If you shutting down an active server, first ensure that a backup server is online to enable failover.
2. Delete the <server> block associated with the removed server from the Terracotta configuration file being used by the cluster. Make sure you are connected to the TMC, and that the TMC is connected to the target cluster.
   See the TMC documentation for more information on using the TMC.
3. With the target cluster selected in the TMC, click the **Administration** tab, then choose the **Change Topology** panel.
4. Click **Reload**.
   A message appears with the result of the reload operation. A successful operation logs a message similar to the following:

   ```
   2013-03-14 13:25:44,821 INFO - Successfully overridden server topology from file at '/bigm
   ```

   The TMC will also display the event `Server topology reloaded from file at '/bigmemory-max-4/tc-config.xml'`.

Editing the Configuration of an Existing Server

If you edit the configuration of an existing ("live") server and attempt to reload its configuration, the reload operation will fail. However, you can successfully edit an existing server's configuration by following these steps:

1. Remove the server by following the steps in Removing an Existing Server. Instead of deleting the server's <server> block, you can comment it out.
2. Edit the server's <server> block with the changed values.
3. Add (or uncomment) the edited <server> block.
4. In the TMC's **Change Server Topology** panel, click **Reload**. A message appears with the result of the reload operation.

NOTE: To be able to edit the configuration of an existing server, all clients must load their configuration from the Terracotta Server Array. Clients that load configuration from another source will fail to remain connected to the TSA due to a configuration mismatch.

Production Mode

Production mode can be set by setting the Terracotta property in the Terracotta configuration:

```
<tc-properties>
  ...
  <property name="l2.enable.legacy.production.mode" value="true" />
</tc-properties>
```

Production mode requires the `--force` flag to be used with the stop-tc-server script if the target is an active server with no mirror.

Distributed Garbage Collection

There are two types of DGC: periodic and inline. The periodic DGC is configurable and can be run manually (see below). Inline DGC, which is an automatic garbage-collection process intended to maintain the server's memory, runs even if the periodic DGC is disabled.

Note that the inline DGC algorithm operates at intervals optimal to maximizing performance, and so does not necessarily collect distributed garbage immediately.

Running the Periodic DGC

The periodic DGC can be run in any of the following ways:

- run-dgc shell script – Call the run-dgc shell script to trigger DGC externally.
- JMX – Trigger DGC through the server's JMX management interface.

By default, DGC is disabled in the Terracotta configuration file in the `<garbage-collection>` section. However, even if disabled, it will run automatically under certain circumstances when clearing garbage is necessary but the inline DGC does not run (such as when a crashed server returns to the cluster).

Monitoring and Troubleshooting the DGC

DGC events (both periodic and inline) are reported in a Terracotta server instance's logs. DGC events can also be monitored using the Terracotta Management Console.

If DGC does not seem to be collecting objects at the expected rate, one of the following issues may be the cause:

- Java GC is not able to collect objects fast enough. Client nodes may be under resource pressure, causing GC collection to run behind, which then causes DGC to run behind.
- Certain client nodes continue to hold references to objects that have become garbage on other nodes, thus preventing DGC from being able to collect those objects.

If possible, shut down all Terracotta clients to see if DGC then collects the objects that were expected to be collected.

Starting up TSA or CLC as Windows Service using the Service Wrapper

You might want to run the Terracotta Server Array or the Cross-Language Connector, which are Java programs, as a **Windows service**. If so, use the Service Wrapper located inside the kit at `$installdir/server/wrapper` (or, for 3.7, `$installdir/wrapper`).

Set JAVA_HOME

To start the service, set your `JAVA_HOME` in `conf/wrapper-tsa.conf` or `conf/wrapper-clc.conf`. For example:

```
set.JAVA_HOME=C:/Java/jdk1.7.0_21
```

The wrapper does not read your `JAVA_HOME` from the environment. For Windows, if you do not want to set it in the configuration file, comment it out and set `JAVA_HOME` in the registry instead.

Configuration Files

For the Cross-Language Connector, you need these configuration files:

- conf/cross-language-config.xml
- conf/ehcache.xml

For the TSA, you need this configuration file:

- conf/tc-config.xml

Overwrite those files with your own. If you want to change these file names, modify the names in the wrapper configurations.

Modify the `TSA conf/wrapper-tsa.conf` file to match the server name in your tc-config.xml:

```
set.SERVER_NAME=server0
```

where *server0* represents the name of the server you want to start.

Set Permissions

The services are controlled by an Administrator user, so you have to confirm for every action, such as install, start, stop, remove.

In addition, the Administrator user needs to have read/write permission for the "wrapper" directory.

Install and Start the Service

The wrapper service is located at `$installdir/server/wrapper` .

Install and Start the Service

To install the service wrapper, run the script with the install parameter:

```
%> bin/tsa-service.bat install
```

NOTE: The examples in this section show the TSA script. For the Cross-Language Connector, use the `clc-service` or `clc-service.bat` script.

Then you can either start/stop the service:

```
%> bin/tsa-service.bat start
%> bin/tsa-service.bat stop
```

If you want to remove the service:

```
%> bin/tsa-service.bat remove
```

There are more commands available when you run the script without any parameter:

```
%> bin/tsa-service.bat
```

Changing Wrapper Configuration

There are comments in `wrapper-tsa.conf` and `wrapper-clc.conf` to explain each parameter. If you need to modify JVM system properties, classpath, or command line parameters, follow the current pattern. Pay close attention to their numerical order and parameter counts.

For more information, see http://wrapper.tanukisoftware.com/doc/english/properties.html

Terracotta Configuration Reference

Introduction

This document is a reference to all of the Terracotta configuration elements found in the Terracotta configuration file. The Terracotta configuration file is named `tc-config.xml` by default.

You can use a sample configuration file provided in the kit as the basis for your Terracotta configuration. Some samples have inline comments describing the configuration elements. Be sure to start with a clean file for your configuration.

The Terracotta configuration XML document is divided into the sections <servers> and <clients>. Each of these sections provides a number of configuration options relevant to its particular configuration topic.

Configuration Variables

Certain variables can be used that are interpolated by the configuration subsystem using local values:

Variable	Interpolated Value
%h	The fully-qualified hostname
%i	The IP address
%o	The operating system
%v	The version of the operating system
%a	The CPU architecture
%H	The home directory of the user running the application
%n	The username of the user running the application
%t	The path to the temporary directory (for example, /tmp on *NIX)
%D	Time stamp (yyyyMMddHHmmssSSS)
%(_system property_)	The value of the given Java system property

These variables can be used where appropriate, including for elements or attributes that expect strings or paths for values:

- the "name", "host" and "bind" attributes of the `<server>` element
- the password file location for JMX authentication
- client logs location
- server logs location
- server data location

NOTE: Value of %i

The variable %i is expanded into a value determined by the host's networking setup. In many cases that setup is in a `hosts` file containing mappings that may influence the value of %i. Test this variable in your production environment to check the value it interpolates.

Using Paths as Values

Some configuration elements take paths as values. Relative paths are interpreted relative to the current working directory (the directory from which the server was started). Specifying an absolute path is recommended.

Overriding tc.properties

Every Terracotta installation has a default `tc.properties` file containing system properties. Normally, the settings in `tc.properties` are pre-tuned and should not be edited.

If tuning is required, you can override certain properties in `tc.properties` using `tc-config.xml`. This can make a production environment more efficient by allowing system properties to be pushed out to clients with `tc-config.xml`. Those system properties would normally have to be configured separately on each client.

Setting System Properties in tc-config

To set a system property with the same value for all clients, you can add it to the Terracotta server's `tc-config.xml` file using a configuration element with the following format:

```
<property name="<tc_system_property>" value="<new_value>" />
```

All `<property />` tags must be wrapped in a `<tc-properties>` section placed at the beginning of `tc-config.xml`.

For example, to override the values of the system properties `l1.cachemanager.enabled` and `l1.cachemanager.leastCount`, add the following to the beginning of `tc-config.xml`:

```
<tc-properties>
  <property name="l1.cachemanager.enabled" value="false" />
  <property name="l1.cachemanager.leastCount" value="4" />
</tc-properties>
```

Override Priority

System properties configured in `tc-config.xml` override the system properties in the default `tc.properties` file provided with the Terracotta kit. The default `tc.properties` file should *not* be edited or moved.

If you create a *local* `tc.properties` file in the Terracotta `lib` directory, system properties set in that file are used by Terracotta and will override system properties in the *default* `tc.properties` file. System properties in the local `tc.properties` file are *not* overridden by system properties configured in `tc-config.xml`.

System property values passed to Java using -D override all other configured values for that system property. In the example above, if `-Dcom.tc.l1.cachemanager.leastcount=5` was passed at the command line or through a script, it would override the value in `tc-config.xml` and `tc.properties`. The order of precedence is shown in the following list, with highest precedence shown last:

Override Priority

1. default `tc.properties`
2. `tc-config.xml`
3. local, or user-created `tc.properties` in Terracotta `lib` directory
4. Java system properties set with -D

Failure to Override

If system properties set in `tc-config.xml` fail to override default system properties, a warning is logged to the Terracotta logs. The warning has the following format:

```
The property <system_property_name> was set by local settings to <value>.
This value will not be overridden to <value> from the tc-config.xml file.
```

System properties used early in the Terracotta initialization process may fail to be overridden. If this happens, a warning is logged to the Terracotta logs. The warning has the following format:

```
The property <system_property_name> was read before initialization completed.
```

The warning is followed by the value assigned to `<system_property_name>`.

NOTE: The property `tc.management.mbeans.enabled` is known to load before initialization completes and cannot be overridden.

Servers Configuration Section

This section contains the information that defines and configures the Terracotta Server Array (TSA) and its component servers.

/tc:tc-config/servers

This section defines the Terracotta server instances present in your cluster. One or more entries can be defined, either directly under the <servers> element or in mirror groups. If this section is omitted, Terracotta configuration behaves as if there's a single server instance with default values.

This section also defines certain global settings that affect all servers, including the attribute `secure`. This is a global control for enabling ("true") or disabling ("false" DEFAULT) SSL-based security for the entire cluster.

/tc:tc-config/servers/server

A server stanza encapsulates the configuration for a Terracotta server instance. The server element takes three optional attributes (see table below).

Attribute	Definition	Value	Default Value
host	The address of the machine hosting the Terracotta server	Host machine's IP address or resolvable hostname	Host machine's IP address
name		user-defined string	:

	The symbolic name of the Terracotta server; can be passed to Terracotta scripts such as start-tc-server using `-n <name>`		
bind	The network interface on which the Terracotta server listens cluster traffic; 0.0.0.0 specifies all interfaces	interface's IP address	0.0.0.0

Each Terracotta server instance needs to know which configuration it should use as it starts up. If the server's configured name is the same as the hostname of the host it runs on and no host contains more than one server instance, then configuration is found automatically.

For more information on how to use the Terracotta configuration file with servers, see the Terracotta configuration guide.

Sample configuration snippet:

```
<server>
  <!-- my host is '%i', my name is '%i:tsa-port', my bind is 0.0.0.0 -->
  ...
</server>
<server host="myhostname">
  <!-- my host is 'myhostname', my name is 'myhostname:tsa-port', my bind is 0.0.0.0 -->
  ...
</server>
<server host="myotherhostname" name="server1" bind="192.168.1.27">
  <!-- my host is 'myotherhostname', my name is 'server1', my bind is 192.168.1.27 -->
  ...
</server>
```

/tc:tc-config/servers/server/data

This element specifies the path where the server should store its data for persistence.

Default: data (creates the directory `data` under the working directory)

/tc:tc-config/servers/server/logs

This section lets you declare where the server should write its logs.

Default: logs (creates the directory `logs` under the working directory)

You can also specify `stderr:` or `stdout:` as the output destination for log messages. For example:

```
<logs>stdout:</logs>
```

To set the logging level, see this FAQ entry.

/tc:tc-config/servers/server/index

This element specifies the path where the server should store its search indexes.

Default: index (creates the directory `index` under the working directory)

/tc:tc-config/servers/server/data-backup

This element specifies the path where the server should store backups (if a backup call is initiated).

Default: data-backup (creates the directory `data-backup` under the working directory)

/tc:tc-config/servers/server/tsa-port

This section lets you set the port that the Terracotta server listens to for client traffic.

The default value of "tsa-port" is 9510.

Sample configuration snippet:

```
<tsa-port>9510</tsa-port>
```

/tc:tc-config/servers/server/jmx-port

This section lets you set the port that the Terracotta server's JMX Connector listens to.

The default value of "jmx-port" is 9520. If tsa-port is set, this port defaults to the value of the tsa-port plus 10.

Sample configuration snippet:

```
<jmx-port>9520</jmx-port>
```

/tc:tc-config/servers/server/tsa-group-port

This section lets you set the port that the Terracotta server uses to communicate with other Terracotta servers.

The default value of "tsa-group-port" is 9530. If tsa-port is set, this port defaults to the value of the tsa-port plus 20.

Sample configuration snippet:

```
<tsa-group-port>9530</tsa-group-port>
```

/tc:tc-config/servers/server/security

This section contains the data necessary for running a secure cluster based on SSL, digital certificates, and node authentication and authorization.

See the advanced-security page for a configuration example.

/tc:tc-config/servers/server/security/ssl/certificate

The element specifying certificate entry and location of the certificate store. The format is:

```
<store-type>:<certificate-alias>@</path/to/keystore.file>
```

The Java Keystore (JKS) type is supported by Terracotta 3.7 and higher.

/tc:tc-config/servers/server/security/keychain

This element contains the following subelements:

- `<class>` – Element specifying the class defining the keychain file. If a class is not specified, `com.terracotta.management.keychain.FileStoreKeyChain` is used.
- `<url>` – The URI for the keychain file. It is passed to the keychain class to specify the keychain file.
- `<secret-provider>` – The fully qualified class name of the user implementation of `com.terracotta.management.security.SecretProviderBackEnd`. This class can read and provide the keychain file.

/tc:tc-config/servers/server/security/auth

This element contains the following subelements:

- `<realm>` – Element specifying the class defining the security realm. If a class is not specified, `com.tc.net.core.security.ShiroIniRealm` is used.
- `<url>` – The URI for the Realm configuration (.ini) file. It is passed to the realm class to specify authentication file. Alternatively, URIs for LDAP or Microsoft Active directory can also be used if one of these schemes is implemented instead.
- `<user>` – The username that represents the server and is authenticated by other servers. This name is part of the credentials stored in the .ini file. The default value is "terracotta".

/tc:tc-config/servers/server/security/management

This element contains the subelements needed to allow the Terracotta Management Server (TMS) to make a secure connection to the TSA:

- `ia` – The HTTPS URL with the domain of the TMS, followed by the port 9443 and the path `/tmc/api/assertIdentity`.
- `timeout` – The timeout value (in milliseconds) for connections from the server to the TMS.
- `hostname` – Used only if the DNS hostname of the server does not match server hostname used in its certificate. If there is a mismatch, enter the DNS address of the server here.

/tc:tc-config/servers/server/authentication

Turn on JMX authentication for the Terracotta server. An empty tag (`<authentication />`) defaults to the standard Java JMX authentication mechanism referring to password and access files in `$JAVA_HOME/jre/lib/management`:

```
$JAVA_HOME/jre/lib/management/jmxremote.password
$JAVA_HOME/jre/lib/management/jmxremote.access
```

You must modify these files as as follows (or, if none exist create them).

jmxremote.password Add a line to the end of the file declaring a username and password followed by a carriage return:

```
secretusername secretpassword
```

jmxremote.access Add the following line (with a carriage return) to the end of your file:

```
secretusername      readwrite
```

Be sure to assign the appropriate permissions to the file. For example, in *NIX:

```
$ chmod 500 jmxremote.password
$ chown <user who will run the server> jmxremote.password
```

For information on alternatives to JMX authentication, see Terracotta Cluster Security.

/tc:tc-config/servers/server/http-authentication/user-realm-file

Turn on authentication for the embedded Terracotta HTTP Server. This requires a properties file that contains the users and passwords that have access to the HTTP server.

The format of the properties file is:

```
username: password [,rolename ...]
```

The supported roles and protected sections are: * `statistics` (for the statistics gatherer at `/statistics-gatherer`.)

Passwords may be clear text, obfuscated or checksummed. The class `com.mortbay.Util.Password` should be used to generate obfuscated passwords or password checksums.

By default, HTTP authentication is turned off.

Sample configuration snippet:

```
<http-authentication>
  <user-realm-file>/opt/terracotta/realm.properties</user-realm-file>
</http-authentication>
```

/tc:tc-config/servers/server/offheap

The off-heap mechanism must be enabled for each server. Use this configuration block:

```
<offheap>
  <enabled>true</enabled>
  <maxDataSize>512m</maxDataSize>
</offheap>
```

The minimum setting for <maxDataSize> is 512MB. It should be set to the maximum amount of memory that the server can make use of, but not the total physical memory available to the server.

/tc:tc-config/servers/mirror-group

A mirror group is a *stripe* in a TSA, consisting of one active server and one or more mirror (or backup) servers. A configuration that does not use the <mirror-group> element would produce a one-stripe TSA:

```
<servers>
```

```
  <server name="A">
  ...
  </server>
  <server name="B">
  ...
  </server>
  <server name="C">
  ...
  </server>
  <server name="D">
  ...
  </server>
...
</servers>
```

One of the named servers would assume the role of active (the one started first or that wins the election), while the remaining servers become mirrors. Note that in a typical stripe, having only one or two mirrors is sufficient and less taxing on the active server's resources (as it needs to sync with each mirror).

The following example shows the same servers split into two stripes:

```
<servers>
  <mirror-group group-name="team1">
    <server name="A">
    ...
    </server>
    <server name="B">
    ...
    </server>
  </mirror-group>
  <mirror-group group-name="team2">
    <server name="C">
    ...
    </server>
    <server name="D">
    ...
    </server>
  </mirror-group>
...
</servers>
```

Each stripe will have one active and one mirror server.

NOTE: Server vs. Mirror Group

Under <servers>, you may use either <server> or <mirror-group> configurations, but not both. All <server> configurations directly under <servers> work together as one mirror group, with one active server and the rest mirrors. To create more than one stripe, use <mirror-group> configurations directly under <servers>. The mirror group configurations then include one or more <server> configurations.
For more examples and information, see the Terracotta Configuration Guide.

/tc:tc-config/servers/garbage-collection

This section lets you configure the periodic distributed garbage collector (DGC) that runs in the TSA. The DGC collects shared data made garbage by Java garbage collection.

For many use cases, there is no need to enable periodic DGC. For caches, the more efficient automatic inline DGC is normally sufficient for clearing garbage. In addition, certain read-heavy applications will never require the periodic DGC as little shared data becomes garbage.

However, in certain situations such as when Terracotta Toolkit data structures are in use, the periodic DGC may need to be enabled. Inline DGC is not available for these data structures.

For more on how DGC functions, see TSA Architecture.

Configuration snippet:

```
<garbage-collection>

 <!--    Default: false -->
  <enabled>true</enabled>

  <!-- If "true", additional information is logged when a
     server performs distributed garbage collection.

     Default: false
  -->
  <verbose>false</verbose>

  <!-- How often should distributed garbage collection
     be performed, in seconds?

     Default: 3600 (60 minutes)
  -->
  <interval>3600</interval>
</garbage-collection>
```

/tc:tc-config/servers/restartable

The fast-restart persistence mechanism must be explicitly enabled for the TSA using this element:

```
<restartable enabled="true"/>
```

In case of TSA failure, fast-restart persistence allows the TSA to reload all shared cluster data.

To function, this feature requires <offheap> to be enabled on each server. To make backups of TSA data, the backup feature requires this feature to be enabled.

/tc:tc-config/servers/client-reconnect-window

This section lets you declare the window of time servers will allow disconnected clients to reconnect to the cluster as the same client. Outside of this window, a client can only rejoin as a new client. The value is specified in seconds and the default is 120 seconds.

If adjusting value, note that a too-short reconnection window can lead to unsuccessful reconnections during failure recovery, while a too-long window lowers the efficiency of the cluster since it is paused for the time the window is in effect.

Further reading: For more information on how client and server reconnection is executed in a Terracotta cluster, and on tuning reconnection properties in a high-availability environment, see Configuring Terracotta

Clusters For High Availability.

Clients Configuration Section

The clients section contains configuration about how clients should behave.

/tc:tc-config/clients/logs

This section lets you configure where the Terracotta client writes its logs.

Sample configuration snippet:

```
<!--
   This value undergoes parameter substitution before being used;
   thus, a value like 'client-logs-%h' would expand to
   'client-logs-banana' if running on host 'banana'. See the
   Product Guide for more details.

   If this is a relative path, then it is interpreted relative to
   the current working directory of the client (that is, the directory
   you were in when you started the program that uses Terracotta
   services). It is thus recommended that you specify an absolute
   path here.


   Default: 'logs-%i'; this places the logs in a directory relative
   to the directory you were in when you invoked the program that uses
   Terracotta services (your client), and calls that directory, for example,
   'logs-10.0.0.57' if the machine that the client is on has assigned IP
   address 10.0.0.57.
-->
<logs>logs-%i</logs>
```

You can also specify `stderr:` or `stdout:` as the output destination for log messages. For example:

```
<logs>stdout:</logs>
```

To set the logging level, see this FAQ entry.

Quartz Scheduler 2.2

Quartz Scheduler is a full-featured job scheduling service for Java applications. The TerracottaJobStore for Quartz Scheduler clusters that service. Clustering Quartz Scheduler using TerracottaJobStore and running on the Terracotta Server Array provides a number of advantages:

- Adds High-Availability – Use "hot" standbys to immediately replace failed servers with no downtime, no lost data.
- Includes a Locality API – Route jobs to specific node groups or base routing decisions on system characteristics and available resources.
- Improves Performance – Offload traditional databases and automatically distribute load.
- Provides a Clear Scale-Out Path – Add capacity without requiring additional database resources.
- Ensures Persistence – Automatically back up shared data without impacting cluster performance.

To install the TerracottaJobStore for Quartz Scheduler, see Clustering Quartz Scheduler.

Clustering Quartz Scheduler

This document shows you how to add Terracotta clustering to an application that is using Quartz Scheduler. Use this installation if you have been running your application:

- on a single JVM, or
- on a cluster using JDBC-Jobstore.

To set up the cluster with Terracotta, you will add a Terracotta JAR to each application and run a Terracotta server array. Except as noted below, you can continue to use Quartz in your application as specified in the Quartz documentation.

To add Terracotta clustering to an application that is using Quartz, follow these steps:

Step 1: Requirements

- JDK 1.6 or higher.
- BigMemory Max 4.0.2 or higher Download the kit and run the installer on the machine that will host the Terracotta server.
- All clustered Quartz objects must be serializable. If you create Quartz objects such as Trigger types, they must be serializable.

Step 2: Install Quartz Scheduler

For guaranteed compatibility, use the JAR files included with the Terracotta kit you are installing. Mixing with older components may cause errors or unexpected behavior.

To install the Quartz Scheduler in your application, add the following JAR files to your application's classpath:

- `${TERRACOTTA_HOME}/quartz/quartz-ee-<quartz-version>.jar`

  <quartz-version> is the current version of Quartz (2.2.0 or higher).
- `${TERRACOTTA_HOME}/common/terracotta-toolkit-runtime-ee-<version>.jar`

  The Terracotta Toolkit JAR contains the Terracotta client libraries. <version> is the current version of the Terracotta Toolkit JAR (4.0.2 or higher).

If you are using a WAR file, add these JAR files to its `WEB-INF/lib` directory.

NOTE: Application Servers

Most application servers (or web containers) should work with this installation of the Quartz Scheduler. However, note the following:
- GlassFish application server – You must add `<jvm-options>-Dcom.sun.enterprise.server.ss.ASQuickStartup=false</jvm-options>` to `domains.xml`.
- WebLogic application server – You must use the supported version of WebLogic.

Step 3: Configure Quartz Scheduler

The Quartz configuration file, `quartz.properties` by default, should be on your application's classpath. If you are using a WAR file, add the Quartz configuration file to `WEB-INF/classes` or to a JAR file that is included in `WEB-INF/lib`.

Add Terracotta Configuration

To be clustered by Terracotta, the following properties in `quartz.properties` must be set as follows:

```
# If you use the jobStore class TerracottaJobStore,
# Quartz Where will not be available.
org.quartz.jobStore.class = org.terracotta.quartz.EnterpriseTerracottaJobStore
org.quartz.jobStore.tcConfigUrl = <path/to/Terracotta/configuration>
```

The property `org.quartz.jobStore.tcConfigUrl` must point the client (or application server) at the location of the Terracotta configuration.

TIP: Terracotta Clients and Servers

In a Terracotta cluster, the application server is also known as the client.
The client must load the configuration from a file or a Terracotta server. If loading from a server, give the server's hostname and its tsa-port (9510 by default), found in the Terracotta configuration. The following example shows a configuration that is loaded from the Terracotta server on the local host:

```
# If you use the jobStore class TerracottaJobStore,
# Quartz Where will not be available.
org.quartz.jobStore.class = org.terracotta.quartz.EnterpriseTerracottaJobStore
org.quartz.jobStore.tcConfigUrl = localhost:9510
```

To load Terracotta configuration from a Terracotta configuration file (`tc-config.xml` by default), use a path. For example, if the Terracotta configuration file is located on `myHost.myNet.net` at `/usr/local/TerracottaHome`, use the full URI along with the configuration file's name:

```
# If you use the jobStore class TerracottaJobStore,
# Quartz Where will not be available.
org.quartz.jobStore.class = org.terracotta.quartz.EnterpriseTerracottaJobStore
org.quartz.jobStore.tcConfigUrl =
            file://myHost.myNet.net/usr/local/TerracottaHome/tc-config.xml
```

If the Terracotta configuration source changes at a later time, it must be updated in configuration.

Scheduler Instance Name

A Quartz scheduler has a default name configured by the following `quartz.properties` property:

```
org.quartz.scheduler.instanceName = QuartzScheduler
```

Setting this property is not required. However, you can use this property to instantiate and differentiate between two or more instances of the scheduler, each of which then receives a separate store in the Terracotta cluster.

Using different scheduler names allows you to isolate different job stores within the Terracotta cluster (logically unique scheduler instances). Using the same scheduler name allows different scheduler instances to share the same job store in the cluster.

Step 4: Start the Cluster

1. Start the Terracotta server:

   UNIX/Linux

   ```
   [PROMPT] ${TERRACOTTA_HOME}/server/bin/start-tc-server.sh &
   ```

   Microsoft Windows

   ```
   [PROMPT] ${TERRACOTTA_HOME}\server\bin\start-tc-server.bat
   ```
2. Start the application servers.
3. To monitor the servers, start the Terracotta Management Server:

   UNIX/Linux

   ```
   [PROMPT] ${TERRACOTTA_HOME}/tools/management-console/bin/start-tmc.sh &
   ```

   Microsoft Windows

   ```
   [PROMPT] ${TERRACOTTA_HOME}\tools\management-console\bin\start-tmc.bat
   ```

Step 5: Edit the Terracotta Configuration

This step shows you how to run clients and servers on separate machines and add failover (High Availability). You will expand the Terracotta cluster and add High Availability by doing the following:

- Moving the Terracotta server to its own machine
- Creating a cluster with multiple Terracotta servers
- Creating multiple application nodes

These tasks bring your cluster closer to a production architecture.

Procedure:

1. Shut down the Terracotta cluster.

   UNIX/Linux

   ```
   [PROMPT] ${TERRACOTTA_HOME}/server/bin/stop-tc-server.sh
   ```

   Microsoft Windows

   ```
   [PROMPT] ${TERRACOTTA_HOME}\server\bin\stop-tc-server.bat
   ```
2. Create a Terracotta configuration file called `tc-config.xml` with contents similar to the following:

Procedure:

```
<?xml version="1.0" encoding="UTF-8"?>
<con:tc-config xmlns:con="http://www.terracotta.org/config">
  <servers>
    <mirror-group group-name="default-group">
       <!-- Sets where the Terracotta server can be found.
        Replace the value of host with the server's IP address. -->
      <server host="%i" name="server1">
        <offheap>
          <enabled>true</enabled>
          <maxDataSize>512M</maxDataSize>
        </offheap>
        <tsa-port>9510</tsa-port>
        <jmx-port>9520</jmx-port>
        <data>terracotta/data</data>
        <logs>terracotta/logs</logs>
        <data-backup>terracotta/backups</data-backup>
      </server>
       <!-- If using a mirror Terracotta server, also referred to as an
        ACTIVE-PASSIVE configuration, add the second server here. -->
      <server host="%i" name="Server2">
        <offheap>
          <enabled>true</enabled>
          <maxDataSize>512M</maxDataSize>
        </offheap>
        <tsa-port>9511</tsa-port>
        <data>terracotta/data-dos</data>
        <logs>terracotta/logs-dos</logs>
        <data-backup>terracotta/backups-dos</data-backup>
      </server>
    </mirror-group>
    <update-check>
      <enabled>false</enabled>
    </update-check>
    <garbage-collection>
      <enabled>true</enabled>
    </garbage-collection>
    <restartable enabled="true"/>
  </servers>
  <!-- Sets where the generated client logs are saved on clients. -->
  <clients>
    <logs>terracotta/logs</logs>
  </clients>
</con:tc-config>
```

3. Install BigMemory Max on a separate machine for each server you configure in `tc-config.xml`.
4. Copy the `tc-config.xml` to a location accessible to the Terracotta servers.
5. Perform Step 2: Install Quartz Scheduler and Step 3: Configure Quartz Scheduler on each application node you want to run in the cluster. Be sure to install your application and any application servers on each node.
6. Edit the `org.quartz.jobStore.tcConfigUrl` property in `quartz.properties` to list both Terracotta servers: org.quartz.jobStore.tcConfigUrl = <server.1.ip.address>:9510,<server.2.ip.address>:9510
7. Copy `quartz.properties` to each application node and ensure that it is on your application's classpath. If you are using a WAR file, add the Quartz configuration file to `WEB-INF/classes` or to a JAR file that is included in `WEB-INF/lib`.
8. Start the Terracotta server in the following way, replacing "Server1" with the name you gave your server in `tc-config.xml`:

   UNIX/Linux

Procedure:

```
[PROMPT] ${TERRACOTTA_HOME}/server/bin/start-tc-server.sh \
   -f <path/to/tc-config.xml> -n Server1 &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\server\bin\start-tc-server.bat ^
   -f <path\to\tc-config.xml> -n Server1 &
```

If you configured a second server, start that server in the same way on its machine, entering its name after the `-n` flag. The second server to start up becomes the mirror. Any other servers you configured will also start up as mirrors.

9. Start all application servers.
10. Start the Terracotta Management Server and view the cluster.

Quartz Scheduler Where (Locality API)

Introduction

Terracotta Quartz Scheduler Where is an Enterprise feature that allows jobs and triggers to run on specified Terracotta clients instead of randomly chosen ones. Quartz Scheduler Where provides a locality API that has a more readable fluent interface for creating and scheduling jobs and triggers. This locality API, together with configuration, can be used to route jobs to nodes based on defined criteria:

- Specific resources constraints such as free memory.
- Specific system characteristics such as type of operating system.
- A member of a specified group of nodes.

This document shows you how to configure and use the locality API. You should already be familiar with using Quartz Scheduler (see the installation guide and the Quartz Scheduler documentation).

Configuring Quartz Scheduler Where

To configure Quartz Scheduler Where, follow these steps:

1. Edit `quartz.properties` to cluster with Terracotta. See Clustering Quartz Scheduler for more information.
2. If you intend to use node groups, configure an implementation of `org.quartz.spi.InstanceIdGenerator` to generate instance IDs to be used in the locality configuration. See Understanding Generated Node IDs for more information about generating instance IDs.
3. Configure the node and trigger groups in `quartzLocality.properties`. For example:

```
# Set up node groups that can be referenced from application code.
# The values shown are instance IDs:
org.quartz.locality.nodeGroup.slowJobs = node0, node3
org.quartz.locality.nodeGroup.fastJobs = node1, node2
org.quartz.locality.nodeGroup.allNodes = node0, node1, node2, node3

# Set up trigger groups whose triggers fire only on nodes
# in the specified node groups. For example, a trigger in the
# trigger group slowTriggers will fire only on node0 and node3:
org.quartz.locality.nodeGroup.slowJobs.triggerGroups = slowTriggers
org.quartz.locality.nodeGroup.fastJobs.triggerGroups = fastTriggers
```

4. Ensure that `quartzLocality.properties` is on the classpath, the same as `quartz.properties`.

   See Quartz Scheduler Where Code Sample for an example of how to use Quartz Scheduler Where.

Understanding Generated Node IDs

Terracotta clients each run an instance of a clustered Quartz Scheduler scheduler. Every instance of this clustered scheduler must use the same scheduler name, specified in `quartz.properties`. For example:

```
# Name the clustered scheduler.
org.quartz.scheduler.instanceName = myScheduler
```

myScheduler's data is shared across the cluster by each of its instances. However, every instance of myScheduler must also be identified uniquely, and this unique ID is specified in `quartz.properties` by the property `org.quartz.scheduler.instanceId`. This property should have one of the following values:

- `<string>` – A string value that identifies the scheduler instance running on the Terracotta client that loaded the containing `quartz.properties`. Each scheduler instance must have a unique ID value.
- AUTO – Delegates the generation of unique instance IDs to the class specified by the property `org.quartz.scheduler.instanceIdGenerator.class`.

For example, you can set `org.quartz.scheduler.instanceId` to equal "node1" on one node, "node2" on another node, and so on.

If you set `org.quartz.scheduler.instanceId` equal to "AUTO", then you should specify a generator class in `quartz.properties` using the property `org.quartz.scheduler.instanceIdGenerator.class`. This property can have one of the values listed in the following table.

Value	Notes
`org.quartz.simpl.HostnameInstanceIdGenerator`	Returns the hostname as the instance ID
`org.quartz.simpl.SystemPropertyInstanceIdGenerator`	Returns the value of the `org.quartz.scheduler.instanceId` system property. Available with Quartz 2.0 or higher.
`org.quartz.simpl.SimpleInstanceIdGenerator`	Returns an instance ID composed of the local hostname with the current timestamp appended. Ensures a unique name. If you do not specify a generator class, this generator class is used by default. However, this class is not suitable for use with Quartz Scheduler Where because the IDs it generates are not predictable.
Custom	Specify your own implementation of the interface `org.quartz.spi.InstanceIdGenerator`.

Using SystemPropertyInstanceIdGenerator

`org.quartz.simpl.SystemPropertyInstanceIdGenerator` is useful in environments that use initialization scripts or configuration files. For example, you could add the instanceId property to an application server's startup script in the form `-Dorg.quartz.scheduler.instanceId=node1`, where "node1" is the instance ID assigned to the local Quartz Scheduler scheduler. Or it could also be added to a configuration resource such as an XML file that is used to set up your environment.

The instanceId property values configured for each scheduler instance can be used in `quartzLocality.properties` node groups. For example, if you configured instance IDs node1, node2, and node3, you can use these IDs in node groups:

```
org.quartz.locality.nodeGroup.group1 = node1, node2
org.quartz.locality.nodeGroup.allNodes = node1, node2, node3
```

Available Constraints

Quartz Scheduler Where offers the following constraints:

- CPU – Provides methods for constraints based on minimum number of cores, available threads, and maximum amount of CPU load.
- Resident keys – Use a node with a specified Enterprise Ehcache distributed cache that has the best match for the specified keys.
- Memory – Minimum amount of memory available.
- Node group – A node in the specified node group, as defined in quartzLocality.properties.
- OS – A node running the specified operating system.

See the code samples provided below for how to use these constraints.

Quartz Scheduler Where Code Sample

A cluster has Terracotta clients running Quartz Scheduler running on the following hosts: node0, node1, node2, node3. These hostnames are used as the instance IDs for the Quartz Scheduler scheduler instances because the following `quartz.properties` properties are set as shown:

```
org.quartz.scheduler.instanceId = AUTO

#This sets the hostnames as instance IDs:
org.quartz.scheduler.instanceIdGenerator.class =
    org.quartz.simpl.HostnameInstanceIdGenerator
```

quartzLocality.properties has the following configuration:

```
org.quartz.locality.nodeGroup.slowJobs = node0, node3
org.quartz.locality.nodeGroup.fastJobs = node1, node2
org.quartz.locality.nodeGroup.allNodes = node0, node1, node2, node3

org.quartz.locality.nodeGroup.slowJobs.triggerGroups = slowTriggers
org.quartz.locality.nodeGroup.fastJobs.triggerGroups = fastTriggers
```

The following code snippet uses Quartz Scheduler Where to create locality-aware jobs and triggers.

```
// Note the static imports of builder classes that define a Domain Specific Language (DSL).
import static org.quartz.JobBuilder.newJob;
import static org.quartz.TriggerBuilder.newTrigger;
import static org.quartz.locality.LocalityTriggerBuilder.localTrigger;
import static org.quartz.locality.NodeSpecBuilder.node;
import static org.quartz.locality.constraint.NodeGroupConstraint.partOfNodeGroup;

import org.quartz.JobDetail;
import org.quartz.locality.LocalityTrigger;
// Other required imports...

// Using the Quartz Scheduler fluent interface, or the DSL.

/***** Node Group + OS Constraint
Create a locality-aware job that can be run on any node
from nodeGroup "group1" that runs a Linux OS:
*****/
```

Quartz Scheduler Where Code Sample

```
LocalityJobDetail jobDetail1 =
        localJob(
            newJob(myJob1.class)
                .withIdentity("myJob1")
                .storeDurably(true)
                .build())
            .where(
                node()
                    .is(partOfNodeGroup("group1"))
                    .is(OsConstraint.LINUX))
            .build();

// Create a trigger for myJob1:
Trigger trigger1 = newTrigger()
            .forJob("myJob1")
            .withIdentity("myTrigger1")
            .withSchedule(simpleSchedule()
                   .withIntervalInSeconds(10)
                   .withRepeatCount(2))
            .build();

// Create a second job:
JobDetail jobDetail2 = newJob(myJob2.class)
                .withIdentity("myJob2")
                .storeDurably(true)
                .build();

/***** Memory Constraint
Create a locality-aware trigger for myJob2 that will fire on any
node that has a certain amount of free memory available:
*****/
LocalityTrigger trigger2 =
        localTrigger(newTrigger()
            .forJob("myJob2")
            .withIdentity("myTrigger2"))
            .where(
                node()
                    // fire on any node in allNodes
        //  with at least 100MB in free memory.
                    .is(partOfNodeGroup("allNodes"))
                    .has(atLeastAvailable(100, MemoryConstraint.Unit.MB)))
            .build();

/***** A Locality-Aware Trigger For an Existing Job
 The following trigger will fire myJob1 on any node in the allNodes group
 that's running Linux:
*****/

LocalityTrigger trigger3 =
        localTrigger(newTrigger()
            .forJob("myJob1")
            .withIdentity("myTrigger3"))
            .where(
                node()
                    .is(partOfNodeGroup("allNodes")))
            .build();


/***** Locality Constraint Based on Cache Keys

The following job detail sets up a job (cacheJob) that will be fired on the node
where myCache has, locally, the most keys specified in the collection myKeys.
```

```
After the best match is found, missing elements will be faulted in.
If these types of jobs are fired frequently and a large amount of data must often be
faulted in, performance could degrade. To maintain performance, ensure that most of
the targeted data is already cached.

*****/

// myCache is already configured, populated, and distributed.
Cache myCache = cacheManager.getEhcache("myCache");

// A Collection is needed to hold the keys for the elements to be targeted by cacheJob.
// The following assumes String keys.

Set<String> myKeys = new HashSet<String>();

... // Populate myKeys with the keys for the target elements in myCache.

// Create the job that will do work on the target elements.

LocalityJobDetail cacheJobDetail =
        localJob(
            newJob(cacheJob.class)
                .withIdentity("cacheJob")
                .storeDurably(true)
                .build())
                .where(
                    node()
                         .has(elements(myCache, myKeys)))
                .build();
```

Notice that trigger3, the third trigger defined, overrode the partOfNodeGroup constraint of myJob1. Where triggers and jobs have conflicting constraints, the triggers take priority. However, since trigger3 did not provide an OS constraint, it did *not* override the OS constraint in myJob1. If any of the constraints in effect — trigger or job — are not met, the trigger will go into an error state and the job will not be fired.

CPU-Based Constraints

The CPU constraint allows you to run jobs on machines with adequate processing power:

```
...

import static org.quartz.locality.constraint.CpuConstraint.loadAtMost;

...

// Create a locality-aware trigger for someJob.
LocalityTrigger trigger =
        localTrigger(newTrigger()
            .forJob("someJob")
            .withIdentity("someTrigger"))
            .where(
                node()
                    // fire on any node in allNodes
                    // with at most the specified load:
                    .is(partOfNodeGroup("allNodes"))
                    .has(loadAtMost(.80)))
            .build();
```

The load constraint refers to the CPU load (a standard *NIX load measurement) averaged over the last minute. A load average below 1.00 indicates that the CPU is likely to execute the job immediately. The smaller the load, the freer the CPU, though setting a threshold that is too low could make it difficult for a match to be found.

Other CPU constraints include `CpuContraint.coresAtLeast(int amount)`, which specifies a node with a minimum number of CPU cores, and `CpuConstraint.threadsAvailableAtLeast(int amount)`, which specifies a node with a minimum number of available threads.

NOTE: Unmet Constraints Cause Errors

If a trigger cannot fire because it has constraints that cannot be met by any node, that trigger will go into an error state. Applications using Quartz Scheduler Where with constraints should be tested under conditions that simulate those constraints in the cluster.

This example showed how memory and node-group constraints are used to route locality-aware triggers and jobs. trigger2, for example, is set to fire myJob2 on a node in a specific group ("allNodes") with a specified minimum amount of free memory. A constraint based on operating system (Linux, Microsoft Windows, Apple OSX, and Oracle Solaris) is also available.

Failure Scenarios

If a trigger cannot fire on the specified node or targeted node group, the associated job will not execute. Once the misfireThreshold timeout value is reached, the trigger misfires and any misfire instructions are executed.

Locality With the Standard Quartz Scheduler API

It is also possible to add locality to jobs and triggers created with the standard Quartz Scheduler API by assigning the triggers to a trigger group specified in `quartzLocality.properties`.

Quartz Scheduler Reference

This section contains information on functional aspects of Terracotta Quartz Scheduler and optimizing your use of TerracottaJobstore for Quartz Scheduler.

Execution of Jobs

In the general case, exactly one Quartz Scheduler node, or Terracotta client, executes a clustered job when that job's trigger fires. This can be any of the nodes that have the job. If a job repeats, it may be executed by any of the nodes that have it exactly once per the interval configured. It is not possible to predict which node will execute the job.

With Quartz Scheduler Where, a job can be assigned to a specific node based on certain criteria.

Working With JobDataMaps

JobDataMaps contain data that may be useful to jobs at execution time. A JobDataMap is stored at the time its associated job is added to a scheduler.

Updating a JobDataMap

If the stored job is stateful (implements the StatefulJob interface), and the contents of its JobDataMap is updated (from within the job) during execution, then a new copy of the JobDataMap is stored when the job completes.

If the job is not stateful, then it must be explicitly stored again with the changed JobDataMap to update the stored copy of the job's JobDataMap. This is because TerracottaJobStore contains deep copies of JobDataMap objects and does not reflect updates made after a JobDataMap is stored.

Best Practices for Storing Objects in a JobDataMap

Because TerracottaJobStore contains deep copies of JobDataMap objects, application code should not have references to mutable JobDataMap objects. If an application does rely on these references, there is risk of getting stale data as the mutable objects in a deep copy do not reflect changes made to the JobDataMap after it is stored.

To maximize performance and ensure long-term compatibility, place only Strings and primitives in JobDataMap. JobDataMap objects are serialized and prone to class-versioning issues. Putting complex objects into a clustered JobDataMap could also introduce other errors that are avoided with Strings and primitives.

Cluster Data Safety

By default, Terracotta clients (application servers) do not block to wait for a "transaction received" acknowledgement from a Terracotta server when writing data transactions to the cluster. This asynchronous write mode translates into better performance in a Terracotta cluster.

However, the option to maximize data safety by requiring an acknowledgement is available using the following Quartz configuration property:

```
org.quartz.jobStore.synchronousWrite = true
```

When synchronousWrite is set to "true", a client blocks with each transaction written to the cluster until an acknowledgement is received from a Terracotta server. This ensures that the transaction is committed in the cluster before the client continues work.

Effective Scaling Strategies

Clustering Quartz schedulers is an effective approach to distributing load over a number of nodes if jobs are long-running or are CPU intensive (or both). Distributing the jobs lessens the burden on system resources. In this case, and with a small set of jobs, lock contention is usually infrequent.

However, using a single scheduler forces the use of a cluster-wide lock, a pattern that degrades performance as you add more clients. The cost of this cluster-wide lock becomes more evident if a large number of short-lived jobs are being fired by a single scheduler. In this case, consider partitioning the set of jobs across more than one scheduler.

If you do employ multiple schedulers, they can be run on every node, striping the cluster-wide locks. This is an effective way to reduce lock contention while adding scale.

If you intend to scale, measure your cluster's throughput in a test environment to discover the optimal number of schedulers and nodes.

Terracotta Web Sessions Tutorial

Web Sessions version 4.0

Web Sessions 4.0 works with the latest version of the Terracotta Server Array, which backs your sessions with BigMemory resource management, nonstop, rejoin, and many other features. For more information, refer to the Terracotta Server Array page.

Follow these steps to get a sample clustered web sessions application running with Terracotta on your machine.

{toc-zonel3:3}

Step 1: Download Terracotta Web Sessions

Download the BigMemory Max kit, which includes Terracotta Web Sessions.

Unpack the Kit

Unpack the distribution into a directory on your system. In the following instructions, we will refer to the directory as <terracotta>/. Where forward slashes ("/") are given in directory paths, substitute back slashes ("\") for Microsoft Windows installations.

Copy the License Key

Copy the license key to the terracotta distribution directory. This file, called `terracotta-license.key`, was attached to an email you received after registering for the download.

```
%> cp terracotta-license.key <terracotta>/
```

Step 2: Start the Shopping Cart Tutorial

We'll use the "Cart" sample in the sessions samples directory of the Terracotta distribution to demonstrate clustered Web Sessions.

Start the Terracotta Server

```
%> cd <terracotta>/sessions/code-samples/cart

%> bin/start-sample-server.sh
```

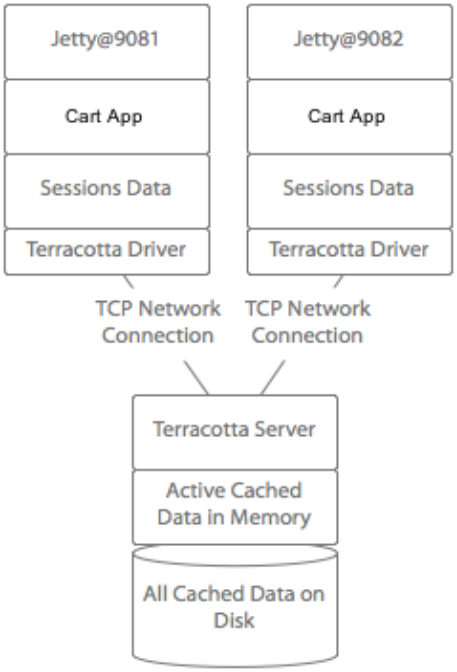
Start the Sample

```
%> bin/start-sample.sh
```

Step 3: View the Sample

The Shopping Cart sample starts up two instances of Jetty, each connected to the Terracotta server for access to shared session data. The sample application uses Terracotta to store session data for scalable high availability. Any session can be read from any application server.

Step 3: View the Sample



View the Cart

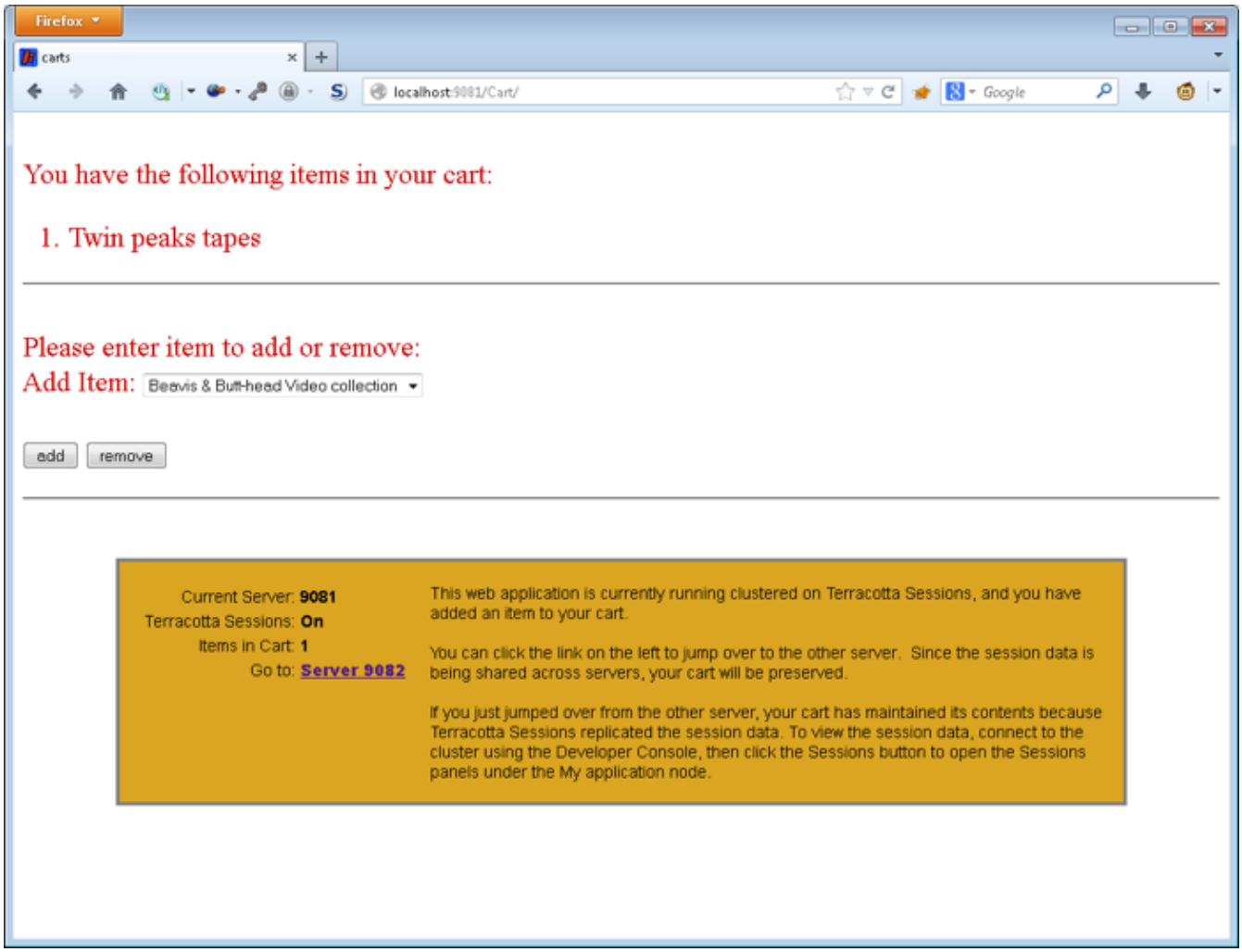
Once the Jetty instances have started, you can view the sample application by visiting the following links:

http://localhost:9081/Cart ›
http://localhost:9082/Cart ›

Note: These links *will not work* unless you have the sample running.

Create Session Data

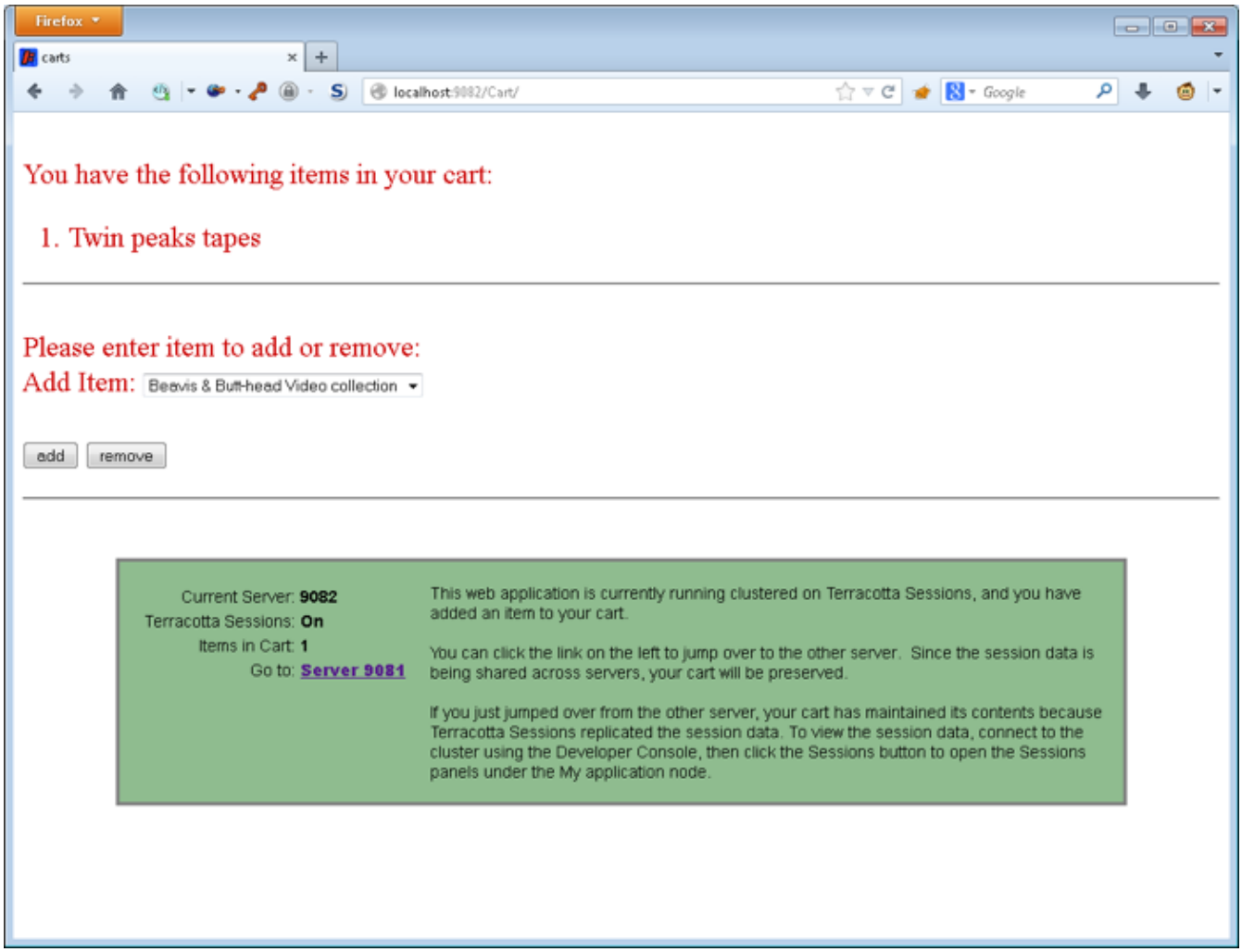
After Jetty loads the sample application, select an item to place in the cart. It should look something like this in your browser:



Notice that your item is stored in the session and displayed on the page.

Step 4: View the Session in Other JVMs

To see the session data automatically made consistent and available in the other application server, click the Server link in the colored box.



Notice that your product browsing history is intact, even on the other application server. Web Sessions is automatically and transparently clustering your session data and sharing it between the two server instances on demand. Make another change to your cart and then click the other server link. You will observe that the session data is shared across the two server instances at a fine-grained, field-change level, independent of the application code.

Step 5: Install Web Sessions with Your Application

Go to the Web Sessions Installation Guide to learn how to install Web Sessions with your application, configure Terracotta clustering, and view runtime statistics.

{/toc-zone}

Web Sessions Installation Guide

Step 1: Requirements

- JDK 1.6 or higher.
- An application server.
- All clustered objects must be serializable.
- Download and unpack the BigMemory Max kit, which includes Terracotta Web Sessions.

Step 2: Install the Terracotta Sessions JAR

To cluster your application's web sessions, add the following two JAR files to your application's classpath, and place these files in the `WEB-INF/lib` folder of the Web Application Archive (WAR) on all of your application servers. These files are under ${TERRACOTTA_HOME}/.

- `sessions/lib/web-sessions-<version>.jar`

  <version> is the current version of the Terracotta Web Sessions JAR.
- `apis/toolkit/lib/terracotta-toolkit-runtime-ee-<version>.jar`

  The Terracotta Toolkit JAR contains the Terracotta client libraries. <version> is the current version of the Terracotta Toolkit JAR.

Step 3: Configure Web-Session Clustering

Terracotta servers, and Terracotta clients running on the application servers in the cluster, are configured with a Terracotta configuration file, `tc-config.xml` by default. Servers that are not started with a specified configuration will use a default configuration.

To add Terracotta clustering to your application, you must specify how Terracotta clients get their configuration by including the source in your `web.xml` file. Add the following configuration snippet to `web.xml`:

```
<filter>
 <filter-name>terracotta</filter-name>
 <!-- The filter class is specific to the application server. -->
 <filter-class>org.terracotta.session.{container-specific-class}</filter-class>
 <init-param>
   <param-name>tcConfigUrl</param-name>
   <!-- <init-param> of type tcConfigUrl has a <param-value> element containing
        the URL or filepath (for example, /lib/tc-config.xml) to tc-config.xml.
    If the Terracotta configuration source changes at a later time,
    it must be updated in configuration. -->
   <param-value>localhost:9510</param-value>
 </init-param>
<!-- The following init-params are optional.
 <init-param>
   <param-name>synchronousWrite</param-name>
   <param-value>false</param-value>
 </init-param>
 <init-param>
   <param-name>sessionLocking</param-name>
```

```
    <param-value>false</param-value>
  </init-param>
  <init-param>
    <param-name>maxBytesOnHeap</param-name>
    <param-value>128M</param-value>
  </init-param>
  <init-param>
    <param-name>maxBytesOffHeap</param-name>
    <param-value>4G</param-value>
  </init-param>
  <init-param>
    <param-name>rejoin</param-name>
    <param-value>false</param-value>
  </init-param>
  <init-param>
    <param-name>nonStopTimeout</param-name>
    <param-value>-1</param-value>
  </init-param>
  <init-param>
    <param-name>concurrency</param-name>
    <param-value>0</param-value>
  </init-param>
<!-- End of optional init-params. -->
</filter>
<filter-mapping>
  <!-- Must match filter name from above. -->
  <filter-name>terracotta</filter-name>
  <url-pattern>/*</url-pattern>
  <!-- Enable all available dispatchers. -->
  <dispatcher>ERROR</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

<filter-name> can contain a string of your choice. However, the value of <filter>/<filter-name> must match <filter-mapping>/<filter-name>.

Choose the appropriate value for <filter-class> from the following table.

Container	**Value of <filter-class>**
GlassFish	org.terracotta.session.TerracottaGlassfish31xSessionFilter
JBoss 6.1.x	org.terracotta.session.TerracottaJboss61xSessionFilter
JBoss 7.1.x	org.terracotta.session.TerracottaJboss71xSessionFilter
JBoss 7.2.x	org.terracotta.session.TerracottaJboss72xSessionFilter
Jetty 8.1.x	org.terracotta.session.TerracottaJetty81xSessionFilter
Jetty 9.0.x	org.terracotta.session.TerracottaJetty90xSessionFilter
Resin	org.terracotta.session.TerracottaResin40xSessionFilter
Tomcat 6.0.x	org.terracotta.session.TerracottaTomcat60xSessionFilter
Tomcat 7.0.x	org.terracotta.session.TerracottaTomcat70xSessionFilter
WebLogic 10.3.x	org.terracotta.session.TerracottaWeblogic103xSessionFilter
WebLogic 12.1.x	org.terracotta.session.TerracottaWeblogic121xSessionFilter
WebSphere 7.0.x	org.terracotta.session.TerracottaWebsphere70xSessionFilter
WebSphere 8.0.x	org.terracotta.session.TerracottaWebsphere80xSessionFilter

WebSphere 8.5.x	org.terracotta.session.TerracottaWebsphere85xSessionFilter

Ensure that the Terracotta filter is the first <filter> element listed in `web.xml`. Filters processed ahead of the Terracotta filter may disrupt its processing.

`web.xml` should be in `/WEB-INF` if you are using a WAR file.

For more information about the optional init-params, refer to the Web Sessions Reference Guide.

Step 4: Start the Cluster

For Terracotta to function properly, make sure that your JAVA_HOME setting is set.

1. Start the Terracotta server:

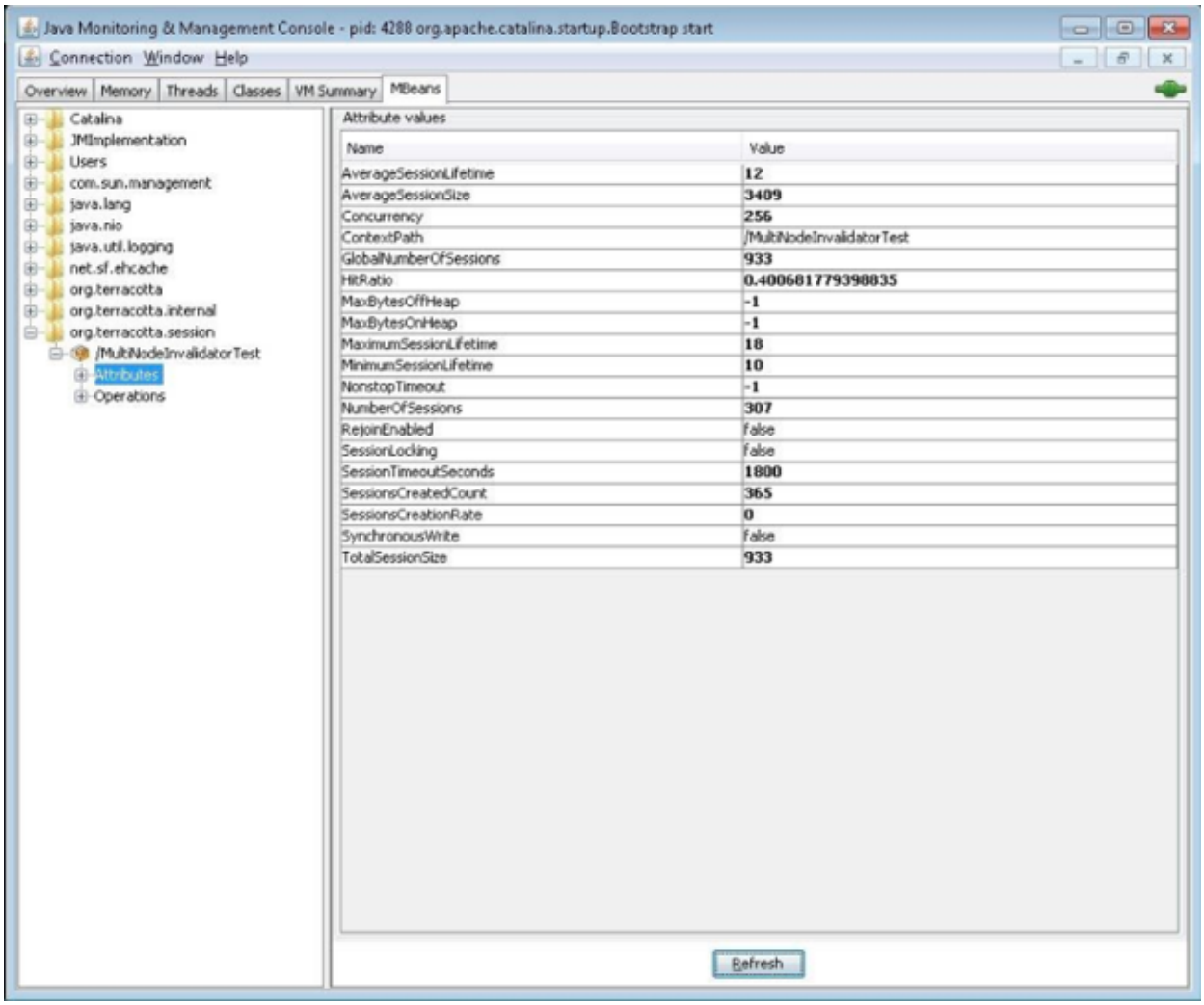
   UNIX/Linux

   ```
   [PROMPT] ${TERRACOTTA_HOME}/server/bin/start-tc-server.sh
   ```

   Microsoft Windows

   ```
   [PROMPT] ${TERRACOTTA_HOME}\server\bin\start-tc-server.bat
   ```
2. Start your application servers.
3. Start your management console (any JMX monitoring tool is fine) and connect it to your application server. For more information about Terracotta and JMX, refer to JMX Management and Monitoring.

   The console below displays the Web Sessions MBeans.



The following table describes the Web Sessions MBeans:

MBean	Data Type	Description
Average Session Lifetime	Integer	Average local session lifetime (in seconds) @return average session lifetime, or -1 if no sessions have expired.
Average Session Size	Long	Average local session size (in bytes).
Concurrency	(Integer)	Number of segments for the map backing the underlying server store managed by the Terracotta Server Array. If concurrency is not explicitly set (or set to "0"), the system selects an optimized value.
Context Path	String	Context path for this servlet context.
Global Number Of Sessions	Long	Total number of live sessions that exist in the cluster.
Hit Ratio	Double	Local hit ratio of the sessions cache @return hit ratio n, or NaN if unavailable.
Max Bytes Off Heap	Long	Maximum number of bytes allowed in the off-heap tier.
Max Bytes On	Long	Maximum number of bytes allowed in the heap tier.

Heap		
Maximum Session Lifetime	Integer	Maximum local session lifetime (in seconds) @return maximum session lifetime, or -1 if no sessions have expired.
Minimum Session Lifetime	Integer	Minimum local session lifetime (in seconds) @return minimum session lifetime, or -1 if no sessions have expired.
Nonstop Timeout	Long	Number of milliseconds an application waits for any cache operation to return before timing out, or -1 if no timeout is configured.
Number Of Sessions	Long	Number of session objects in local memory.
Rejoin Enabled	Boolean	Whether client is configured to automatically rejoin its cluster if ejected.
Session Locking	Boolean	Whether session locking is enabled.
Session Timeout Seconds	Integer	Number of seconds a session waits before timing out, or -1 if no timeout is configured.
Sessions Created Count	Long	Number of sessions created locally since initial startup.
Sessions Creation Rate	Long	Rate of local session creation since initial startup. This number represents the number sessions created in the previous minute.
Synchronous Write	Boolean	Whether configured to synchronously flush enclosed changes to the Terracotta Server Array, blocking the unlocking thread until changes have been acknowledged as committed.
Total Session Size	Long	Total local session size (in bytes).

Step 5: Configure Terracotta Clustering

This step shows you how to run clients and servers on separate machines and add failover (High Availability). You will expand the Terracotta cluster and add High Availability by doing the following:

- Moving the Terracotta server to its own machine
- Creating a cluster with multiple Terracotta servers
- Creating multiple application nodes

These tasks bring your cluster closer to a production architecture.

Procedure:

1. Shut down the Terracotta cluster.
2. Create a Terracotta configuration file called `tc-config.xml` with contents similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- All content copyright Terracotta, Inc., unless otherwise indicated.
 All rights reserved. -->
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-8.xsd"
xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

 <servers>
   <!-- Sets where the Terracotta server can be found.
    Replace the value of host with the server's IP address. -->
   <server host="server.1.ip.address" name="Server1">
     <data>%(user.home)/terracotta/server-data</data>
```

Procedure:

```
      <logs>%(user.home)/terracotta/server-logs</logs>
    </server>
    <!-- If using a standby Terracotta server, also referred to as
     an ACTIVE-PASSIVE configuration, add the second server here. -->
    <server host="server.2.ip.address" name="Server2">
      <data>%(user.home)/terracotta/server-data</data>
      <logs>%(user.home)/terracotta/server-logs</logs>
    </server>
  </servers>
  <!-- Sets where the generated client logs are saved on clients. -->
  <clients>
    <logs>%(user.home)/terracotta/client-logs</logs>
  </clients>
</tc:tc-config>
```

3. Install Terracotta on a separate machine for each server you configure in `tc-config.xml`.
4. Copy the `tc-config.xml` to a location accessible to the Terracotta servers.
5. Perform Step 2: Install the Terracotta Sessions JAR on each application node you want to run in the cluster. Be sure to install your application and any application servers on each node.
6. Edit `web.xml` on each application server to list both Terracotta servers:

   ```
   <param-value>server.1.ip.address:9510,server.2.ip.address:9510</param-value>
   ```
7. Start the Terracotta server in the following way, replacing "Server1" with the name you gave your server in `tc-config.xml`:

   UNIX/Linux

   ```
   [PROMPT] ${TERRACOTTA_HOME}/server/bin/start-tc-server.sh -f <path/to/tc-config.xml> \
      -n Server1
   ```

   Microsoft Windows

   ```
   [PROMPT] ${TERRACOTTA_HOME}\server\bin\start-tc-server.bat -f <path\to\tc-config.xml> ^
      -n Server1
   ```

   If you configured a second server, start that server in the same way on its machine, entering its name after the `-n` flag. The second server to start up becomes the hot standby, or PASSIVE. Any other servers you configured will also start up as standby servers.
8. Start all application servers.
9. Start your JMX console tool and view the cluster.

Step 6: Learn More

To learn more about working with a Terracotta cluster, see the following documents:

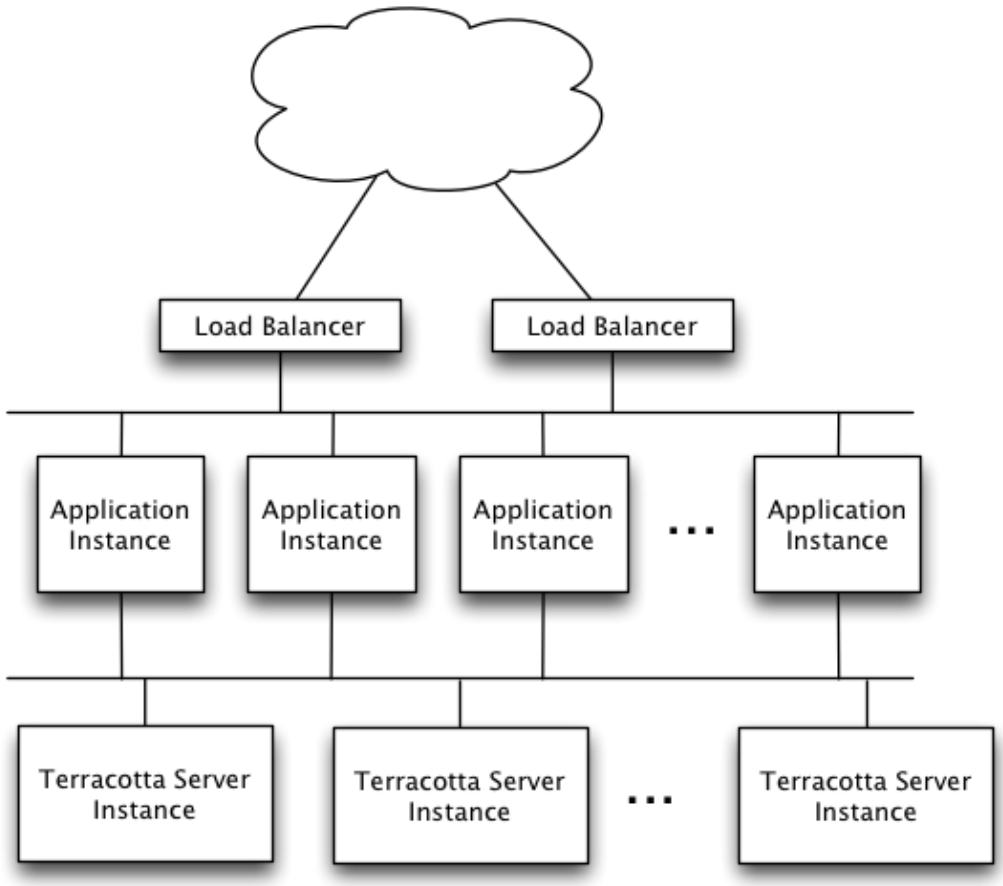
- Working with Terracotta Configuration Files – Explains how `tc-config.xml` is propagated and loaded in a Terracotta cluster in different environments.
- Terracotta Server Arrays – Shows how to design Terracotta clusters that are fault-tolerant, maintain data safety, and provide uninterrupted uptime.
- Configuring Terracotta Clusters For High Availability – Defines High Availability configuration properties and explains how to apply them.

Web Sessions Reference Guide

This page contains further information on configuring and troubleshooting Terracotta Web Sessions.

Architecture of a Terracotta Cluster

The following diagram shows the architecture of a typical Terracotta-enabled web application.



The load balancer parcels out HTTP requests from the Internet to each application server. To maximize the locality of reference of the clustered HTTP session data, the load balancer uses HTTP session affinity so all requests corresponding to the same HTTP session are routed to the same application server. However, with a Terracotta-enabled web application, any application server can process any request. Terracotta Web Sessions clusters the sessions, allowing sessions to survive node hops and failures.

The application servers run both your web application *and* the Terracotta client software, and are called "clients" in a Terracotta cluster. As many application servers may be deployed as needed to handle your site load.

For more information about the Terracotta clusters, refer to the pages in the Terracotta Server Array section.

Optional Configuration Attributes

While Terracotta Web Sessions is designed for optimum performance with the configuration you set at installation, in some cases it may be necessary to use the configuration attributes described in the following sections.

Session Locking

By default, session locking is off in Terracotta Web Sessions. If your application requires disabling concurrent requests in sessions, you can enable session locking.

To enable session locking, add an `<init-param>` block as follows:

```
<filter>
 <filter-name>terracotta-filter</filter-name>
 <filter-class>org.terracotta.session.TerracottaContainerSpecificSessionFilter</filter-class>
 <init-param>
   <param-name>tcConfigUrl</param-name>
   <param-value>localhost:9510</param-value>
 </init-param>
 <init-param>
   <param-name>sessionLocking</param-name>
   <param-value>true</param-value>
 </init-param>
</filter>
```

If you enable session locking, see Deadlocks When Session Locking Is Enabled.

Synchronous Writes

Synchronous write locks provide an extra layer of data protection by having a client node wait until it receives acknowledgement from the Terracotta Server Array that the changes have been committed. The client releases the write lock after receiving the acknowledgement. Note that enabling synchronous write locks *can substantially raise latency rates, thus degrading cluster performance*.

To enable synchronous writes, add an `<init-param>` block as follows:

```
<filter>
 <filter-name>terracotta-filter</filter-name>
 <filter-class>org.terracotta.session.TerracottaContainerSpecificSessionFilter</filter-class>
 <init-param>
   <param-name>tcConfigUrl</param-name>
   <param-value>localhost:9510</param-value>
 </init-param>
 <init-param>
   <param-name>synchronousWrite</param-name>
   <param-value>true</param-value>
 </init-param>
</filter>
```

Sizing Options

Web Sessions gives you the option to configure both heap and off-heap memory tiers.

- Memory store – Heap memory that holds a copy of the hottest subset of data from the off-heap store. Subject to Java garbage collection (GC).
- Off-heap store – Limited in size only by available RAM. Not subject to Java GC. Can store serialized data only. Provides overflow capacity to the memory store. **Note**: If using off-heap, refer to Allocating direct memory in the JVM.

To set the sizing attributes, add one or both `<init-param>` blocks to your web.xml as follows:

```
<filter>
 <filter-name>terracotta-filter</filter-name>
 <filter-class>org.terracotta.session.TerracottaContainerSpecificSessionFilter</filter-class>
 <init-param>
   <param-name>tcConfigUrl</param-name>
   <param-value>localhost:9510</param-value>
 </init-param>
 <init-param>
   <param-name>maxBytesOnHeap</param-name>
   <param-value>128M</param-value>
 </init-param>
 <init-param>
   <param-name>maxBytesOffHeap</param-name>
   <param-value>2G</param-value>
 </init-param>
</filter>
```

Nonstop and Rejoin Options

The nonstop timeout is the number of milliseconds an application waits for any cache operation to return before timing out. Nonstop allows certain operations to proceed on clients that have become disconnected from the cluster. One way clients go into nonstop mode is when they receive a "cluster offline" event. Note that a nonstop cache can go into nonstop mode even if the node is not disconnected, such as when a cache operation is unable to complete within the timeout allotted by the nonstop configuration.

To set the nonstop timeout, add an `<init-param>` block to your web.xml as follows:

```
<filter>
 <filter-name>terracotta-filter</filter-name>
 <filter-class>org.terracotta.session.TerracottaContainerSpecificSessionFilter</filter-class>
 <init-param>
   <param-name>tcConfigUrl</param-name>
   <param-value>localhost:9510</param-value>
 </init-param>
 <init-param>
   <param-name>nonStopTimeout</param-name>
   <param-value>30000</param-value>
 </init-param>
</filter>
```

Tuning Nonstop Timeout

You can tune the timeout value to fit your environment. The following information provides additional guidance for choosing a nonStopTimeout value:

- In an environment with regular network interruptions, consider increasing the timeout value to prevent timeouts for most of the interruptions.

- In an environment where cache operations can be slow to return and data is required to always be in sync, increase timeout value to prevent frequent timeouts. For example, a locking operation may exceed the nonstop timeout while waiting for a lock. This would trigger nonstop mode only because the lock couldn't be acquired in time. Setting the method's timeout to less than the nonstop timeout avoids this problem.
- If a nonstop cache employs bulk loading, be aware that a timeout multiplier may be applied by the bulk-loading method.

Concurrency

The concurrency attribute allows you to set the number of segments for the map backing the underlying server store managed by the Terracotta Server Array. If concurrency is not explicitly set (or set to "0"), the system selects an optimized value.

To configure or tune concurrency, add an `<init-param>` block to your web.xml as follows:

```
<filter>
 <filter-name>terracotta-filter</filter-name>
 <filter-class>org.terracotta.session.TerracottaContainerSpecificSessionFilter</filter-class>
 <init-param>
   <param-name>tcConfigUrl</param-name>
   <param-value>localhost:9510</param-value>
 </init-param>
 <init-param>
   <param-name>concurrency</param-name>
   <param-value>256</param-value>
 </init-param>
</filter>
```

Tuning Concurrency

The server map underlying the Terracotta Server Array contains the data used by clients in the cluster and is segmented to improve performance through added concurrency. Under most circumstances, the concurrency value is optimized by the Terracotta Server Array and does not require tuning.

If an explicit and fixed segmentation value must be set, use the concurrency attribute, making sure to set an appropriate concurrency value. A too-low concurrency value could cause unexpected eviction of elements. A too-high concurrency value may create many empty segments on the Terracotta Server Array (or many segments holding a few or just one element).

The following information provides additional guidance for choosing a concurrency value:

- In general, the concurrency value should be no less than than the number of active servers in the Terracotta Server Array, and optimally at least twice the number of active Terracotta servers.
- With extremely large data sets, a high concurrency value can improve performance by hashing the data into more segments, which reduces lock contention.
- In environments with very few cache elements, set concurrency to a value close to the number of expected elements.

Troubleshooting

The following sections summarize common issues than can be encountered when clustering Web Sessions.

Sessions Time Out Unexpectedly

Sessions that are set to expire after a certain time instead seem to expire at unexpected times, and sooner than expected. This problem can occur when sessions hop between nodes that do not have the same system time. A node that receives a request for a session that originated on a different node still checks local time to validate the session, not the time on the original node. Adding the Network Time Protocol (NTP) to all nodes can help avoid system-time drift. However, note that having nodes set to different time zones can cause this problem, even with NTP.

This problem can also cause sessions to time out later than expected, although this variation can have many other causes.

Changes Not Replicated

Terracotta Web Sessions must run in serialization mode. In serialization mode, sessions are clustered, and your application must follow the standard servlet convention on using `setAttribute()` for mutable objects in replicated sessions.

Deadlocks When Session Locking Is Enabled

In some containers or frameworks, it is possible to see deadlocks when session locking is in effect. This happens when an *external* request is made from inside the locked session to access that same session. This type of request fails because the session is locked.

Events Not Received on Node

Most Servlet spec-defined events will work with Terracotta clustering, but the events are generated on the node where they occur. For example, if a session is created on one node and destroyed on a second node, the event is received on the second node, not on the first node.

Working with Terracotta License Files

A Terracotta license file is required to run enterprise versions of Terracotta products. The name of the file is `terracotta-license.key` and must not be changed. Trial versions of Terracotta enterprise products expire after a trial period. Expiration warnings are issued both to logs and standard output to allow enough time to contact Terracotta for an extension.

Each node using an enterprise version of Terracotta software requires a copy of the license file or configuration that specifies the file's location. By default, the file is provided in the root directory of the Terracotta software kit. To avoid having to explicitly specify the file's location, you can leave it in the kit's root directory.

Or, more generally, ensure that the resource `/terracotta-license.key` is on the same classpath as the Terracotta Toolkit runtime JAR. (The standard Terracotta Toolkit runtime JAR is included with Terracotta kits. See the installation section in the chapter for your Terracotta product for more information on how to install this JAR file). For example, the license file could be placed in `WEB-INF/classes` when using a web application.

Explicitly Specifying the Location of the License File

If the file is in the Terracotta installation directory, you can specify it with:

```
-Dtc.install-root=/path/to/terracotta-install-dir
```

If the file is in a different location, you can specify it with:

```
-Dcom.tc.productkey.path=/path/to/terracotta-license.key
```

Alternatively, the path to the license file can be specified by adding the following to the beginning of the Terracotta configuration file (`tc-config.xml` by default):

```
<tc-properties>
    <property name="productkey.path" value="path/to/terracotta-license.key" />
    <!-- Other tc.properties here. -->
</tc-properties>
```

To refer to a license file that is in a WAR or JAR file, substitute `productkey.resource.path` for `productkey.path`.

Verifying Products and Features

There are a number of ways to verify what products and features are allowed and what limitations are imposed by your product key. The first is by looking at the readable file (`terracotta-license.key`) containing the product key.

Second, at startup Terracotta software logs a message detailing the product key. The message is printed to the log and to standard output. The message should appear similar to the following:

```
2010-11-03 15:56:53,701 INFO - Terracotta license loaded from
/Downloads/terracotta-ee-3.4.0/terracotta-license.key
Capabilities: DCV2, authentication, ehcache, ehcache monitor, ehcache offheap, operator console,
```

```
quartz, roots, server array offheap, server striping, sessions
Date of Issue: 2010-10-16
Edition: FX
Expiration Date: 2011-01-03
License Number: 0000
License Type: Trial
Licensee: Terracotta QA
Max Client Count: 100
Product: Enterprise Suite
ehcache.maxOffHeap: 200G
terracotta.serverArray.maxOffHeap: 200G
```

Terracotta server information panels in the Terracotta Developer Console and Terracotta Operations Center also contain license details.

Working with Apache Maven

Apache Maven users can set up the Terracotta repository for Terracotta artifacts (including Ehcache, Quartz, and other Terracotta projects) using the URL shown:

```
<repository>
    <id>terracotta-repository</id>
    <url>http://www.terracotta.org/download/reflector/releases</url>
    <releases>
        <enabled>true</enabled>
    </releases>
</repository>
```

A complete repository list is given below. Note the following when using Maven:

- The repository URL is not browsable.
- If you intend to work with Terracotta SNAPSHOT projects (usually in trunk), see Working With Terracotta SNAPSHOT Projects.
- You can use the artifact versions in a specific kit when configuring a POM. Artifacts in a specific kit are guaranteed to be compatible.

NOTE: Errors Caused by Outdated Dependencies

Certain frameworks, including Hibernate and certain Spring modules, may have POMs with dependencies on outdated versions of Terracotta products. This can cause older versions of Terracotta products to be installed in your application's classpath ahead of the current versions of those products, resulting in NoClassDefFound, NoSuchMethod, and other errors. At best, your application may run but not perform correctly. Be sure to locate and remove any outdated dependencies before running Maven.

Creating Enterprise Edition Clients

The following example shows the dependencies needed for creating Terracotta 4.0.1 clients. Version numbers can be found in the specific Terracotta kit you are installing. Be sure to update all artifactIds and versions to match those found in your kit.

```
<dependencies>
 <!-- The Terracotta Toolkit is required for running a client. -->
 <dependency>
   <groupId>org.terracotta</groupId>
   <artifactId>terracotta-toolkit-runtime-ee</artifactId>
   <version>4.0.1</version>
 </dependency>

   <!-- The following dependencies are required for using Ehcache.
        Dependencies not listed here include the SLF4J API JAR (version 1.6.1) and an SLF4J
    binding JAR of your choice. These JARs specify the required logging framework.
    It also does not include the explicit-locking JAR.-->
 <dependency>
   <groupId>net.sf.ehcache</groupId>
   <artifactId>ehcache-ee</artifactId>
   <version>2.7.1</version>
 </dependency>

<!-- The following dependencies are required for using Terracotta Sessions. -->
 <dependency>
   <groupId>org.terracotta</groupId>
```

```
   <artifactId>web-sessions</artifactId>
   <version>4.0.0</version>
 </dependency>
</dependencies>
```

Using the tc-maven Plugin

The tc-maven plugin can simplify the process of integrating and testing Terracotta products and other assets. The plugin supplies a number of useful tasks, including starting, stopping, and pausing Terracotta servers. To integrate the plugin, add the following to your project's POM (version shown is for Terracotta 3.7.4):

```
<plugin>
 <groupId>org.terracotta.maven.plugins</groupId>
 <artifactId>tc-maven-plugin</artifactId>
 <version>1.9.4</version>
</plugin>
```

The following is an abbreviated list of goals available with the tc-maven plugin:

Goal	Function
tc:help	Print help.
tc:start	Start the Terracotta server.
tc:stop	Stop the Terracotta server.
tc:restart	Restart the Terracotta server.
tc:dev-console	Start the Terracotta Developer Console.
tc:run	Start multiple Terracotta server and client processes.
tc:clean	Clean Terracotta data and logs directories.
tc:terminate	Stop web servers that started with tc:run.

Execute tc:help to print more detailed information on these goals.

Working With Terracotta SNAPSHOT Projects

If you intend to work with Terracotta SNAPSHOT projects (usually in trunk), you must have the following settings.xml file installed:

```
<settings xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
 <profiles>
   <profile>
     <id>terracotta-repositories</id>
     <repositories>
       <repository>
         <id>terracotta-snapshots</id>
         <url>http://www.terracotta.org/download/reflector/snapshots</url>
       </repository>
     </repositories>
     <pluginRepositories>
       <pluginRepository>
         <id>terracotta-snapshots</id>
         <url>http://www.terracotta.org/download/reflector/snapshots</url>
       </pluginRepository>
     </pluginRepositories>
   </profile>
```

```
 </profiles>
 <activeProfiles>
 <activeProfile>terracotta-repositories</activeProfile>
 </activeProfiles>
</settings>
```

Terracotta Repositories

The following contains all of the Terracotta repositories available:

```
<repositories>
 <repository>
   <id>terracotta-snapshots</id>
   <url>http://www.terracotta.org/download/reflector/snapshots</url>
   <releases><enabled>false</enabled></releases>
   <snapshots><enabled>true</enabled></snapshots>
 </repository>
 <repository>
   <id>terracotta-releases</id>
   <url>http://www.terracotta.org/download/reflector/releases</url>
   <releases><enabled>true</enabled></releases>
   <snapshots><enabled>false</enabled></snapshots>
 </repository>
</repositories>
<pluginRepositories>
 <pluginRepository>
   <id>terracotta-snapshots</id>
   <url>http://www.terracotta.org/download/reflector/snapshots</url>
   <releases><enabled>false</enabled></releases>
   <snapshots><enabled>true</enabled></snapshots>
 </pluginRepository>
 <pluginRepository>
   <id>terracotta-releases</id>
   <url>http://www.terracotta.org/download/reflector/releases</url>
   <releases><enabled>true</enabled></releases>
   <snapshots><enabled>false</enabled></snapshots>
 </pluginRepository>
</pluginRepositories>
```