

JBossCache Tutorial

4.2.0

TreeCache and PojoCache Tutorial

Mr. Primary Writer

ISBN: N/A

Publication date:

This book is a TreeCache and JBossCache Tutorial.

JBoss Cache: TreeCache and PojoCache Tutorial

Author	Mr. Primary Writer	<docs@jboss.org>
Editor	Mr. John Editor	<docs@jboss.org>
Translator	Mr. John Translate	
Copyright ©		

Copyright © 2007 by Red Hat, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.

Distribution of the work or derivative of the work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

1801 Varsity Drive
Raleigh, NC 27606-2072
USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park, NC 27709
USA

1. Introduction	1
1. Scope	1
2. Configuration	3
3. Script	5
4. Example POJO	7
5. Demo	9
6. Plain cache	11
7. PojoCache	13
8. PojoCache with Transaction	15
9. CacheLoader examples	17
1. Local cache with CacheLoader	17
10. Replicated cache with shared datastore	19
11. Replicated cache with unshared (local) datastore	21
12. Troubleshooting	23

Introduction

JBossCache is an in-memory replicated (synchronous or asynchronous), transactional, and fine-grained cache. It consists of two subsystems: TreeCache (plain cache) and PojoCache (object-oriented POJO cache). In this tutorial, we will demonstrate the usage of both cache features. For details of the usage and APIs, please refer to the user manuals for [TreeCache](http://labs.jboss.org/portal/jbosscache/docs/index.html) [http://labs.jboss.org/portal/jbosscache/docs/index.html] and [PojoCache](http://labs.jboss.org/portal/jbosscache/docs/index.html) [http://labs.jboss.org/portal/jbosscache/docs/index.html].

1. Scope

- Cache creation and modification
- Replication
- PojoCache
- Transaction

Configuration

First download the standalone TreeCache code from [here](http://labs.jboss.org/portal/jbosscache/download/index.html) [http://labs.jboss.org/portal/jbosscache/download/index.html]. Unzip it, and you will get a root directory (jboss-cache in our example).

The configuration files are located under the etc directory. You can modify the behavior of TreeCache through editing the various configuration files.

- `log4j.xml`. Logging output. You can turn on logging level or change log file directory (default is `/tmp/jbosscache.log`).
- `replSync-service.xml`. Tree cache configuration file (file name is not fixed. You specify the file to be read in `PropertyConfigurator`). The settings are for a replicated, synchronous, and transactional cache. The default `DummyTransactionManager` is used with a transaction isolation level of `REPEATABLE_READ`. For details of the configuration parameters, please refer to the [TreeCache](http://www.jboss.org/products/jbosscache/docs) [http://www.jboss.org/products/jbosscache/docs]. Note that this file is used in the BSH ([BeanShell](http://www.beanshell.org/) [http://www.beanshell.org/], a lightweight Java compatible scripting language) script to configure the cache.
- `jboss-aop.xml`. AOP pointcut and advice definition for the example POJO classes, `Person` and `Address`, respectively. For details of how to put your own class under AOP, please refer to the [PojoCache](http://www.jboss.org/products/jbosscache/docs) [http://www.jboss.org/products/jbosscache/docs]. This file is read in when the process is started.

Script

The script files that are needed (located under install directory) in this tutorial are:

- `build.sh` (or `build.bat` for DOS/Windows). Simple build script that wraps around ant. Users can simply type `sh build.sh` for help. Note from now on, we will only refer to the Unix version with the understanding that there is a corresponding DOS counterpart. The same goes for `runDemoShell` explained next.
- `runDemoShell.sh`. Simple run script that wraps around BeanShell. This is used to operate the replicated cache through interactive command line.
- `plain.bsh`. Java code that instantiate and configure the cache. It also creates an example cache entry.
- `aop.bsh`. Java codes that instantiate and configure the aop cache. In addition, it also sets up the example POJO (plain old Java object) classes (e.g., `Person` and `Address`).
- `aopWithTx.bsh`. Same with `aop.bsh` except it also instantiates a transaction context.

Example POJO

The example POJO classes used for PojoCache demo are: `Person` and `Address`. They are located under `tests/org/jboss/cache/aop` directory. `Person` has attributes of `String` `age`, `Address` `addr`, `List` `languages`, etc. We will demonstrate that once you put the POJO instance in the cache, plain `get/set` POJO methods will be intercepted by the cache.

Here is the snippet of the class definition for `Person` and `Address`.

```
public class Person {
    String name=null;
    int age=0;
    Map hobbies=null;
    Address address=null;
    Set skills;
    List languages;

    public String getName() { return name; }
    public void setName(String name) { this.name=name; }
    ...
}
```

```
public class Address {
    String street=null;
    String city=null;
    int zip=0;

    public String getStreet() { return street; }
    public void setStreet(String street) { this.street=street; }
    ...
}
```


Demo

To run the demo, you will need at least two windows: one to peruse the cache contents (plus non-aop operations) and the other to operate the cache directly. Of course, you can also open more than one GUI window to see the cache replication at work to multiple members. You will also need to run the scripts under jboss-cache installation directory after you unzip the release package (jboss-cache-dist.zip). Due to the limitation of the GUI, please note that:

- For each demo example, it'd be best if you re-start the whole setup.
- While you can modify the cache content on the GUI window and it will show up on the BSH cache content (e.g., through `cache.printDetails()`), this won't work on PojoCache demo. That is, you can only modify the cache content on the BSH window.

The two demo programs to run are:

- On the first window for the GUI, type `sh build.sh` to see the available commands. To run the GUI, type `sh build.sh run.demo`. It will startup a PojoCache GUI. Later on, you can click on a node to view the the contents. Note that you can also add/modify the node contents for non-AOP cache entries. Since the GUI entry only accepts String for now, operation on aop cache from the GUI will not always work (unless it is a String type).
- On the second window for the interactive Java commands, type `sh runShellDemo.sh` to fire off the BeanShell interactive command shell (you can use either `^D` or `^Z` in Windows and Unix to exit afterward). You can then read in the Java code scripts to showcase the cache capabilities (e.g., `plain.bsh`, `aop.bsh`, and `aopWithTx.bsh`). See the following for details.

Plain cache

Once you are in the shell, you can either execute the script to populate the cache, or type it in manually by command line. To run the script, type `sourceRelative("plain.bsh");` under the interactive BSH shell. For this to work, you'll need to have your working directory set to the directory in which `plain.bsh` resides (otherwise give the full pathname of `plain.bsh`). Basically, the script will create cache entries that will be replicated onto the GUI. (You may have to type `show()` into the resulting beanshell window yourself) Here are the snippets for the script:

```
import org.jboss.cache.*;
    show(); // verbose mode for bean shell
    TreeCache tree = new TreeCache();
    PropertyConfigurator config = new PropertyConfigurator();
    // configure tree cache. Needs to be in the classpath
    config.configure(tree, "META-INF/replSync-service.xml");
    tree.startService();
    // kick start tree cache
    tree.put("/a/b/c", "ben", "me");
    // create a cache entry.
    // Node "/a/b/c" will be created if it does not yet exist.
```

You should see in the GUI that a new entry of `/a/b/c` has been created. Click on the node `c` to see the content. You can modify the contents from the GUI as well. To create another node, for example, you can type in the shell:

```
tree.put("/a/b/c/d", "JBoss", "Open Source");
    tree.get("/a/b/c/d", "JBoss");
```


PojoCache

Once you are in the shell, type `sourceRelative("aop.bsh");` to execute the shell script. Basically, `aop.bsh` illustrates the steps to instantiate a cache, configure it, and then create entries under it. Here are the snippets:

```
import org.jboss.cache.PropertyConfigurator;
import org.jboss.cache.aop.PojoCache;
import org.jboss.cache.aop.test.Person;
import org.jboss.cache.aop.test.Address;
show(); // verbose mode for bean shell
PojoCache tree = new PojoCache();
PropertyConfigurator config = new PropertyConfigurator(); //
configure tree cache.
config.configure(tree, "META-INF/replSync-service.xml");
Person joe = new Person(); // instantiate a Person object named joe
joe.setName("Joe Black");
joe.setAge(31);
Address addr = new Address(); // instantiate a Address object named
addr
addr.setCity("Sunnyvale");
addr.setStreet("123 Albert Ave");
addr.setZip(94086); joe.setAddress(addr); // set the address
reference
tree.startService(); // kick start tree cache
tree.putObject("/aop/joe", joe);
// add aop sanctioned object (and sub-objects) into cache.
// since it is aop-sanctioned, use of plain get/set methods will
take care
// of cache contents automatically.
joe.setAge(41);
```

Note the API needed to put the object (and its dependent ones) into cache is `putObject`. Once the second window finishes execution, you should see the first GUI window has been populated with entries of `/aop/joe/address`. Click on each tree node will display different values associated with that node.

Next step to see AOP in action, you can do plain `get/set` methods without ever worrying about put it in the cache. For example, you can do in the shell window `joe.setAge(20);` and see that GUI gets updated with the age field automatically (if not, click away and back will refresh the GUI content). Also to demonstrate the object graph replication, you can modify Joe's address and see the cache will update it automatically. For example, type `addr.setCity("San Jose");` in the interactive shell, you should see in the GUI that the address got modified.

Finally, `PojoCache` also supports `get/set` with parameter type of Collection classes (i.e., `List`, `Map`, and `Set`). For example, type the following in the shell command line:

```
ArrayList lang = new ArrayList();
lang.add("Ensligh");
lang.add("Mandarin");
```

```
joe.setLanguages(lang);
```

PojoCache with Transaction

To see TreeCache transaction at work, you start with the same setup with last section except you load the bsh of `aopWithTx.bsh` instead of `aop.bsh`. The additional snippets are:

```
import org.jboss.cache.PropertyConfigurator;
import org.jboss.cache.aop.PojoCache;
import org.jboss.cache.aop.test.Person;
import org.jboss.cache.aop.test.Address; // Tx imports
import javax.transaction.UserTransaction; import javax.naming.*;
import org.jboss.cache.transaction.DummyTransactionManager;
show(); // verbose mode for bean shell
// Set up transaction manager
DummyTransactionManager.getInstance();
Properties prop = new Properties();
prop.put(Context.INITIAL_CONTEXT_FACTORY,
    "org.jboss.cache.transaction.DummyContextFactory");
UserTransaction tx = (UserTransaction)new
    InitialContext(prop).lookup("UserTransaction");
PojoCache tree = new PojoCache();
PropertyConfigurator config = new PropertyConfigurator();
// configure tree cache.
config.configure(tree, "META-INF/replSync-service.xml");
joe = new Person();
joe.setName("Joe Black");
joe.setAge(31);

Address addr = new Address();
addr.setCity("Sunnyvale");
addr.setStreet("123 Albert Ave");
addr.setZip(94086);
joe.setAddress(addr);

tree.startService(); // kick start tree cache
tree.putObject("/aop/joe", joe); // add aop sanctioned object
// since it is aop-sanctioned, use of plain get/set methods will
take care
of cache contents automatically.
// Also it is transacted
tx.begin();
joe.setAge(41);
joe.getAddress().setZip(95124);
tx.commit();
```

In this example, a default dummy transaction manager is used.

```
tx.begin();
addr.setZip(95131);
tx.rollback();
```


CacheLoader examples

All the examples below are based on the JBossCache standalone distribution. We assume the ZIP file has been unzipped into a directory `jboss-cache`.

1. Local cache with CacheLoader

This demo shows a local PojoCache with a CacheLoader. We will insert a POJO into the cache, and see that the POJO is transparently saved using the CacheLoader.

To run this, you have to modify `jboss-cache/output/etc/META-INF/oodb-service.xml`: change `CacheLoaderConfig` to point to a valid directory (create it if it doesn't yet exist):

```
<attribute name="CacheLoaderConfig">
    location=c:\\tmp\\oodb
</attribute>
```

Then start the beanshell and source `oodb.bsh` into it. Note that `oodb.bsh` already contains code to create and retrieve POJO from the cache. So remember to comment them out if you decide to create the Person instance yourself.

```
bela@laptop /cygdrive/c/jboss-cache
$ ./runShellDemo.sh
BeanShell 1.3.0 - by Pat Niemeyer (pat@pat.net)
bsh % sourceRelative("oodb.bsh");
interceptor chain is:
class org.jboss.cache.interceptors.CallInterceptor
class org.jboss.cache.interceptors.CacheLoaderInterceptor
class org.jboss.cache.interceptors.TransactionInterceptor
<null>
bsh %
```

Next, create an instance of Person, and set its address and other fields:

```
bsh % p=new Person();
      <name=null, age=0, hobbies=, address=null, skills=null,
languages=null>
bsh % p.age=3;
      <3>
bsh % p.name="Michelle";
      <Michelle>
bsh % addr=new Address();
      <street=null, city=null, zip=0>
bsh % addr.city="San Jose";
      <San Jose>
bsh % addr.zip=95124;
      <95124>
bsh % addr.street="1704 Almond Blossom Lane";
```

```
<1704 Almond Blossom Lane>
bsh % p.setAddress(addr);
bsh % tree.putObject("/person/me", p);
bsh % p;
<name=Michelle, age=3, hobbies=, address=street=1704 Almond
Blossom Lane,
      city=San Jose, zip=95124, skills=null, languages=null>
bsh %
```

The `Person` object with all of its fields and subobjects is now saved. Let's kill beanshell and restart it. At this point, because the instance of `Person` we created was given the name "p", we can retrieve it again:

```
belalaptop /cygdrive/c/jboss-cache
$ ./runShellDemo.sh
BeanShell 1.3.0 - by Pat Niemeyer (pat@pat.net)
bsh % sourceRelative("oodb.bsh");
interceptor chain is:
class org.jboss.cache.interceptors.CallInterceptor
class org.jboss.cache.interceptors.CacheLoaderInterceptor
class org.jboss.cache.interceptors.TransactionInterceptor
<null>
bsh % tree;
</>
bsh % p=tree.getObject("/person/me");
<name=Michelle, age=3, hobbies=, address=street=1704 Almond
Blossom Lane,
      city=San Jose, zip=95124, skills=null, languages=null>
bsh % tree;
</p
/address
>
bsh %
```

The interesting thing here is that the cache was initially empty ("/"). Only when we loaded "p", did it get populated (lazy loading). You can see that the values of "p" are loaded from the datastore where they were previously saved.

Replicated cache with shared datastore

The scenario that we'll run in this example is described in the documentation for JBossCache. It consists of 2 separate nodes that replicate their contents between each other. In addition, they both point to the *same* datastore. The configuration is in file

jboss-cache/output/etc/META-INF/replAsyncSharedCacheLoader-service.xml :

```
<!-- Whether or not to fetch state on joining a cluster -->
    <attribute name="FetchStateOnStartup">false</attribute>
  <attribute
name="CacheLoaderClass">org.jboss.cache.loader.FileCacheLoader</attribute>
    <attribute name="CacheLoaderConfig">
      location=c:\\tmp
    </attribute>
    <attribute name="CacheLoaderShared">true</attribute>
    <attribute name="CacheLoaderPreload">/</attribute>
    <attribute
name="CacheLoaderFetchTransientState">false</attribute>
    <attribute
name="CacheLoaderFetchPersistentState">true</attribute>
```

The `FetchStateOnStartup` attribute set to `false` means that a newly started cache will *not* attempt to fetch the state (neither transient nor persistent). Therefore, attributes `CacheLoaderFetchTransientState` and `CacheLoaderFetchPersistentState` will be ignored. `CacheLoaderShared` set to `true` means that both nodes will share the same datastore, which resides in `c:\\tmp` in the example (this assumes that both nodes have access to the same file system). Please make sure that `c:\\tmp` exists, or point the config string to a different existing directory.

This configuration would essentially provide for two `cold` nodes, in the sense that all contents of a new cache is in the datastore, and is lazy-loaded via the `CacheLoader` when accessed. However, this is not true, as `CacheLoaderPreload` points to `/`, which is the root of the entire tree. Therefore, all of the contents of the cache are recursively pre-loaded. This is probably a bad configuration when you have a lot of data in the cache, because *all* of your data will be loaded into the cache.

Note that with a shared datastore, the node that makes a modification is the one who writes it to the store using the `CacheLoader`. This prevents both nodes from writing the same data twice.

We can now start 2 instances by opening two shells and executing the following ANT target:

```
bela@laptop /cygdrive/c/jboss-cache
$ ./build.sh run.demo.async.shared.cacheloader
Buildfile: build.xml

init:
```

```
compile:

run.demo.async.shared.cacheloader:
[java] ** node loaded: /a
[java] ** node loaded: /a/b
[java] ** node loaded: /a/b/c
[java] ** node loaded: /uno
[java] ** node loaded: /uno/due

[java] -----
[java] GMS: address is 192.168.1.184:1357
[java] -----
[java] interceptor chain is:
[java] class org.jboss.cache.interceptors.CallInterceptor
[java] class org.jboss.cache.interceptors.ReplicationInterceptor
[java] class org.jboss.cache.interceptors.CacheLoaderInterceptor
[java] class org.jboss.cache.interceptors.TransactionInterceptor
[java] ** view change: [192.168.1.184:1355|1]
[192.168.1.184:1355,
 192.168.1.184:1357]
[java] ** node modified: /
```

2 GUI instances will appear, showing the tree structure of the cache graphically. Nodes can be added, modified or removed by right-clicking or using the menu. Any modification is replicated between the two nodes. If both nodes are killed, and subsequently one or both nodes are restarted, the state is the same as before shutdown as it was persisted to the shared store via the CacheLoader.

Note that the example above shows the 2 nodes running on the same machine (192.168.1.184) on ports 1355 and 1357.

Replicated cache with unshared (local) datastore

In this example, we'll run 2 nodes again, but this time, instead of sharing the same datastore, each node has its own datastore. The configuration is in file

`jboss-cache/output/etc/META-INF/node{1,2}.xml` . We'll look at `node1.xml`:

```
<attribute
name="CacheLoaderClass">org.jboss.cache.loader.bdbje.BdbjeCacheLoader</attribute>
  <attribute name="CacheLoaderConfig">
    location=c:\\tmp\\node1
  </attribute>
  <attribute name="CacheLoaderShared">false</attribute>
  <attribute name="CacheLoaderPreload"></attribute>
  <attribute
name="CacheLoaderFetchTransientState">false</attribute>
  <attribute
name="CacheLoaderFetchPersistentState">true</attribute>
```

Again, we use the Sleepycat CacheLoader implementation in `CacheLoaderClass` . The `CacheLoaderConfig` points to `c:\\tmp\\node1` . This is the directory in which the Sleepycat DB for node1 will reside. File `node2.xml` has a configuration that points to `c:\\tmp\\node2` , so we have 2 different unshared datastores. Note that, of course, we still have the same filesystem in our case, because we run the 2 nodes on the same machine. In practice those two directories would reside on two different machines, and each machine would run one JBossCache process. Note that the 2 directories have to exist

To create an unshared datastore, we set the `CacheLoaderShared` attribute to `false` .

The example can be run by again opening 2 shells, and running 2 ANT targets (here we show the target for node1):

```
bela@laptop /cygdrive/c/jboss-cache
$ ./build.sh run.demo.unshared.node2
Buildfile: build.xml

init:

compile:

run.demo.unshared.node2:
[java] ** node loaded: /a
[java] ** node loaded: /a/a2
...
```

The `run.demo.unshared.node2` target runs node2, which will have its own store located at

c:\tmp\node2 (shown above). Whenever a change is made on either of the 2 nodes, it is replicated to the other node, and persisted in both local datastores. You can kill and restart a node, or even both nodes, and the data will still be available due to the persistent backend store(s).

Troubleshooting

Here are some tips for troubleshooting, if you encounter problems during this demo.

- Most of the time, the problem will come from cache replication layer, i.e., JGroups package. On the output window, you can see the JGroups membership view. See if it is updated when you run the BSH commands. It should show a view with at least two members. For example, on my window, I see

```
[java] ** view change: [BWANG-HOME:4381|1] [BWANG-HOME:4381, BWANG-HOME:4383]
```

with 2 members: 4381 and 4383. On the other hand, if you don't close the previous running cache instance, the membership view will also include the previous existing ones. This can corrupt the states. So you will have to make sure there is no running TreeCache processes before each demo. If you have problem with this, please consult the [JGroups website](http://www.jgroups.org/javagroupsnew/docs/index.html) [http://www.jgroups.org/javagroupsnew/docs/index.html]

