Getting Started with {brandname} 9.3

The {brandname} community

Table of Contents

This is a guide to getting started with {brandname}. It is meant to be read alongside the more complete User Guide, and as a supplement to the Tutorials that have been made available.

Chapter 1. Introduction

This guide will walk you through downloading, installing and running {brandname} for the first time. It will then introduce to some of the key features of {brandname}.

1.1. Runtimes

{brandname} can be used in a variety of runtimes:

- *Java SE*, started by your application
- an *application server* which provides {brandname} as a service (such as JBoss AS)
- *bundled as a library* in your application, deployed to an application server, and started on by your application (for example, you could use {brandname} with Tomcat or GlassFish)
- inside an OSGi runtime environment (such as Apache Karaf)

1.2. Modes

{brandname} offers four modes of operation, which determine how and where the data is stored:

- *Local* , where entries are stored on the local node only, regardless of whether a cluster has formed. In this mode {brandname} is typically operating as a local cache
- *Invalidation* , where all entries are stored into a cache store (such as a database) only, and invalidated from all nodes. When a node needs the entry it will load it from a cache store. In this mode {brandname} is operating as a distributed cache, backed by a canonical data store such as a database
- *Replication* , where all entries are replicated to all nodes. In this mode {brandname} is typically operating as a data grid or a temporary data store, but doesn't offer an increased heap space
- *Distribution* , where entries are distributed to a subset of the nodes only. In this mode {brandname} is typically operating as a data grid providing an increased heap space
- *Scattered*, which is similar to Distribution mode but is more suitable for write-intensive applications.

Invalidation, Replication and Distribution can all use synchronous or asynchronous communication, Scattered mode is only synchronous.

1.3. Interacting with {brandname}

{brandname} offers two access patterns, both of which are available in any runtime:

- *Embedded* into your application code
- As a *Remote* server accessed by a client (REST, memcached or Hot Rod wire protocols are supported)

This guide will introduce to each of the runtime options, access patterns and modes of operations

by walking you through simple applications for each.

Chapter 2. Downloading and installing {brandname}

To run {brandname}, you'll need

- A Java 1.8 JDK
- Maven 3.2+, if you wish to use the quickstart examples or create a new project using {brandname} archetype
- the {brandname} distribution zip, if you wish to use {brandname} in server mode, or want to use the jars in an ant project

If you already have any of these pieces of software, there is no need to install them again!

2.1. JDK

Choose your Java runtime, and follow their installation instructions. For example, you could choose one of:

- OpenJDK
- Oracle Java SE
- Oracle JRockit

2.2. Maven

Follow the official Maven installation guide if you don't already have Maven 3.2 installed. You can check which version of Maven you have installed (if any) by running mvn --version . If you see a version newer than 3.2, you are ready to go.

You can also deploy the examples using your favorite IDE. We provide instructions for using Eclipse only.

2.3. {brandname}

Finally, download {brandname} from the {brandname} downloads page.

2.3.1. Getting {brandname} from Maven

Add to your pom:

<dependency> <groupId>org.infinispan</groupId> <artifactId>infinispan-embedded</artifactId> <version>8.2.0.Final</version> </dependency>

2.3.2. Installing {brandname} inside Apache Karaf

The {brandname} jars contain the required OSGi manifest headers and can be used inside OSGi runtime environments as OSGi bundles. In addition to them you will need to install the required 3rd party dependencies. You can install them one by one if you wish but to make things easier for you we are providing Apache Karaf "features" files (also called "feature repositories") which define all required dependencies and can be used to install everything in just a few steps.

Installing bundles using "features" requires:

- registering the feature repositories inside Karaf
- installing the features contained in the repositories

You will first need to start the Apache Karaf console:

```
$ cd <APACHE_KARAF_HOME>/bin
$ ./karaf
```

To register a feature repository you need to use the `feature:repo-add` command (or `features:addUrl` if you are using Apache Karaf 2.3.x) and provide its URL (Apache Maven URLs are preferred):

```
karaf@root()> feature:repo-add mvn:org.infinispan/infinispan-
core/${version}/xml/features
```

Replace `${version}` with the actual version you plan to use. You can now get the list of available features using:

```
karaf@root()> feature:list | grep infinispan
infinispan-core               | ${version}   |           | infinispan-core-${version}
|
```

and install them using:

```
karaf@root()> feature:install infinispan-core/${version}
```

In Apache Karaf the commands are `features:list` and `features:install`.

Alternatively you can just pass the `-i` flag to the `feature:repo-add` command which will install all the features defined in that repository:

```
karaf@root()> feature:repo-add -i mvn:org.infinispan/infinispan-
core/${version}/xml/features
```

This should get you started using {brandname} in library mode. To get additional functionality just

install the corresponding features. For example to use the LevelDB cachestore install:

```
karaf@root()> feature:repo-add -i mvn:org.infinispan/infinispan-cachestore-
leveldb/${version}/xml/features
```

The URL for the feature repositories is constructed from the Maven artifact coordinates using the format:

```
mvn:<groupId>/<artifactId>/<version>/xml/features
```

To use {brandname} in client/server mode install the Hot Rod Client feature:

```
karaf@root()> feature:repo-add -i mvn:org.infinispan/infinispan-client-
hotrod/${version}/xml/features
```

Currently feature repositories are available for the following artifacts:

- infinispan-commons
- infinispan-core
- infinispan-cachestore-jdbc
- infinispan-cachestore-jpa
- infinispan-cachestore-leveldb
- infinispan-cachestore-remote
- infinispan-client-hotrod

For more details regarding the commands available inside Apache Karaf please consult its user manual.

2.4. Download the quickstarts

The quickstarts are in GitHub, in https://github.com/infinispan/infinispan-quickstart.

Clone this repository using:

```
$ git clone https://github.com/infinispan/infinispan-quickstart
```

Chapter 3. {brandname} GUI demo

This document walks you through using the {brandname} GUI demo that ships with {brandname}, and assumes that you have downloaded the latest version of {brandname} and unzipped the archive. I will refer to the {brandname} directory created by unzipping the archive as `${INFINISPAN_HOME}`.

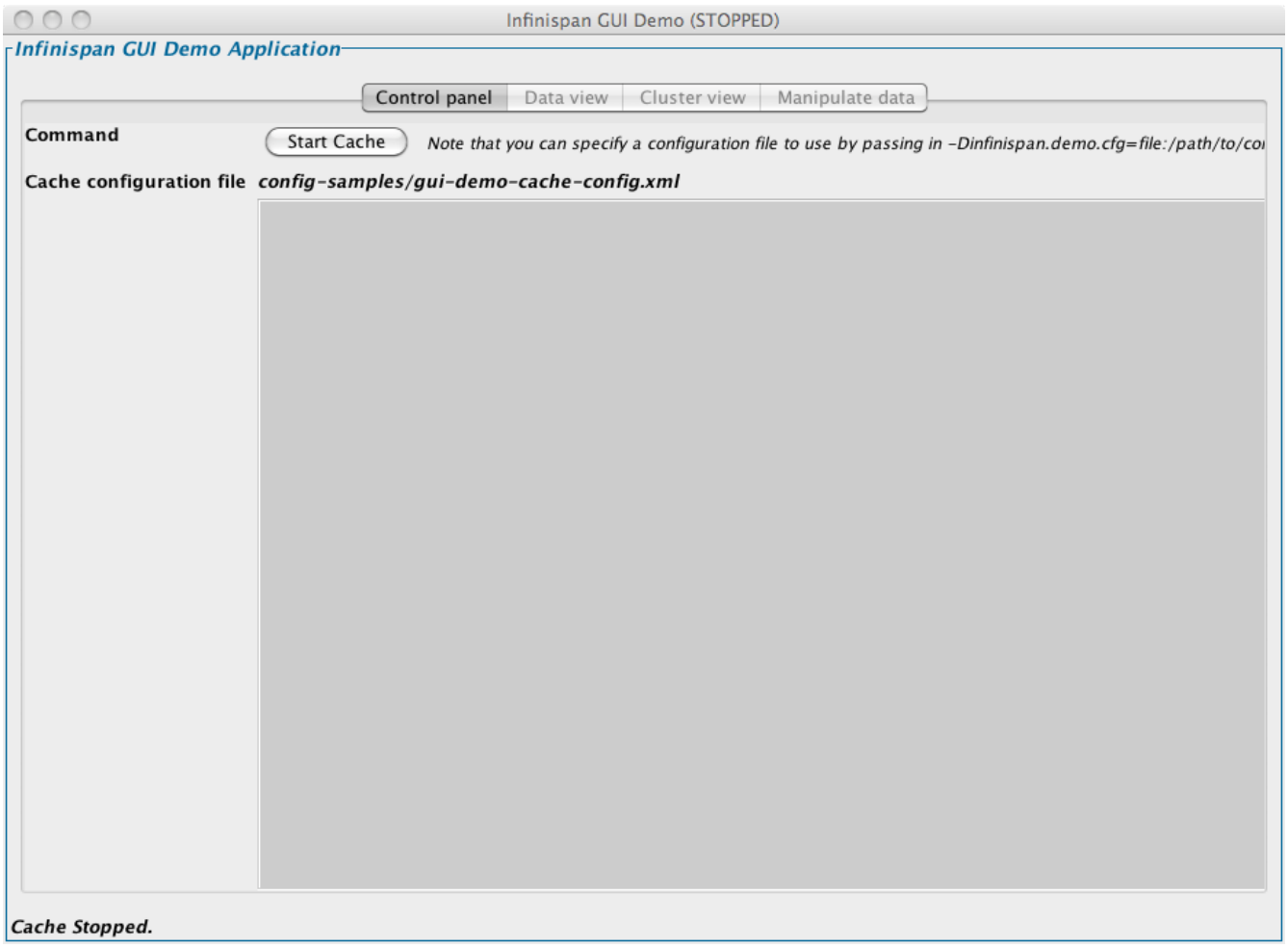
You will need either the -bin.zip or -all.zip version for this demo.

3.1. Step 1: Start the demo GUI

Open up a console and type:

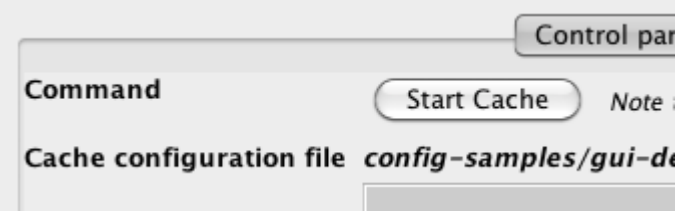
```
$ cd ${INFINISPAN_HOME}
$ bin/runGuiDemo.sh
```

An equivalent `runGuiDemo.bat` file is also provided for Windows users.



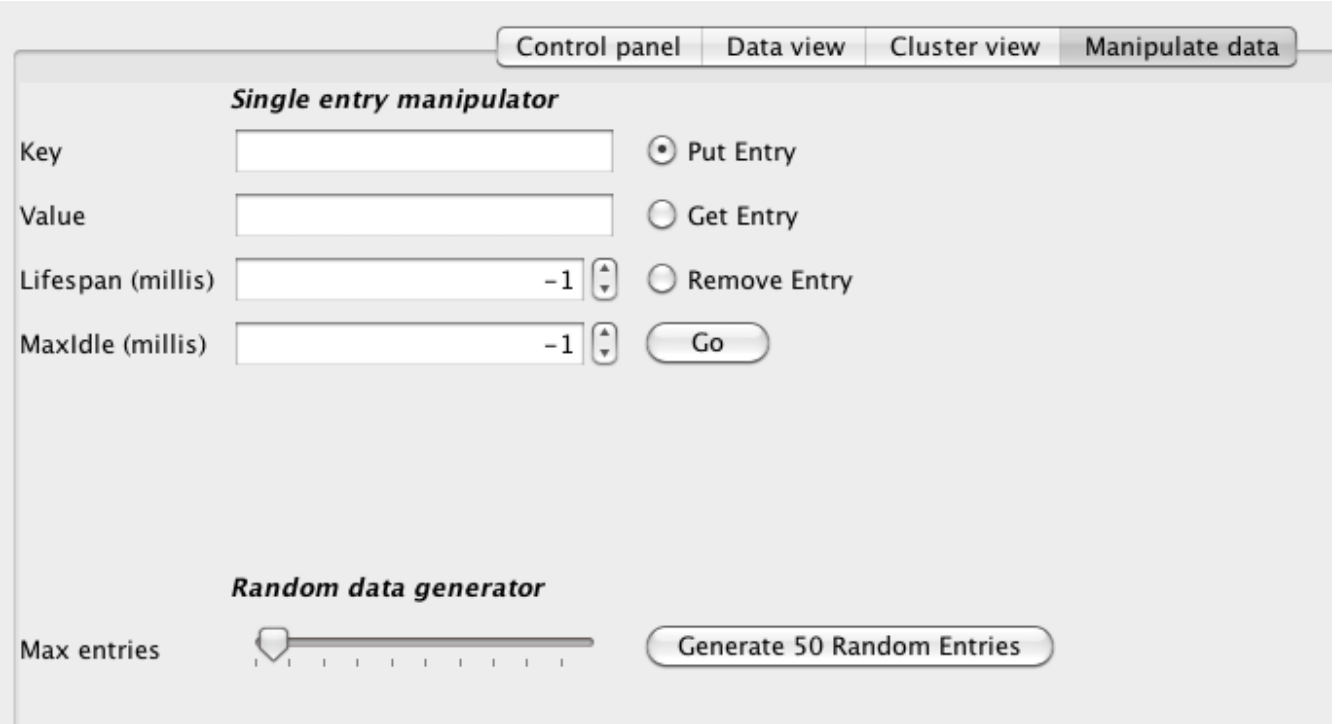
3.2. Step 2: Start the cache

Start the cache in the GUI that starts up, using the *Start Cache* button.



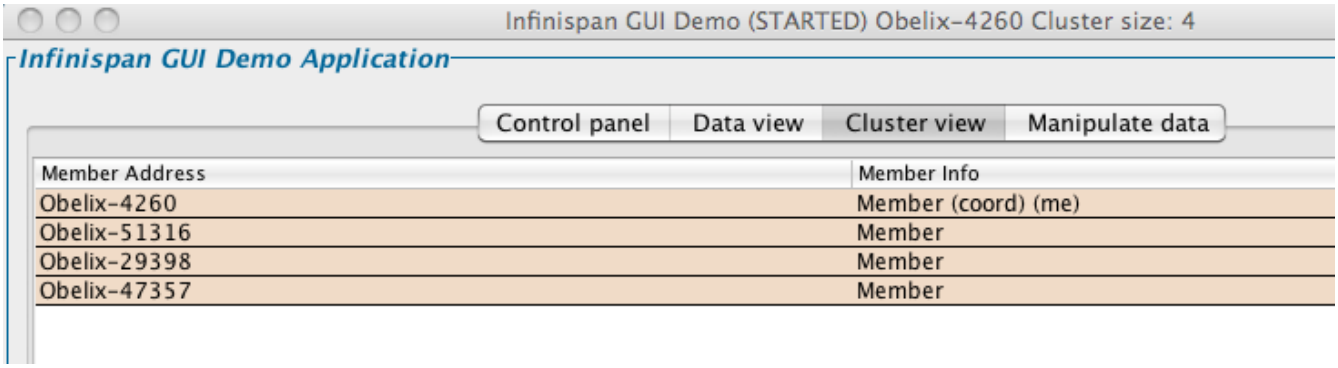
3.3. Step 3: Manipulate data

In the *Manipulate Data* tab, add entries, generate random data, etc.



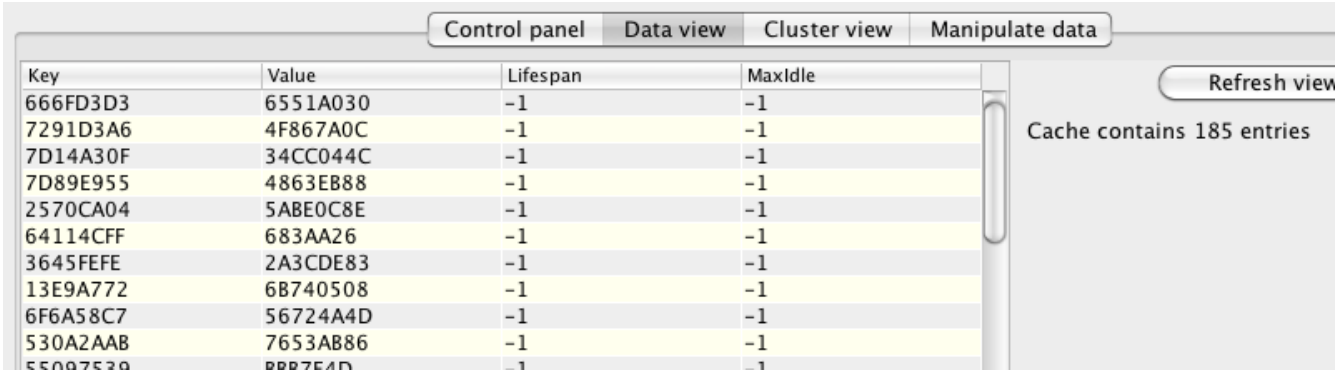
3.4. Step 4: Start more cache instances

Repeat steps 1 and 2 to launch and start up more caches. Watch cluster formation in the *Cluster View* tab.



3.5. Step 5: Manipulate more data

Add and remove data on any of the nodes, and watch state being distributed. Shut nodes down as well to witness data durability.


Control panel
Data view
Cluster view
Manipulate data
Key
Value
Lifespan
MaxIdle
666FD3D3 6551A030 -1 -1
7291D3A6 4F867A0C -1 -1
7D14A30F 34CC044C -1 -1
7D89E955 4863EB88 -1 -1
2570CA04 5ABE0C8E -1 -1
64114CFF 683AA26 -1 -1
3645FEFE 2A3CDE83 -1 -1
13E9A772 6B740508 -1 -1
6F6A58C7 56724A4D -1 -1
530A2AAB 7653AB86 -1 -1
Refresh view
Cache contains 185 entries

Chapter 4. Using {brandname} as an embedded cache in Java SE

Running {brandname} in embedded mode is very easy. First, we'll set up a project, and then we'll run {brandname}, and start adding data.

embedded-cache quickstart

All the code discussed in this tutorial is available in the embedded-cache quickstart.

4.1. Creating a new {brandname} project

The only thing you need to set up {brandname} is add it's dependencies to your project.

4.1.1. Maven users

If you are using Maven (or another build system like Gradle or Ivy which can use Maven dependencies), then this is easy. Just add:

pom.xml

```
<dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-embedded</artifactId>
    <version>${infinispan.version}</version>
</dependency>
```

to the `<dependencies>` section of the POM. You'll need to substitute `${infinispan.version}` for the version of {brandname} you wish to use.

Which version of {brandname} should I use?

We recommend using the latest stable version of {brandname}. All releases are displayed on the downloads page.

Alternatively, you can use the POM from the quickstart that accompanies this tutorial.

4.1.2. Ant users

If you are using Ant, or another build system which doesn't provide declarative dependency management, then the {brandname} distribution zip contains a lib/ directory. Add the contents of this to the build classpath.

4.2. Running {brandname} on a single node

In order to run {brandname}, we're going to create a `main()` method in the `Quickstart` class.

{brandname} comes configured to run out of the box; once you have set up your dependencies, all you need to do to start using {brandname} is to create a new cache manager and get a handle on the default cache.

Quickstart.java

```
public class Quickstart {

   public static void main(String args[]) throws Exception {
      Cache<Object, Object> c = new DefaultCacheManager().getCache();
   }

}
```

We now need a way to run the main method! To run the Quickstart main class: If you are using Maven:

```
$ mvn compile exec:java
-Dexec.mainClass="org.infinispan.quickstart.embeddedcache.Quickstart"
```

You should see {brandname} start up, and the version in use logged to the console.

Congratulations, you now have {brandname} running as a local cache!

4.3. Use the default cache

{brandname} exposes a Map-like, JSR-107-esque interface for accessing and mutating the data stored in the cache. For example:

DefaultCacheQuickstart.java

```
// Add a entry
cache.put("key", "value");
// Validate the entry is now in the cache
assertEqual(1, cache.size());
assertTrue(cache.containsKey("key"));
// Remove the entry from the cache
Object v = cache.remove("key");
// Validate the entry is no longer in the cache
assertEqual("value", v);
```

{brandname} offers a thread-safe data-structure:

DefaultCacheQuickstart.java

```
// Add an entry with the key "key"
cache.put("key", "value");
// And replace it if missing
cache.putIfAbsent("key", "newValue");
// Validate that the new value was not added
```

By default entries are immortal but you can override this on a per-key basis and provide lifespans.

DefaultCacheQuickstart.java

```
//By default entries are immortal but we can override this on a per-key basis and
provide lifespans.
cache.put("key", "value", 5, SECONDS);
assertTrue(cache.containsKey("key"));
Thread.sleep(10000);
```

to run using maven:

```
$ mvn compile exec:java
-Dexec.mainClass="org.infinispan.quickstart.embeddedcache.DefaultCacheQuickstart"
```

4.4. Use a custom cache

Each cache in {brandname} can offer a different set of features (for example transaction support, different replication modes or support for eviction), and you may want to use different caches for different classes of data in your application. To get a custom cache, you need to register it with the manager first:

CustomCacheQuickstart.java

```
public static void main(String args[]) throws Exception {
   EmbeddedCacheManager manager = new DefaultCacheManager();
   manager.defineConfiguration("custom-cache", new ConfigurationBuilder()
              .eviction().strategy(LIRS).maxEntries(10)
              .build());
   Cache<Object, Object> c = manager.getCache("custom-cache");
}
```

The example above uses {brandname}'s fluent configuration, which offers the ability to configure your cache programmatically. However, should you prefer to use XML, then you may. We can create an identical cache to the one created with a programmatic configuration:

To run using maven:

```
$ mvn compile exec:java
```

-Dexec.mainClass="org.infinispan.quickstart.embeddedcache.CustomCacheQuickstart"

infinispan.xml

```
<infinispan
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:infinispan:config:9.3
http://www.infinispan.org/schemas/infinispan-config-9.3.xsd"
        xmlns="urn:infinispan:config:9.3">

   <cache-container default-cache="default">
       <local-cache name="xml-configured-cache">
          <eviction strategy="LIRS" max-entries="10" />
       </local-cache>
   </cache-container>

</infinispan>
```

We then need to load the configuration file, and use the programmatically defined cache:

XmlConfiguredCacheQuickstart.java

```
public static void main(String args[]) throws Exception {
    Cache<Object, Object> c = new DefaultCacheManager("infinispan.xml").getCache("xml-
configured-cache");
}
```

To run using maven:

$ mvn compile exec:java -Dexec.mainClass="org.infinispan.quickstart.embeddedcache.XmlConfiguredCacheQuickstart" == Using {brandname} as an embedded data grid in Java SE Clustering {brandname} is simple. Under the covers, {brandname} uses JGroups as a network transport, and JGroups handles all the hard work of forming a cluster.

clustered-cache quickstart

All the code discussed in this tutorial is available in the clustered-cache quickstart.

4.5. Sharing JGroups channels

By default all caches created from a single CacheManager share the same JGroups channel and multiplex RPC messages over it. In this example caches 1, 2 and 3 all use the same JGroups channel.

```
EmbeddedCacheManager cm = new DefaultCacheManager("infinispan.xml");
Cache<Object, Object> replSyncCache = cm.getCache("replSyncCache");
Cache<Object, Object> replAsyncCache = cm.getCache("replAsyncCache");
Cache<Object, Object> invalidationSyncCache = cm.getCache("invalidationSyncCache");
```

4.6. Running {brandname} in a cluster

It is easy set up a clustered cache. This tutorial will show you how to create two nodes in different processes on the same local machine. The quickstart follows the same structure as the embedded-cache quickstart, using Maven to compile the project, and a main method to launch the node.

If you are following along with the quickstarts, you can try the examples out.

The quickstart defines two clustered caches, one in *replication mode* and one *distribution mode*.

4.6.1. Replicated mode

To run the example in replication mode, we need to launch two nodes from different consoles. For the first node:

```
$ mvn exec:java -Djava.net.preferIPv4Stack=true -Djgroups.bind_addr=127.0.0.1
-Dexec.mainClass="org.infinispan.quickstart.clusteredcache.Node" -Dexec.args="A"
```

And for the second node:

```
$ mvn exec:java -Djava.net.preferIPv4Stack=true -Djgroups.bind_addr=127.0.0.1
-Dexec.mainClass="org.infinispan.quickstart.clusteredcache.Node" -Dexec.args="B"
```

Note: You need to set `-Djava.net.preferIPv4Stack=true` because the JGroups configuration uses IPv4 multicast address. Normally you should not need `-Djgroups.bind_addr=127.0.0.1`, but many wireless routers do not relay IP multicast by default.

Each node will insert or update an entry every second, and it will log any changes.

4.6.2. Distributed mode

To run the example in distribution mode and see how entries are replicated to only two nodes, we need to launch three nodes from different consoles. For the first node:

```
$ mvn compile exec:java -Djava.net.preferIPv4Stack=true
-Dexec.mainClass="org.infinispan.quickstart.clusteredcache.Node" -Dexec.args="-d A"
```

For the second node:

```
$ mvn compile exec:java -Djava.net.preferIPv4Stack=true
-Dexec.mainClass="org.infinispan.quickstart.clusteredcache.Node" -Dexec.args="-d B"
```

And for the third node:

```
$ mvn compile exec:java -Djava.net.preferIPv4Stack=true
-Dexec.mainClass="org.infinispan.quickstart.clusteredcache.Node" -Dexec.args="-d C"
```

The same as in replication mode, each node will insert or update an entry every second, and it will log any changes. But unlike in replication mode, not every node will see every modification.

You can also see that each node holds a different set of entries by pressing Enter.

4.7. clustered-cache quickstart architecture

4.7.1. Logging changes to the cache

An easy way to see what is going on with your cache is to log mutated entries. An {brandname} listener is notified of any mutations:

```
import org.infinispan.notifications.Listener;
import org.infinispan.notifications.cachelistener.annotation.*;
import org.infinispan.notifications.cachelistener.event.*;
import org.jboss.logging.Logger;

@Listener
public class LoggingListener {

   private BasicLogger log = BasicLogFactory.getLog(LoggingListener.class);

   @CacheEntryCreated
   public void observeAdd(CacheEntryCreatedEvent<String, String> event) {
      if (event.isPre())
         return;
      log.infof("Cache entry %s = %s added in cache %s", event.getKey(), event
.getValue(), event.getCache());
   }

   @CacheEntryModified
   public void observeUpdate(CacheEntryModifiedEvent<String, String> event) {
      if (event.isPre())
         return;
      log.infof("Cache entry %s = %s modified in cache %s", event.getKey(), event
.getValue(), event.getCache());
   }

   @CacheEntryRemoved
   public void observeRemove(CacheEntryRemovedEvent<String, String> event) {
      if (event.isPre())
         return;
      log.infof("Cache entry %s removed in cache %s", event.getKey(), event.getCache(
));
   }
}
```

Listeners methods are declared using annotations, and receive a payload which contains metadata about the notification. Listeners are notified of any changes. Here, the listeners simply log any entries added, modified, or removed.

4.7.2. What's going on?

The example allows you to start two or more nodes, each of which are started in a separate process. The node code is very simple, each node starts up, prints the local cache contents, registers a listener that logs any changes, and starts storing entries of the form `key-<counter> = <local address>-counter`.

State transfer

{brandname} automatically replicates the cache contents from the existing members to joining members. This can be controlled in two ways:

- If you don't want the `getCache()` call to block until the entire cache is transferred, you can configure `clustering.stateTransfer.awaitInitialTransfer = false`. Note that `cache.get(key)` will still return the correct value, even before the state transfer is finished.
- If it's fast enough to re-create the cache entries from another source, you can disable state transfer completely, by configuring `clustering.stateTransfer.fetchInMemoryState = false`.

4.8. Configuring the cluster

First, we need to ensure that the cache manager is cluster aware. {brandname} provides a default configuration for a clustered cache manager:

```
GlobalConfigurationBuilder.getClusteredDefault().build()
```

4.8.1. Tweaking the cluster configuration for your network

Depending on your network setup, you may need to tweak your JGroups set up. JGroups is configured via an XML file; the file to use can be specified via the GlobalConfiguration:

```
DefaultCacheManager cacheManager = new DefaultCacheManager(
      GlobalConfigurationBuilder.defaultClusteredBuilder()
            .transport().nodeName(nodeName).addProperty("configurationFile",
"jgroups.xml")
            .build()
);
```

The JGroups documentation provides extensive advice on getting JGroups working on your network. If you are new to configuring JGroups, you may get a little lost, so you might want to try tweaking these configuration parameters:

- Using the system property `-Djgroups.bind_addr=127.0.0.1` causes JGroups to bind only to your loopback interface, meaning any firewall you may have configured won't get in the way. Very useful for testing a cluster where all nodes are on one machine.

TODO - add more tips!

You can also configure the JGroups configuration to use in {brandname}'s XML configuration:

```
<infinispan>
  <jgroups>
     <stack-file name="external-file" path="jgroups.xml"/>
  </jgroups>
  <cache-container>
    <transport stack="external-file" />
  </cache-container>

  ...

</infinispan>
```

4.9. Configuring a replicated data-grid

In replicated mode, {brandname} will store every entry on every node in the grid. This offers high durability and availability of data, but means the storage capacity is limited by the available heap space on the node with least memory. The cache should be configured to work in replication mode (either synchronous or asynchronous), and can otherwise be configured as normal. For example, if you want to configure the cache programmatically:

```
cacheManager.defineConfiguration("repl", new ConfigurationBuilder()
      .clustering()
      .cacheMode(CacheMode.REPL_SYNC)
      .build()
);
```

You can configure an identical cache using XML:

infinispan-replication.xml

```
<infinispan>
  <jgroups/>
  <cache-container default-cache="repl">
     <transport/>
     <replicated-cache name="repl" mode="SYNC" />
  </cache-container>
</infinispan>
```

along with

```
private static EmbeddedCacheManager createCacheManagerFromXml() throws IOException {
   return new DefaultCacheManager("infinispan-replication.xml");
}
```

4.10. Configuring a distributed data-grid

In distributed mode, {brandname} will store every entry on a subset of the nodes in the grid (the parameter numOwners controls how many owners each entry will have). Compared to replication, distribution offers increased storage capacity, but with increased latency to access data from non-owner nodes, and durability (data may be lost if all the owners are stopped in a short time interval). Adjusting the number of owners allows you to obtain the trade off between space, durability, and latency.

{brandname} also offers a *topology aware consistent hash* which will ensure that the owners of entries are located in different data centers, racks, or physical machines, to offer improved durability in case of node crashes or network outages.

The cache should be configured to work in distributed mode (either synchronous or asynchronous), and can otherwise be configured as normal. For example, if you want to configure the cache programmatically:

```
cacheManager.defineConfiguration("dist", new ConfigurationBuilder()
      .clustering()
      .cacheMode(CacheMode.DIST_SYNC)
      .hash().numOwners(2)
      .build()
);
```

You can configure an identical cache using XML:

infinispan-distribution.xml:

```
<infinispan>
  <jgroups/>
  <cache-container default-cache="repl">
    <transport/>
    <distributed-cache owners="2" mode="SYNC" />
  </cache-container>
</infinispan>
```

along with

```
private static EmbeddedCacheManager createCacheManagerFromXml() throws IOException {
   return new DefaultCacheManager("infinispan-distribution.xml");
}
```

Chapter 5. Creating your own {brandname} project

5.1. Maven Archetypes

{brandname} currently has 2 separate Maven archetypes you can use to create a skeleton project and get started using {brandname}. This is an easy way to get started using {brandname} as the archetype generates sample code, a sample Maven pom.xml with necessary dependencies, etc.

You don't need to have any experience with or knowledge of Maven's Archetypes to use this! Just follow the simple steps below.

5.1.1. Starting a new project

Use the newproject-archetype project. The simple command below will get you started, and

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.infinispan.archetypes \
    -DarchetypeArtifactId=newproject-archetype \
    -DarchetypeVersion=1.0.18 \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

You will be prompted for a few things, including the *artifactId* , *groupId* and *version* of your new project. And that's it - you're ready to go!

5.1.2. Playing with your new project

The skeleton project ships with a sample application class, interacting with {brandname}. You should open this new project in your IDE - most good IDEs such as IntelliJ and Eclipse allow you to import Maven projects, see this guide and this guide . Once you open your project in your IDE, you should examine the generated classes and read through the comments.

5.1.3. On the command line...

Try running

```
$ mvn install -Prun verify
```

in your newly generated project! This runs the main() method in the generated application class.

5.1.4. Writing a test case for {brandname}

This archetype is useful if you wish to contribute a test to the {brandname} project and helps you get set up to use {brandname}'s testing harness and related tools. Use

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.infinispan.archetypes \
    -DarchetypeArtifactId=testcase-archetype \
    -DarchetypeVersion=1.0.18 \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

As above, this will prompt you for project details and again as above, you should open this project in your IDE. Once you have done so, you will see some sample tests written for {brandname} making use of {brandname}'s test harness and testing tools along with extensive comments and links for further reading.

5.1.5. On the command line...

Try running

```
$ mvn test
```

in your newly generated project to run your tests.

The generated project has a few different profiles you can use as well, using Maven's -P flag. E.g.,

```
$ mvn test -Pudp
```

Available profiles

The profiles available in the generated sample project are:

- udp: use UDP for network communications rather than TCP
- tcp: use TCP for network communications rather than UDP
- jbosstm: Use the embedded JBoss Transaction Manager rather than {brandname}'s embedded transaction manager

Contributing tests back to {brandname}

If you have written a functional, unit or stress test for {brandname} and want to contribute this back to {brandname}, your best bet is to fork the {brandname} sources on GitHub . The test you would have prototyped and tested in an isolated project created using this archetype can be simply dropped in to {brandname}'s test suite. Make your changes, add your test, prove that it fails even on {brandname}'s upstream source tree and issue a pull request .

New to working with {brandname} and GitHub? Want to know how best to work with the repositories and contribute code? Read {brandname} and GitHub

5.1.6. Versions

The archetypes generate poms with dependencies to specific versions of {brandname}. You should edit these generated poms by hand to point to other versions of {brandname} that you are interested in.

5.1.7. Source Code

The source code used to generate these archetypes are on GitHub . If you wish to enhance and contribute back to the project, fork away!

Chapter 6. Using {brandname} as a second level cache for Hibernate

TODO

Chapter 7. Accessing an {brandname} data grid remotely

7.1. Using Hot Rod to access an {brandname} data-grid

TODO

7.2. Using REST to access an Infinipsan data-grid

TODO

7.3. Using memcached to access an {brandname} data-grid

TODO

Chapter 8. Using {brandname} in WildFly

TODO

Chapter 9. Using {brandname} in servlet containers (such as Tomcat or Jetty) and other application servers (such as GlassFish)

TODO

Chapter 10. Monitoring {brandname}

- TODO *

Chapter 11. Example with Groovy

The idea by this tutorial is to give an introduction in the use of the {brandname} API and its configuration file. As trying to do it in a more interactive fashion, the tutorial makes use of the Groovy dynamic language that will allow to interact with the API by using a console.

The tutorial will start by showing the basic usage of the {brandname} API and a use of a simple cache configuration, then it will walk through different configuration scenarios and use cases. By the end of the tutorial you should have a clear understanding of the use the {brandname} API and some of the various configuration options.

The scenarios and use cases shown are:

- Basic cache configuration
- Cache with transaction management configuration
- Cache with a cache store configuration
- Cache with eviction configuration
- Cache with eviction and cache store configuration
- Cache with REPL_SYNC & transaction management configuration.

All the sample configurations are in the sample-configurations.xml file attached to this tutorial, check the configuration documentation to know how to make use of this configuration file. Lets get started:

11.1. Introduction

The {brandname} tutorial makes use of Groovy to get a more interactive experience when starting to learn about how to use the {brandname} API. So you will need to install a few prerequisites before getting started:

- The Groovy Platform (I used Groovy 1.6.3)
- Java and {brandname}

Download those and extract/install where you feel appropriate, depending on your operating system and personal preferences you will either have installers or compressed distributions. You can read more about read installing Java and {brandname} in Installing {brandname} for the tutorials .

11.1.1. Installing Groovy

You can use the installer or compressed file to install the Groovy Platform, I used the compressed file and decompressed at C:\Program Files\groovy\groovy-1.6.3. Once you have installed the Groovy Platform you should set some environment variables:

```
GROOVY_HOME=C:\Program Files\groovy\groovy-1.6.3
```

and add to the PATH environment variable:

```
PATH=%PATH%;%GROOVY_HOME%\bin
```

test that everything is correct by executing in a Command Shell/Terminal the commands shown:

```
$> groovy -v
Groovy Version: 1.6.3 JVM: 1.6.0_14
```

If you get a similar result as shown, everything went well.

11.1.2. Installing {brandname}

Now you should add the {brandname} libraries to the Groovy Platform so you will able to access the API from the Groovy console. Add the infinispan-core.jar and its dependencies to the $USER_HOME/.groovy/lib directory, the jar is located in $INFINISPAN_HOME/modules/core and the dependencies at $INIFINISPAN_HOME/modules/core/lib.

For example, on Windows, you need to copy it to:

```
C:\Documents and Settings\Alejandro Montenegro\.groovy\lib
```

or on Linux:

```
/home/amontenegro/.groovy/lib
```

and $INFINISPAN_HOME is where you decompressed the {brandname} distribution.

To test the installation, download the attached file infinispantest.groovy and in a Command Shell/Terminal execute

```
$> groovy infinispantest
4.0.0.ALPHA5
```

11.1.3. Setting the classpath

The last thing to do is to add to the CLASSPATH environment variable the sample configuration file, this file contains definitions of cache's that will be used in the tutorial. I created the directory $USER_HOME/.groovy/cp and added it to the classpath

For example, on Windows:

```
CLASSPATH=%CLASSPATH%;C:\Documents and Settings\Alejandro Montenegro\.groovy\cp
```

or, on Linux:

```
CLASSPATH=$CLASSPATH:/home/amontenegro/.groovy/cp
```

finally add the sample-configurations.xml and infinispan-config-4.0.xsd files(attached) to the directory.

11.2. Loading the configuration file

The cache manager is the responsible to manage all the cache's , so you have to start by indicating where to get the cache definitions to the cache manager , remember that the cache definitions are in the sample-configurations.xml file. If no cache definitions are indicated, the cache manager will use a default cache.

Start by open a groovy console by typing groovy.sh in a command shell or terminal. You should now have something similar to:

Groovy Shell (1.6.3, JVM: 1.6.0_14) Type 'help' or '\h' for help.

```
groovy:000>
```

It's time to start typing some commands, first start by importing the necessary libraries

groovy:000> import org.infinispan.* === > [import org.infinispan.**] groovy:000> import org.infinispan.manager.** === > [import org.infinispan.**, import org.infinispan.manager.**]

And now, create a cache manager indicating the file with the cache definitions.

```
groovy:000> manager = new DefaultCacheManager("sample-configurations.xml")
=== > org.infinispan.manager.DefaultCacheManager@19cc1b@Address:null
```

the cache manager has now the knowledge of all the named caches defined in the configuration file and also has a no named cache that's used by default. You can now access any of the cache's by interacting with the cache manager as shown.

```
groovy:000> defaultCache = manager.getCache()
=== > Cache 'org.infinispan.manager.DefaultCacheManager.DEFAULT_CACHE_NAME'@7359733
//TO GET A NAMED CACHE
groovy:000> cache = manager.getCache("NameOfCache")
```

11.3. Basic cache configuration

The basic configuration, is the simplest configuration that you can have, its make use of default settings for the properties of the cache configuration, the only thing you have to set is the name of the cache.

```
<namedCache name="Local"/>
```

That's all you have to add to the configuration file to have a simple named cache, now its time to interact with the cache by using the {brandname} API. Lets start by getting the named cache and put some objects inside it.

```
//START BY GETTING A REFERENCE TO THE NAMED CACHE
groovy:000> localCache = manager.getCache("Local")
=== > Cache 'Local'@19521418
//THE INITIAL SIZE IS 0
groovy:000> localCache.size()
=== > 0
//NOW PUT AN OBJECT INSIDE THE CACHE
groovy:000> localCache.put("aKey", "aValue")
=== > null
//NOW THE SIZE IS 1
groovy:000> localCache.size()
=== > 1
//CHECK IF IT HAS OUR OBJECT
groovy:000> localCache.containsKey("aKey")
=== > true
//BY OBTAINING AN OBJECT DOESN'T MEAN TO REMOVE
groovy:000> localCache.get("aKey")
=== > aValue
groovy:000> localCache.size()
=== > 1
//TO REMOVE ASK IT EXPLICITLY
groovy:000> localCache.remove("aKey")
=== > aValue
groovy:000> localCache.isEmpty()
=== > true
```

So you have seen the basic of the {brandname} API, adding, getting and removing from the cache, there is more, but don't forget that you are working with a cache that are an extension of java.util.ConcurrentHasMap and the rest of the API is as simple as the one shown above, many of the cool things in {brandname} are totally transparent (that's actually the coolest thing about {brandname}) and depends only on the configuration of your cache.

If you check the {brandname} JavaDoc you will see that the Cache#put() method has been overridden several times.

```
//YOU WILL NEED TO IMPORT ANOTHER LIBRARY
groovy:000> import java.util.concurrent.TimeUnit
=== > [import org.infinispan.*, import org.infinispan.manager.*, import
java.util.concurrent.TimeUnit]
//NOTHING NEW HERE JUST PUTTING A NEW OBJECT
groovy:000> localCache.put("bKey", "bValue")
=== > null
//WOW! WHATS HAPPEN HERE? PUTTED A NEW OBJECT BUT IT WILL TIMEOUT AFTER A SECOND
groovy:000> localCache.put("timedKey", "timedValue", 1000, TimeUnit.MILLISECONDS)
=== > null
//LETS CHECK THE SIZE
groovy:000> localCache.size()
=== > 2
//NOW TRY TO GET THE OBJECT, OOPS ITS GONE! (IF NOT, IT'S BECAUSE YOU ARE A
SUPERTYPER, CALL GUINNESS!))
groovy:000> localCache.get("timedKey")
=== > null
//LETS CHECK THE SIZE AGAIN, AS EXPECTED THE SIZE DECREASED BY 1
groovy:000> localCache.size()
=== > 1
```

The {brandname} API also allows you to manage the life cycle of the cache, you can stop and start a cache but by default you will loose the content of the cache except if you configure a cache store, more about that later in the tutorial. lets check what happens when you restart the cache

```
groovy:000> localCache.size()
=== > 1
//RESTARTING CACHE
groovy:000> localCache.stop()
=== > null
groovy:000> localCache.start()
=== > null
//DAMN! LOST THE CONTENT OF THE CACHE
groovy:000> localCache.size()
=== > 0
```

Thats all related to the use of the {brandname} API, now lets check some different behaviors depending on the configuration of the cache.

11.4. Cache with transaction management

You are able to specify the cache to use a transaction manager, and even explicitly control the transactions. Start by configuring the cache to use a specific TransactionManagerLookup class. {brandname} implements a couple TransactionManagerLookup classes.

- org.infinispan.transaction.lookup.EmbeddedTransactionManager
- org.infinispan.transaction.lookup.GenericTransactionManagerLookup

- org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup

Each use different methods to lookup the transaction manager, depending on the environment you are running {brandname} you should figure out which one to use. Check the JavaDoc for more details.

For the tutorial its enough to use:

```
<namedCache name="LocalTX">
    <transaction transactionManagerLookupClass=
"org.infinispan.transaction.lookup.EmbeddedTransactionManagerLookup"/>
</namedCache>
```

Lets check how to interact with the Transaction Manager and to have the control over a transaction

```
groovy:000> import javax.transaction.TransactionManager
=== > [import org.infinispan.*, import org.infinispan.manager.*, import
java.util.concurrent.TimeUnit, import javax.transaction.TransactionManager]
//GET A REFERENCE TO THE CACHE WITH TRANSACTION MANAGER
groovy:000> localTxCache = manager.getCache("LocalTX")
=== > Cache 'LocalTX'@16075230
groovy:000> cr = localTxCache.getComponentRegistry()
=== > org.infinispan.factories.ComponentRegistry@87e9bf
//GET A REFERENCE TO THE TRANSACTION MANAGER
groovy:000> tm = cr.getComponent(TransactionManager.class)
=== > org.infinispan.transaction.tm.EmbeddedTransactionManager@b5d05b
//STARTING A NEW TRANSACTION
groovy:000> tm.begin()
=== > null
//PUTTING SOME OBJECTS INSIDE THE CACHE
groovy:000> localTxCache.put("key1", "value1")
=== > null
//MMM SIZE DOESN'T INCREMENT
groovy:000> localTxCache.size()
=== > 1
//LETS TRY AGAIN
groovy:000> localTxCache.put("key2", "value2")
=== > null
//MMM NOTHING..
groovy:000> localTxCache.size()
=== > 2
//OH! HAS TO DO THE COMMIT
groovy:000> tm.commit()
=== > null
//AND THE SIZE IS AS EXPECTED.. HAPPY!
groovy:000> localTxCache.size()
=== > 2
```

As shown in the example, the transaction is controlled explicitly and the changes in the cache wont

be reflected until you make the commit.

11.5. Cache with a cache store

{brandname} allows you to configure a persistent store that can be used to persist the content of the cache, so if the cache is restarted the cache will be able to keep the content. It can also be used if you want to limit the size of the cache, then the cache will start putting the objects in the store to keep the size limit, more on that when looking at the eviction configuration.

{brandname} provides several cache store implementations:

- FileCacheStore
- JdbcBinaryCacheStore
- JdbcMixedCacheStore
- JdbcStringBasedCacheStore
- JdbmCacheStore
- S3CacheStore
- BdbjeCacheStore

> The tutorial uses the FileCacheStore, that saves the objects in files in a configured directory, in this case the /tmp directory. If the directory is not set it defaults to {brandname}-FileCacheStore in the current working directory.

```
<namedCache name="CacheStore">
     <loaders passivation="false" shared="false" preload="true">
          <loader class="org.infinispan.loaders.file.FileCacheStore"
fetchPersistentState="true"
               ignoreModifications="false" purgeOnStartup="false">
            <properties>
                 <property name="location" value="/tmp"/>
               </properties>
          </loader>
     </loaders>
</namedCache>
```

Now you have a cache with persistent store, lets try it to see how it works

```
//GETTING THE NEW CACHE
groovy:000> cacheCS = manager.getCache("CacheStore")
=== > Cache 'CacheStore'@23240342
//LETS PUT AN OBJECT INSIDE THE CACHE
groovy:000> cacheCS.put("storedKey", "storedValue")
=== > null
//LETS PUT THE SAME OBJECT IN OUR BASIC CACHE
groovy:000> localCache.put("storedKey", "storedValue")
=== > storedValue
//RESTART BOTH CACHES
groovy:000> cacheCS.stop()
=== > null
groovy:000> localCache.stop()
=== > null
groovy:000> cacheCS.start()
=== > null
groovy:000> localCache.start()
=== > null
//LETS TRY GET THE OBJECT FROM THE RESTARTED BASIC CACHE.. NO LUCK
groovy:000> localCache.get("storedKey")
=== > null
//INTERESTING CACHE SIZE IS NOT CERO
groovy:000> cacheCS.size()
=== > 1
//WOW! JUST RESTARTED THE CACHE AND THE OBKECT KEEPS STAYING THERE!
groovy:000> cacheCS.get("storedKey")
=== > storedValue
```

11.6. Cache with eviction

The eviction allow to define policy for removing objects from the cache when it reach its limit, as the true is that the caches doesn't has unlimited size because of many reasons. So the fact is that you normally will set a maximum number of objects in the cache and when that number is reached then the cache has to decide what to do when a new object is added. That's the whole story about eviction, to define the policy of removing object when the cache is full and want to keep putting objects. You have three eviction strategies:

- NONE
- LRU
- LIRS

Let check the configuration of the cache:

```
<namedCache name="Eviction">
   <eviction wakeUpInterval="500" maxEntries="2" strategy="LRU"/>
</namedCache>
```

The strategy has been set to LRU, so the least recently used objects will be removed first and the maximum number of objects are only 2, so it will be easy to show how it works

```
//GETTING THE NEW CACHE
groovy:000> evictionCache = manager.getCache("Eviction")
=== > Cache 'Eviction'@5132526
//PUT SOME OBJECTS
groovy:000> evictionCache.put("key1", "value1")
=== > null
groovy:000> evictionCache.put("key2", "value2")
=== > null
groovy:000> evictionCache.put("key3", "value3")
=== > null
//HEY! JUST LOST AN OBJECT IN MY CACHE.. RIGHT, THE SIZE IS ONLY TWO
groovy:000> evictionCache.size()
=== > 2
//LETS CHECK WHAT OBJECT WAS REMOVED
groovy:000> evictionCache.get("key3")
=== > value3
groovy:000> evictionCache.get("key2")
=== > value2
//COOL! THE OLDEST WAS REMOVED
groovy:000> evictionCache.get("key1")
=== > null
```

Now you are sure that your cache wont consume all your memory and hang your system, but its an expensive price you have to pay for it, you are loosing objects in your cache. The good news is that you can mix cache store with the eviction policy and avoid loosing objects.

11.7. Cache with eviction and cache store

Ok, the cache has a limited size but you don’t want to loose your objects in the cache. {brandname} is aware of these issues, so it makes it very simple for you combing the cache store with the eviction policy. When the cache is full it will persist an object and remove it from the cache, but if you want to recover an object that has been persisted the the cache transparently will bring it to you from the cache store.

The configuration is simple, just combine eviction and cache store configuration

```
<namedCache name="CacheStoreEviction">
     <loaders passivation="false" shared="false" preload="true">
          <loader class="org.infinispan.loaders.file.FileCacheStore"
fetchPersistentState="true"
            ignoreModifications="false" purgeOnStartup="false">
               <properties>
                    <property name="location" value="/tmp"/>
               </properties>
          </loader>
     </loaders>
     <eviction wakeUpInterval="500" maxEntries="2" strategy="FIFO"/>
</namedCache>
```

Nothing new in the configuration, lets check how it works

```
//GETTING THE CACHE
groovy:000> cacheStoreEvictionCache = manager.getCache("CacheStoreEviction")
=== > Cache 'CacheStoreEviction'@6208201
//PUTTING SOME OBJECTS
groovy:000> cacheStoreEvictionCache.put("cs1", "value1")
=== > value1
groovy:000> cacheStoreEvictionCache.put("cs2", "value2")
=== > value2
groovy:000> cacheStoreEvictionCache.put("cs3", "value3")
=== > value3
///MMM SIZE IS ONLY TWO, LETS CHECK WHAT HAPPENED
groovy:000> cacheStoreEvictionCache.size()
=== > 2
groovy:000> cacheStoreEvictionCache.get("cs3")
=== > value3
groovy:000> cacheStoreEvictionCache.get("cs2")
=== > value2
//WOW! EVEN IF THE CACHE SIZE IS 2, I RECOVERED THE THREE OBJECTS.. COOL!!
groovy:000> cacheStoreEvictionCache.get("cs1")
=== > value1
```

Chapter 12. Example with Scala

This article shows how to use {brandname} with Scala language . It uses the same commands and configurations used in the Example with Groovy. For more details about the scenarios and steps please visit about page since here will will only focus on Scala compatibility.

12.1. Environment

Running the shell

```
 $ scala -cp infinispan-embedded-9.3.0.Final.jar:lib/jboss-transaction-api_1.1_spec-
1.0.1.Final.jar:sample-configurations.xml
```

where infinispan-embedded-9.3.0.Final.jar, jboss-transaction-api_1.1_spec-1.0.1.Final.jar are shipped with the infinispan zip.

sample-configurations.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<infinispan
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:infinispan:config:9.3
http://infinispan.org/schemas/infinispan-config-9.3.xsd"
        xmlns="urn:infinispan:config:9.3">

   <jgroups>
      <stack-file name="tcpStack" path="default-configs/default-jgroups-tcp.xml"/>
   </jgroups>

   <cache-container default-cache="default">
      <transport stack="tcpStack" cluster="sampleCluster"/>
      <local-cache name="LocalTX">
         <transaction transaction-manager-lookup=
"org.infinispan.transaction.lookup.EmbeddedTransactionManagerLookup" />
      </local-cache>
      <local-cache name="CacheStore">
         <persistence>
            <file-store path="${java.io.tmpdir}" shared="false" preload="false" />
         </persistence>
      </local-cache>
      <local-cache name="Eviction">
         <eviction max-entries="2" strategy="FIFO" />
         <expiration interval="500" />
      </local-cache>
      <local-cache name="CacheStoreEviction">
         <eviction max-entries="2" strategy="FIFO" />
         <expiration interval="500" />
         <persistence>
            <file-store path="${java.io.tmpdir}" shared="false" preload="false" purge
="false" fetch-state="true" />
         </persistence>
      </local-cache>
      <replicated-cache name="ReplicatedTX" mode="SYNC" remote-timeout="20000">
         <state-transfer timeout="20000" />
      </replicated-cache>
   </cache-container>

</infinispan>
```

12.2. Testing Setup

The following code shows how to start an Scala console that will allow commands to be entered interactively. To verify that the {brandname} classes have been imported correctly, an import for all {brandname} classes will be attempted and then a request will be made to print the version of

{brandname}:

```
 $ scala -cp infinispan-embedded-9.3.0.Final.jar:lib/jboss-transaction-api_1.1_spec-
1.0.1.Final.jar:sample-configuration.xml
Welcome to Scala version 2.11.1 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_60).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import org.infinispan._
import org.infinispan._

scala> println(Version.VERSION)
9.3.0.Final
```

12.3. Loading the Configuration file

In this next example, a new cache manager will be created using the configuration file downloaded earlier:

```
scala> import org.infinispan.manager._
import org.infinispan.manager._

scala> val manager = new DefaultCacheManager("sample-configurations.xml")
manager: org.infinispan.manager.DefaultCacheManager =
org.infinispan.manager.DefaultCacheManager@38b58e73@Address:null
```

Retrieving cache instances from cache manager

In this example, the default cache is retrieved expecting keys and values to be of String type:

```
scala> val defaultCache = manager.getCache[String, String]()
defaultCache: org.infinispan.Cache[String,String] = Cache '___defaultcache'@1326840752
```

In this next example, a named cache is retrieved, again with keys and values expected to be String:

```
scala> val namedCache = manager.getCache[String, String]("NameOfCache")
namedCache: org.infinispan.Cache[String,String] = Cache 'NameOfCache'@394890130
```

12.4. Basic cache operations

In this section, several basic operations will be executed against the cache that show how it can be populated with data, how data can be retrieved and size can be checked, and finally how after removing the data entered, the cache is empty:

```
scala> val localCache = manager.getCache[String, String]("Local")
localCache: org.infinispan.Cache[String,String] = Cache 'Local'@420875876

scala> localCache.size
res0: Int = 0

scala> localCache.put("aKey", "aValue")
res1: String = null
// This null was returned by put() indicating that
// the key was not associated with any previous value.

scala> localCache.size
res2: Int = 1

scala> localCache.containsKey("aKey")
res3: Boolean = true

scala> localCache.get("aKey")
res4: String = aValue

scala> localCache.size
res5: Int = 1

scala> localCache.remove("aKey")
res6: String = aValue

scala> localCache.isEmpty
res7: Boolean = true
```

12.5. Basic cache operations with TTL

When a cache entry is stored, a maximum lifespan for the entry can be provided. So, when that time is exceeded, the entry will dissapear from the cache:

```
scala> localCache.put("bKey", "bValue")
res8: String = null

scala> import java.util.concurrent.TimeUnit
import java.util.concurrent.TimeUnit

scala> localCache.put("timedKey", "timedValue", 1000, TimeUnit.MILLISECONDS)
res9: String = null

scala> localCache.size
res10: Int = 2

scala> localCache.get("timedKey")
res11: String = null

scala> localCache.size
res12: Int = 1
```

12.6. Cache restarts

When caches are local and not configured with a persistent store, restarting them means that the data is gone. To avoid this issue you can either configure caches to be clustered so that if one cache dissapears, the data is not completely gone, or configure the cache with a persistent cache store. The latter option will be explained later on.

```
scala> localCache.size
res13: Int = 1

scala> localCache.stop

scala> localCache.start

scala> localCache.size
res16: Int = 0
```

12.7. Transactional cache operations

{brandname} caches can be operated within a transaction, in such way that operations can be grouped in order to be executed atomically. The key thing to understand about transactions is that within the transactions changes are visible, but to other non-transactional operations, or other transactions, these are not visible until the transaction is committed. The following example shows how within a transaction an entry can be stored but outside the transaction, this modification is not yet visible, and that once the transaction is committed, the modification is visible to all:

```
scala> import javax.transaction.TransactionManager
import javax.transaction.TransactionManager

scala> val localTxCache = manager.getCache[String, String]("LocalTX")
localTxCache: org.infinispan.Cache[String,String] = Cache 'LocalTX'@955386212

scala> val tm = localTxCache.getAdvancedCache().getTransactionManager()
tm: javax.transaction.TransactionManager =
org.infinispan.transaction.tm.EmbeddedTransactionManager@81ee8c1

scala> tm.begin

scala> localTxCache.put("key1", "value1")
res1: String = null

scala> localTxCache.size
res2: Int = 1

scala> val tx = tm.suspend
res3: javax.transaction.Transaction = EmbeddedTransaction{xid=DummyXid{id=1},
status=0}

scala> localTxCache.size
res4: Int = 0

scala> localTxCache.get("key1")
res5: String = null

scala> tm.resume(tx)

scala> localTxCache.size()
res7: Int = 1

scala> localTxCache get "key1"
res8: String = value1

scala> tm.commit

scala> localTxCache.size
res10: Int = 1

scala> localTxCache get "key1"
res11: String = value1
```

Note how this example shows a very interesting characteristic of the Scala console. Every operation's return value is stored in a temporary variable which can be referenced at a later stage, even if the user forgets to assign the result of a operation when the code was executed.

12.8. Persistent stored backed Cache operations

When a cache is backed by a persistent store, restarting the cache does not lead to data being lost. Upon restart, the cache can retrieve in lazy or prefetched fashion cache entries stored in the backend persistent store:

```
scala> val cacheWithStore = manager.getCache[String, String]("CacheStore")
cacheWithStore: org.infinispan.Cache[String,String] = Cache 'CacheStore'@2054925789

scala> cacheWithStore.put("storedKey", "storedValue")
res21: String = null

scala> localCache.put("storedKey", "storedValue")
res22: String = null

scala> cacheWithStore.stop

scala> localCache.stop

scala> cacheWithStore.start

scala> localCache.start

scala> localCache.get("storedKey")
res27: String = null

scala> cacheWithStore.size
res28: Int = 1

scala> cacheWithStore.get("storedKey")
res29: String = storedValue
```

12.9. Operating against a size bounded cache

{brandname} caches can be configured with a max number of entries, so if this is exceeded certain cache entries are evicted from in-memory cache. Which cache entries get evicted is dependant on the eviction algorithm chosen. In this particular example, FIFO algorithm has been configured, so when a cache entry needs to be evicted, those stored first will go first:

```
scala> val evictionCache = manager.getCache[String, String]("Eviction")
evictionCache: org.infinispan.Cache[String,String] = Cache 'Eviction'@882725548

scala> evictionCache.put("key1", "value1")
res30: String = null

scala> evictionCache.put("key2", "value2")
res31: String = null

scala> evictionCache.put("key3", "value3")
res32: String = null

scala> evictionCache.size()
res33: Int = 2

scala> evictionCache.get("key3")
res34: String = value3

scala> evictionCache.get("key2")
res35: String = value2

scala> evictionCache.get("key1")
res36: String = null
```

12.10. Size bounded caches with persistent store

When caches configured with eviction are configured with a persistent store as well, when the cache exceeds certain size, apart from removing the corresponding cache entries from memory, these entries are stored in the persistent store. So, if they're requested by cache operations, these are retrieved from the cache store:

```
scala> val cacheStoreEvictionCache = manager.getCache[String,
String]("CacheStoreEviction")
cacheStoreEvictionCache: org.infinispan.Cache[String,String] = Cache
'CacheStoreEviction'@367917752

scala> cacheStoreEvictionCache.put("cs1", "value1")
res37: String = null

scala> cacheStoreEvictionCache.put("cs2", "value2")
res38: String = null

scala> cacheStoreEvictionCache.put("cs3", "value3")
res39: String = null

scala> cacheStoreEvictionCache.size()
res40: Int = 2

scala> cacheStoreEvictionCache.get("cs3")
res41: String = value3

scala> cacheStoreEvictionCache.get("cs2")
res42: String = value2

scala> cacheStoreEvictionCache.get("cs1")
res43: String = value1
```