

Hazelcast Documentation

version 3.2.4

Jul 21, 2014

In-Memory Data Grid - Hazelcast | Documentation: version 3.2.4

Publication date Jul 21, 2014

Copyright © 2014 Hazelcast, Inc.

Permission to use, copy, modify and distribute this document for any purpose and without fee is hereby granted in perpetuity, provided that the above copyright notice and this paragraph appear in all copies.

Contents

1	Introduction	9
1.1	Hazelcast Overview	9
1.2	Why Hazelcast?	10
1.3	Getting Started	11
1.3.1	Installing Hazelcast	11
1.3.2	Starting the Cluster and Client	12
1.3.3	Configuring Hazelcast	14
1.4	Deployment Types	14
1.5	Use Cases	14
1.6	Resources	15
2	What's New in Hazelcast 3.2	17
2.1	Release Notes	17
2.1.1	New Features	17
2.1.2	Improvements	17
2.1.3	Fixes	17
2.1.4	Known Issues & Workarounds	20
2.2	Upgrading from 2.x versions	20
2.3	Document Revision History	22
3	Distributed Data Structures	25
3.1	Map	26
3.1.1	Backups	27
3.1.2	Eviction	28
3.1.3	Persistence	29
3.1.4	Interceptors	32
3.1.5	Near Cache	35
3.1.6	Entry Statistics	36
3.1.7	In Memory Format	37
3.2	Queue	37
3.2.1	Persistence	38
3.3	MultiMap	39

3.4	Set	39
3.4.1	Sample Set Code	39
3.4.2	Event Registration and Configuration	40
3.5	List	40
3.5.1	Sample List Code	41
3.5.2	Event Registration and Configuration	41
3.6	Topic	42
3.6.1	Statistics	42
3.6.2	Internals	42
3.6.3	Topic Configuration	43
3.6.4	Sample Topic Code	44
3.7	Lock	45
3.7.1	ICCondition	46
4	Distributed Events	49
4.1	Event Listeners	49
4.2	Global Event Configuration	49
5	Distributed Computing	51
5.1	Executor Service	51
5.1.1	Execution	52
5.1.2	Execution Cancellation	52
5.1.3	Execution Callback	53
5.2	Entry Processor	54
6	Distributed Query	57
6.1	Query	57
6.1.1	Distributed SQL Query	58
6.1.2	Criteria API	59
6.1.3	Paging Predicate (Order & Limit)	59
6.1.4	Indexing	60
6.2	MapReduce	61
6.2.1	MapReduce Essentials	61
6.2.2	Introduction to MapReduce API	63
6.2.3	Hazelcast MapReduce Architecture	70
6.3	Continuous Query	72
7	Transactions	73
7.1	Transaction Interface	73
7.2	J2EE Integration	74
7.2.1	Resource Adapter Configuration	74
7.2.2	Sample Glassfish v3 Web Application Configuration	75
7.2.3	Sample JBoss Web Application Configuration	75

8	Integrated Clustering	77
8.1	Hibernate Second Level Cache	77
8.2	HTTP Session Clustering with Hazelcast WM	79
8.3	Spring Integration	82
8.3.1	Configuration	82
8.3.2	Spring Managed Context	86
8.3.3	Spring Cache	88
8.3.4	Hibernate 2nd Level Cache Config	88
8.3.5	Spring Data - JPA	88
8.3.6	Spring Data - MongoDB	90
9	Storage	91
9.1	Elastic Memory	91
10	Clients	93
10.1	Native Clients	93
10.1.1	Java Client	93
10.1.2	C++ Client	95
10.1.3	C# Client	100
10.2	REST Client	103
10.3	Memcache Client	105
10.3.1	Unsupported Operations	106
11	Serialization	107
11.1	Data Serialization	108
11.1.1	IdentifiedDataSerializable	109
11.2	Portable Serialization	109
11.3	Custom Serialization	111
12	Management	113
12.1	Monitoring with JMX	113
12.2	Cluster Utilities	114
12.2.1	Cluster Interface	114
12.2.2	Cluster Wide ID Generator	115
12.3	Management Center	115
12.3.1	Introduction	115
12.3.2	Tool Overview	116
12.3.3	Home Page	118
12.3.4	Maps	121
12.3.5	Queues	125
12.3.6	Topics	126

12.3.7 MultiMaps	127
12.3.8 Executors	127
12.3.9 Members	128
12.3.10 Scripting	129
12.3.11 Console	130
12.3.12 Alerts	130
12.3.13 Administration	134
12.3.14 Time Travel	135
12.3.15 Documentation	135
13 Security	137
13.1 Socket Interceptor	137
13.2 Encryption	138
13.3 SSL	138
13.4 Enabling Security for Hazelcast Enterprise	140
13.5 Credentials	140
13.6 ClusterLoginModule	141
13.7 Cluster Member Security	142
13.8 Native Client Security	143
13.8.1 Authentication	143
13.8.2 Authorization	144
13.8.3 Permissions	145
14 Performance	149
14.1 Data Affinity	149
15 WAN	153
15.1 WAN Replication	153
16 Configuration	155
16.1 Network Configuration	156
16.1.1 Configuring TCP/IP Cluster	156
16.1.2 Specifying Network Interfaces	157
16.1.3 EC2 Auto Discovery	157
16.1.4 IPv6 Support	158
16.1.5 Restricting Outbound Ports	159
16.2 Partition Group Configuration	159
16.3 Listener Configurations	161
16.4 Wildcard Configuration	163
16.5 Advanced Configuration Properties	164
16.5.1 Declarative Configuration	164

16.5.2 Programmatic Configuration	164
16.5.3 System Property	164
16.6 Logging Configuration	166
16.7 Setting License Key	167
17 Frequently Asked Questions	169
17.1 Why 271 as the default partition count	169
17.2 How do nodes discover each other	169
17.3 What happens when a node goes down	169
17.4 How do I choose keys properly	169
17.5 How do I reflect value modifications	170
17.6 How do I test my Hazelcast cluster	170
17.7 How do I create separate clusters	172
17.8 When RuntimeException is thrown	172
17.9 When ConcurrentModificationException is thrown?	173
17.10 How is Split-Brain syndrome handled	173
17.11 Does Hazelcast support thousands of clients	174
17.12 How do you give support	174
17.13 Does Hazelcast persist	175
17.14 Can I use Hazelcast in a single server	175
17.15 How can I monitor Hazelcast	175
17.16 How can I see debug level logs	175

Chapter 1

Introduction

1.1 Hazelcast Overview

Hazelcast is a clustering and highly scalable data distribution platform for Java. Hazelcast helps architects and developers to easily design and develop faster, highly scalable and reliable applications for their businesses.

- Distributed implementations of `java.util.{Queue, Set, List, Map}`
- Distributed implementation of `java.util.concurrent.ExecutorService`
- Distributed implementation of `java.util.concurrent.locks.Lock`
- Distributed Topic for publish/subscribe messaging
- Transaction support and J2EE container integration via JCA
- Distributed listeners and events
- Support for cluster info and membership events
- Dynamic HTTP session clustering
- Dynamic clustering
- Dynamic scaling to hundreds of servers
- Dynamic partitioning with backups
- Dynamic fail-over
- A very small JAR file
- Super simple to use; include a single jar
- Super fast; thousands of operations per sec.
- Super efficient; very nice to CPU and RAM

Hazelcast is pure Java. JVMs that are running Hazelcast will dynamically cluster. Although by default Hazelcast will use multicast for discovery, it can also be configured to only use TCP/IP for environments where multicast is not available or preferred ([Click here for more info](#)). Communication among cluster members is always TCP/IP with Java NIO beauty. Default configuration comes with 1 backup so if one node fails, no data will be lost. It is as simple as using `java.util.{Queue, Set, List, Map}`. Just add the `hazelcast.jar` into your classpath and start coding.

1.2 Why Hazelcast?

A Glance at Traditional Data Persistence

Data is the essence in software systems and in conventional architectures, relational database persists and provides access to data. Basically, applications are talking directly with a database which has its backup as another machine. To increase the performance capabilities in a conventional architecture, a faster machine is required or utilization of the current resources should be tuned. This leads to a large amount of money or manpower.

Then, there is the idea of keeping copies of data next to the database. This is performed using technologies like external key-value storages or second level caching. Purpose is to protect the database from excessive loads. However, when the database is saturated or if the applications perform mostly “put” operations, this approach is of no use, since it insulates the database only from the “get” loads. Even if the applications heavily perform “get”s, then there appears a consistency issue: when data is changed within the database, what is the reaction of local data cache, how these changes are handled? This is the point where concepts like time-to-live (TTL) or write-through come as solutions.

However, for example in the case of caches having entries with TTL; if the frequency of access to an entry is less than TTL, again there is no use. On the other hand, in the case of write through caches; if there are more than one of these caches in a cluster, then we have again consistency issues between those. This can be avoided by having the nodes communicating with each other so that entry invalidations can be propagated.

We can conclude that an ideal cache would combine TTL and write through features. And, there are several cache servers and in-memory database solutions in this field. However, those are stand-alone single instances with a distribution mechanism to an extent provided by other technologies. This brings us back to square one: we would experience saturation or capacity issues if the product is a single instance or if consistency is not provided by the distribution.

And, there is Hazelcast

Hazelcast, a brand new approach to data, is designed around the concept of distribution. Data is shared around the cluster for flexibility and performance. It is an in-memory data grid for clustering and highly scalable data distribution.

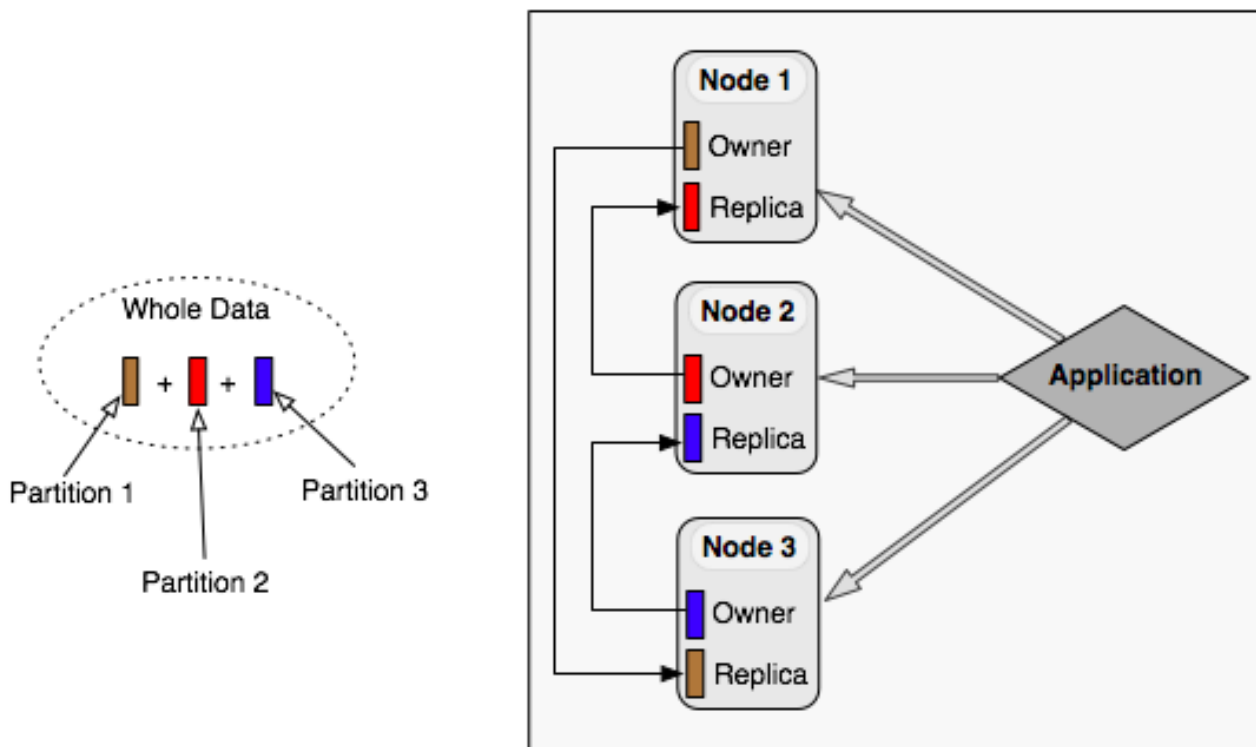
One of the main features of Hazelcast is not having a master node. Each node in the cluster is configured to be the same in terms of functionality. The oldest node manages the cluster members, i.e. automatically performs the data assignment to nodes. When a new node joins to the cluster or a node goes down, this data assignment is repeated across the nodes and the data distribution comes to a balance again. Therefore, getting Hazelcast up and running is simple as the nodes are discovered and clustered automatically at no time.

Another main feature is the data being persisted entirely in-memory. This is fast. In the case of a failure, such as a node crash, no data will be lost since Hazelcast keeps copies of data across all the nodes of cluster. Data is kept in partition slices and each partition slice is owned by a node and backed up on another node. Please see the illustration below.

As it can be seen in the feature list given in [Hazelcast Overview](#) section, Hazelcast supports a number of distributed collections and features. Data can be loaded from various sources into diversity of structures, messages can be sent across a cluster, locks can be put to take measures against concurrent operations and events happening in a cluster can be listened.

Hazelcast’s Distinctive Strengths

- It is open source.
- It is a small JAR file. You do not need to install a software.
- It is a library, it does not impose an architecture on Hazelcast users.
- It provides out of the box distributed data structures (i.e. Map, Queue, MultiMap, Topic, Lock, Executor, etc.).
- There is no “master” in Hazelcast cluster; each node in the cluster is configured to be functionally the same.
- When the size of your data to be stored and processed in memory increases, just add nodes to the cluster to increase the memory and processing power.
- Data is not the only thing which is distributed, backups are distributed, too. As can be noticed, this is a big benefit when a node in the cluster is gone (e.g. crashes). Data will not be lost.



- Nodes are always aware of each other (and they communicate) unlike the traditional key-value caching solutions.
- And, it can be used as a platform to build your own distributed data structures using the Service Programming Interface (SPI), if you are not happy with the ones provided.

And still evolving. Hazelcast has a dynamic open source community enabling it to be continuously developed. Since it has a very clean API that implements Java interfaces, its usage is simple especially for Java developers. These along with the above features make Hazelcast easy to use and simple to manage.

As an in-memory data grid provider, Hazelcast is a perfect fit:

- For data analysis applications requiring big data processings by partitioning the data,
- For retaining frequently accessed data in the grid,
- To be a primary data store for applications with utmost performance, scalability and low-latency requirements,
- For enabling publish/subscribe communication between applications,
- For applications to be run in distributed and scalable cloud environments,
- To be a highly available distributed cache for applications,
- As an alternative to Coherence, Gemfire and Terracotta.

1.3 Getting Started

1.3.1 Installing Hazelcast

It is more than simple to start enjoying Hazelcast:

- Download `hazelcast-<version>.zip` from www.hazelcast.org.

- Unzip `hazelcast-<version>.zip` file.
- Add `hazelcast-<version>.jar` file into your classpath.

That is all.

Alternatively, Hazelcast can be found in the standard Maven repositories. So, if your project uses Maven, you do not need to add additional repositories to your `pom.xml`. Just add the following lines to the `pom.xml`:

```
<dependencies>
  <dependency>      <groupId>com.hazelcast</groupId>
    <artifactId>hazelcast</artifactId>      <version>3.2</version>
  </dependency> </dependencies>
```

1.3.2 Starting the Cluster and Client

Having `hazelcast-<version>.jar` added to your classpath, it is time to get started.

In this short tutorial, we will:

1. Create a simple Java application using Hazelcast distributed map and queue.
2. Then, we will run our application twice to have two nodes (JVMs) clustered.
3. And, connect to our cluster from another Java application by using Hazelcast Native Java Client API.

Let's begin.

- Following code will start the first node and create and use `customers` map and queue.

```
import com.hazelcast.config.Config;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

import java.util.Map;
import java.util.Queue;

public class GettingStarted {

    public static void main(String[] args) {
        Config cfg = new Config();
        HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);
        Map<Integer, String> mapCustomers = instance.getMap("customers");
        mapCustomers.put(1, "Joe");
        mapCustomers.put(2, "Ali");
        mapCustomers.put(3, "Avi");

        System.out.println("Customer with key 1: " + mapCustomers.get(1));
        System.out.println("Map Size: " + mapCustomers.size());

        Queue<String> queueCustomers = instance.getQueue("customers");
        queueCustomers.offer("Tom");
        queueCustomers.offer("Mary");
        queueCustomers.offer("Jane");
        System.out.println("First customer: " + queueCustomers.poll());
        System.out.println("Second customer: " + queueCustomers.peek());
        System.out.println("Queue size: " + queueCustomers.size());
    }
}
```

- Run this class second time to get the second node started. Have you seen they formed a cluster? You should see something like this:

```
Members [2] {  
    Member [127.0.0.1:5701]  
    Member [127.0.0.1:5702] this  
}
```

- Now, add `hazelcast-client-<version>.jar` to your classpath, too. This is required to be able to use a Hazelcast client.
- Following code will start a Hazelcast Client, connect to our two node cluster and print the size of our customers map.

```
package com.hazelcast.test;  
  
import com.hazelcast.client.config.ClientConfig;  
import com.hazelcast.client.HazelcastClient;  
import com.hazelcast.core.HazelcastInstance;  
import com.hazelcast.core.IMap;  
  
public class GettingStartedClient {  
  
    public static void main(String[] args) {  
        ClientConfig clientConfig = new ClientConfig();  
        HazelcastInstance client = HazelcastClient.newHazelcastClient(clientConfig);  
        IMap map = client.getMap("customers");  
        System.out.println("Map Size:" + map.size());  
    }  
}
```

- When you run it, you will see the client properly connecting to the cluster and printing the map size as **3**.

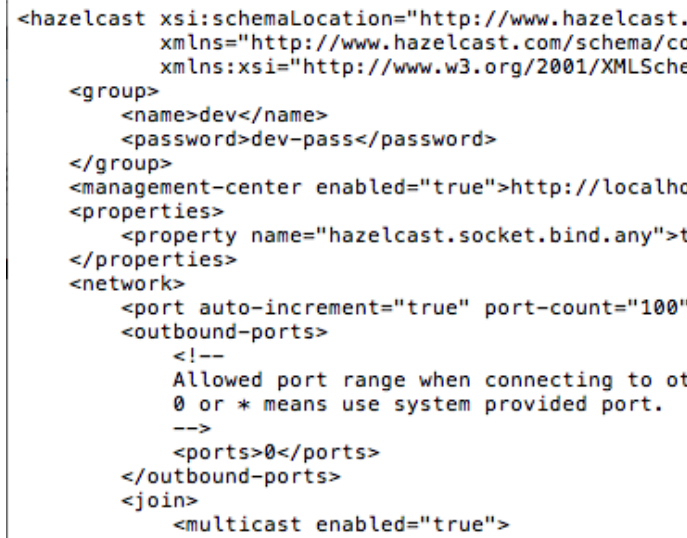
Hazelcast also offers a tool, **Management Center**, that enables monitoring your cluster. To be able to use it, deploy the `mancenter-<version>.war` included in the ZIP file to your web server. You can use it to monitor your maps, queues, other distributed data structures and nodes. Please see [Management Center](#) for usage explanations.

Related Information

You can also check the video tutorials [here](#).

1.3.3 Configuring Hazelcast

When you download and unzip `hazelcast-version.zip` you will see the `hazelcast.xml` in `/bin` folder. This is the



```
<hazelcast xsi:schemaLocation="http://www.hazelcast.com/schema/hazelcast-5.0
  xmlns="http://www.hazelcast.com/schema/hazelcast-5.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <group>
    <name>dev</name>
    <password>dev-pass</password>
  </group>
  <management-center enabled="true">http://localhost:8080</management-center>
  <properties>
    <property name="hazelcast.socket.bind.any">true</property>
  </properties>
  <network>
    <port auto-increment="true" port-count="100">
      <outbound-ports>
        <!--
          Allowed port range when connecting to other nodes.
          0 or * means use system provided port.
        -->
        <ports>0</ports>
      </outbound-ports>
    </port>
    <join>
      <multicast enabled="true">

```

configuration XML file for Hazelcast, a part of which is shown below.

For most of the users, default configuration should be fine. If not, you can tailor this XML file according to your needs by adding/removing/modifying properties (Declarative Configuration). Please refer to [Configuration Properties](#) for details. Besides declarative configuration, you can configure your cluster programmatically (Programmatic Configuration). Just instantiate a `Config` object and add/remove/modify properties.

Related Information

Please refer to [Configuration](#) chapter for more information.

1.4 Deployment Types

Basically, Hazelcast can be deployed in two types: as a Peer-to-Peer cluster or Client/Server cluster.

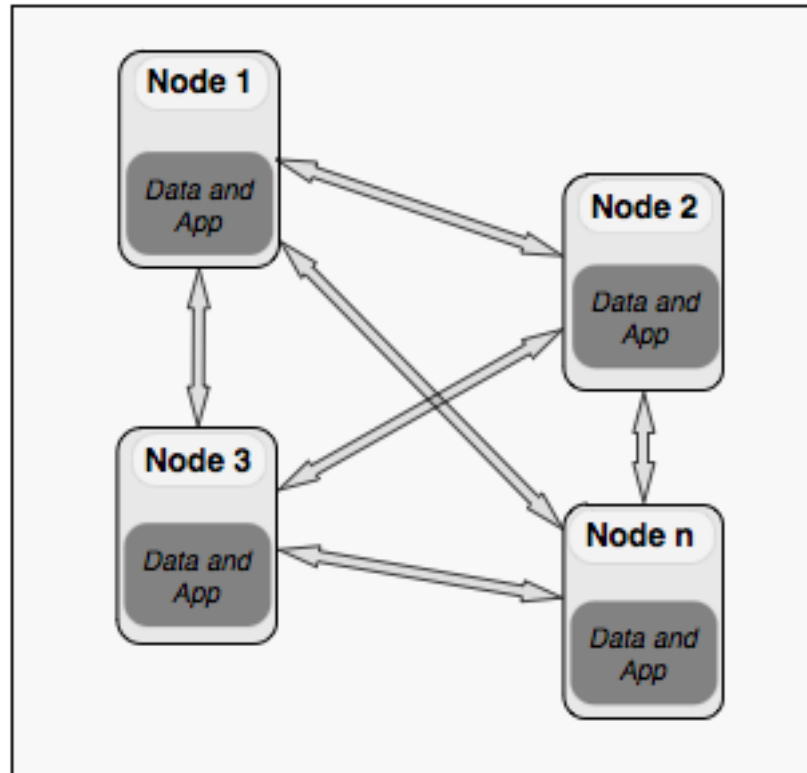
If you have an application whose main focal point is asynchronous or high performance computing and lots of task executions, then Peer-to-Peer deployment is the most useful. In this type, nodes include both the application and data, see the below illustration.

If you do not prefer running tasks in your cluster but storing data, you can have a cluster of server nodes that can be independently created and scaled. Your clients communicate with these server nodes to reach to the data on them. See the below illustration.

1.5 Use Cases

Some example usages are listed below. Hazelcast can be used:

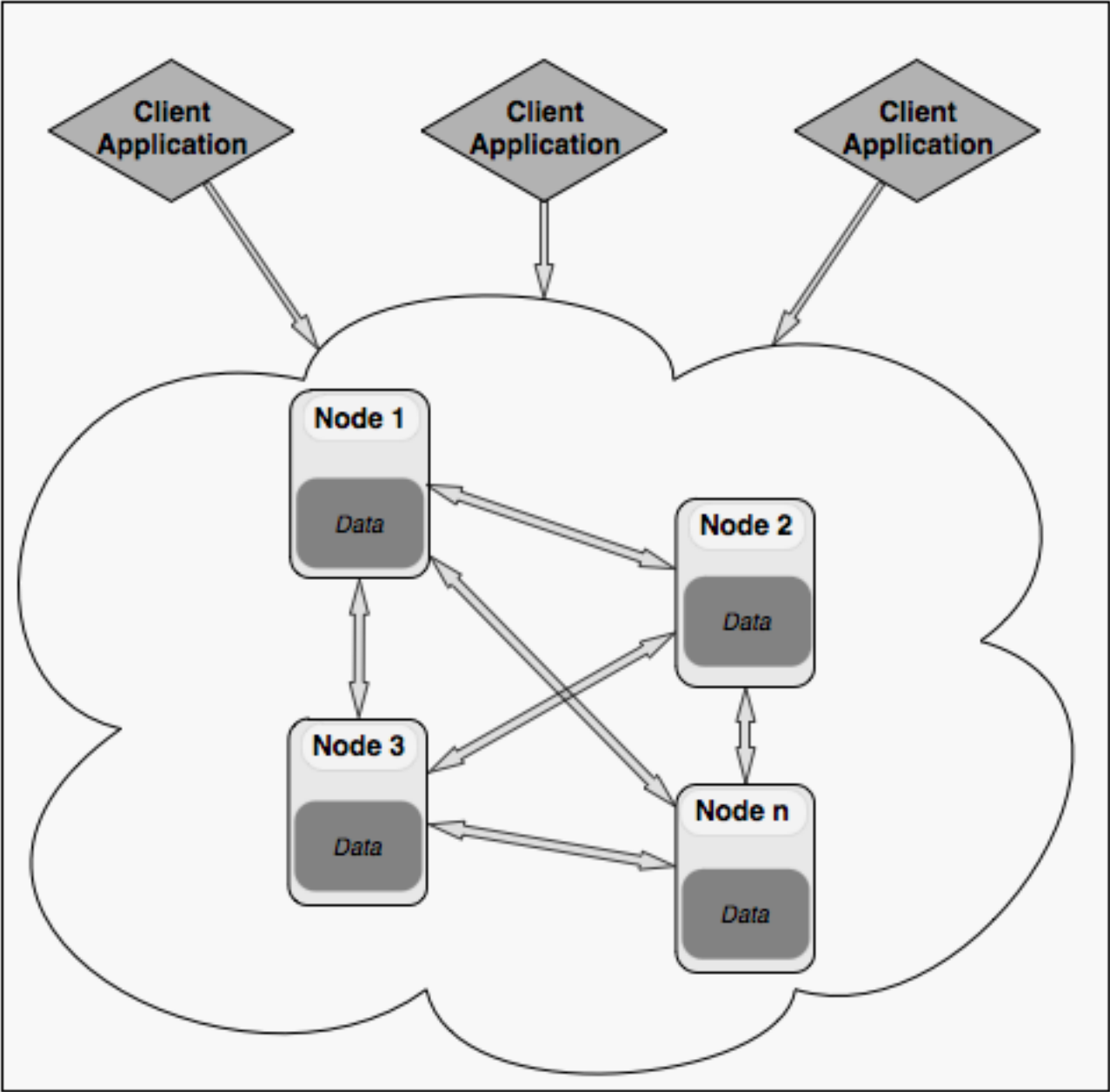
- To share server configuration/information to see how a cluster performs,
- To cluster highly changing data with event notifications (e.g. user based events) and to queue and distribute background tasks,
- As a simple Memcache with near cache,
- As a cloud-wide scheduler of certain processes that need to be performed on some nodes.
- To share information (user information, queues, maps, etc.) on the fly with multiple nodes in different installations under OSGI environments,



- To share thousands of keys in a cluster where there is a web service interface on application server and some validation,
- As a distributed topic (publish/subscribe server) to build scalable chat servers for smartphones,
- As a front layer for Cassandra back end,
- To distribute user object states across the cluster, to pass messages between objects and to share system data structures (static initialization state, mirrored objects, object identity generators),
- As a multi-tenancy cache where each tenant has its own map,
- To share datasets (e.g. table-like data structure) to be used by applications,
- To distribute the load and collect status from Amazon EC2 servers where front-end is developed using, for example, Spring framework,
- As a real time streamer for performance detection,
- As a storage for session data in web applications (enables horizontal scalability of the web application).

1.6 Resources

- Hazelcast source code can be found at [Github/Hazelcast](#).
- Hazelcast API can be found at [Hazelcast.org](#).
- More use cases and resources can be found at [Hazelcast.com](#).
- Questions and discussions can be post in [Hazelcast mail group](#).



Chapter 2

What's New in Hazelcast 3.2

2.1 Release Notes

2.1.1 New Features

This section provides the new features introduced with Hazelcast 3.2 release.

- **NIO Client:** New architecture based on NIO introduced to support more scalable and concurrent client usage.
- **MapReduce Framework:** MapReduce implemented for your key-value collections that need to be reduced by grouping the keys. Please see [the interview](#) and [MapReduce](#) section.
- **Order/Limit Support:** Now you can order and limit results returned by queries performed on Hazelcast Distributed Map.
- **C++ Client:** Native C++ client developed for C++ users which can connect to a Hazelcast cluster and realize almost all operations that a node can perform. Please see [Native Clients](#).
- **C# Client:** Also, Native C# client that has a very similar API with Native Java client developed. Please see [Native Clients](#).

2.1.2 Improvements

This section provides the improvements performed for Hazelcast 3.2 release.

- Size of a distributed queue via REST API can be returned. [\[#1809\]](#)
- InitialLoadMode configuration parameter (having Lazy and Eager as values) added to MapStoreConfig. [\[#1751\]](#)
- Tagging support for Executor Service introduced such that nodes can be tagged for IExecutorService. [\[1457\]](#)
- `getForUpdate()` operation for transactional map introduced. [\[#1033\]](#)
- Entry processor can run on a set of keys with the introduction of `executeOnKeys(keys,entryprocessor)` method for IMap. [\[1423\]](#)
- `getNearCacheStats()` introduced. Statistics for near cache can be retrieved. [\[#30\]](#)

Please see the list of all enhancement issues [here](#).

2.1.3 Fixes

3.2.2 Fixes

This section lists issues solved for Hazelcast 3.2.2 release.

- Client security callable fix: <https://github.com/hazelcast/hazelcast/pull/2561>
- Updating a key in a transaction gives listeners an `entryAdded()` callback instead of `entryUpdated()` <https://github.com/hazelcast/hazelcast/issues/2542>
- Client ssl engine doesn't need `keyStore` and `keyStorePassword` <https://github.com/hazelcast/hazelcast/pull/2525>
- Added support for Mapper, Combiner, Reducer, KeyValueSource to implement `HazelcastInstanceAware` <https://github.com/hazelcast/hazelcast/pull/2502>
- Fixed alter function <https://github.com/hazelcast/hazelcast/pull/2496>
- Return cached value upon `IMap.get()` if near cache is enabled <https://github.com/hazelcast/hazelcast/pull/2482>
- Exception initialising `hz:client` <https://github.com/hazelcast/hazelcast/issues/2480>
- Fixed portable serialization between different services versions <https://github.com/hazelcast/hazelcast/pull/2478>
- Resolves a data race in the client proxy that can lead to an NPE. <https://github.com/hazelcast/hazelcast/pull/2474>
- Fixed partition group hostname matching <https://github.com/hazelcast/hazelcast/pull/2470>
- Client shutdown issue: Improve logging <https://github.com/hazelcast/hazelcast/issues/2442>
- Unnecessary synchronized lock when invoking `com.hazelcast.instance.LifecycleServiceImpl.isRunning()` <https://github.com/hazelcast/hazelcast/issues/2454>
- If `MapStoreFactory` throws exception, instance hangs <https://github.com/hazelcast/hazelcast/issues/2445>
- Semaphore is given to the thread that is coming late <https://github.com/hazelcast/hazelcast/issues/2443>
- Lots of exceptions when shutting down connection <https://github.com/hazelcast/hazelcast/issues/2441>
- Migration fails when statistics are disabled <https://github.com/hazelcast/hazelcast/issues/2436>
- 3.2.1 regression: nested transactions are not caught and prevented. <https://github.com/hazelcast/hazelcast/issues/2404>
- Client proxy init synced <https://github.com/hazelcast/hazelcast/pull/2376>
- Fixes hostname matching problem when interface has wildcards <https://github.com/hazelcast/hazelcast/pull/2398>
- Fix weblogic shutdown backport <https://github.com/hazelcast/hazelcast/pull/2391>
- `NotWritablePropertyException` connectionAttemptLimit with ssl client config <https://github.com/hazelcast/hazelcast/issues/2390>
- Map-Reduce Operation fails, when another instance tries to form a cluster with an instance running a map reduce task <https://github.com/hazelcast/hazelcast/issues/2354>
- `EntryEvent` `getMember` returning null when a node leaves the cluster <https://github.com/hazelcast/hazelcast/issues/2358>
- `NullPointerException` in `Bundle Activator` <https://github.com/hazelcast/hazelcast/issues/2489>

Please see [here](#) for the full list of solved issues.

3.2.1 Fixes

This section lists issues solved for Hazelcast 3.2.1 release.

- JCA problems have been fixed [#2025](#).
- C++ client compilation problems are fixed.
- Redo problem about Java dummy client is fixed.
- Round robin load balancer of Java client is improved.
- Initial timeout is for the initial connections in Java clients.
- Wildcard configuration improvement in near cache configuration.
- Unneeded serializations in `EntryProcessor` should be removed when the object format is *In-Memory* [#2139](#).
- Race condition in near cache has been solved, immediate invalidation of local near cache was needed [#2163](#).
- Predicate issue seen in transactions is solved.
- Comparator issue in map eviction is solved.
- Map eviction part has been refactored due to a race condition on map listener [#2324](#).
- Stale data problem in client near cache has been solved [#2065](#).
- Many checkstyle and findbugs issues are solved.

Please see [here](#) for the full list of solved issues.

3.2 Fixes

This section lists issues solved for Hazelcast 3.2 release.

- `LocalMapStats.getNearCacheStats()` can return null when it is called before a map get that calls `initNearCache()`. [\[#2009\]](#)

- `testMapWithIndexAfterShutDown` fails in OpenJDK. [\[#2001\]](#)
- Portable Serialization needs objects to be shared between client and server. [\[#1957\]](#)
- Near cache entries should be locally invalidate on `IMap.executeOnKey()`. [\[#1951\]](#)
- `OperationTimeoutException` is thrown when executing task that runs longer than `hazelcast.operation.call.timeout.ms`. [\[#1949\]](#)
- `MapStore#store` was called when executing `AbstractEntryProcessor` on backup. [\[#1940\]](#)
- After an `OperationTimeoutException` is thrown from `ILock.tryLock()` (and after the system is back in a normal state), the named lock remains locked. [\[#1937\]](#)
- Hazelcast client needs `OutOfMemoryErrorDispatcher`. [\[#1933\]](#)
- Near Cache: Caching of local entries may lead to race condition. [\[#1905\]](#)
- After key owner node dies, it takes too much time for threads to wakeup from `condition.await()`. [\[#1879\]](#)
- Possible improvements/fixes for NearCache. [\[#1863\]](#)
- `MultipleEntryBackupOperation` does not handle deletion of entries. [\[#1854\]](#)
- If topics are created/destroyed, then the statistics for that topic are not destroyed and this can cause a memory leak. [\[#1847\]](#)
- `PartitionService` backup/replication fixes. [\[#1840\]](#)
- Cached null values remain in near cache after `evict` is called. [\[#1829\]](#)
- `NullPointerException` in `MultiMap` when the service is shutdown before the migration is processed. [\[#1823\]](#)
- Network interruption causes node to continually warn with `WrongTargetException`. [\[#1815\]](#)
- `DefaultRecordStore#removeAll` should be modified so that it keeps “key objects to delete” as a list, not a set. [\[#1795\]](#)
- Very long `operation.run()` call stack especially when high partition count is used. [\[#1745\]](#)
- When executing an entry processor with an index aware predicate, the index is not used, instead the predicate is applied to the entire entry set. [\[#1719\]](#)
- When one node goes down in a cluster with 2 nodes (where near cache is enabled), `containsKey` call hangs in the second node. [\[#1688\]](#)
- When deleting an entry from an entry processor by setting the value to null, it is not removed from the backup store. [\[#1687\]](#)
- Client calls executed at server side cause unwanted (de)serialization. [\[#1669\]](#)
- In `TrackableJobFuture.get(long, TimeUnit)`, there is a 100 ms of sleep-spin while waiting for the result of a MapReduce task to be set. [\[#1648\]](#)
- If `storeAll` takes much time and if instance terminates while map store is running, data can be lost. [\[#1644\]](#)
- A missing Spring 4 Cache method added to hazelcast-spring package (namely `public T get(Object key, Class type)`). [\[#1627\]](#)
- When eviction tasks are canceled, `scheduledExecutorService` is not cleaned. [\[#1595\]](#)
- `storeAll()` with new value for the same key should not be executed until any previous `storeAll()` operations with the same key are not completed. [\[#1592\]](#)
- When using native client to interact with Hazelcast cluster, some JMX MBean attribute values on cluster nodes are not set/updated. [\[#1576\]](#)
- `IMap.getAll(keys)` method does not read from near cache. [\[#1532\]](#)
- Near Cache `cache-local-entries` attribute is missing in `hazelcast-spring-3.2` XSD. [\[#1524\]](#)
- Exception while executing script in OpenJDK 8. [\[#1518\]](#)
- Infinite waiting on merge operations when cluster shuts down. [\[#1504\]](#)
- Client side socket interceptor is not needed to be `MemberSocketInterceptor`. [\[#1444\]](#)
- Near cache on the local node should be enabled if its `InMemoryFormat` is different from that of the map. [\[#1438\]](#)
- Async `EntryProcessor` does not deserialize the value before it is called back. [\[#1433\]](#)
- A submitted task cannot be canceled via the native client. [\[#1394\]](#)
- `executeOnKeys(keys,entryprocessor)` introduced on `IMap`. With this feature entry processor can be run on a set of keys. [\[#1339\]](#)
- FINEST logging should be guarded where appropriate. [\[#1332\]](#)
- False errors reported in Eclipse due to schema definition. [\[#1330\]](#)
- Index based operations are not synchronized with partition changes. [\[#1297\]](#)
- Management Center: `InvocationTargetException` in Tomcat console when a node is started and then stopped. [\[#1267\]](#)

- The system property `hazelcast.map.load.chunk.size` is being ignored in Hazelcast 3.1. [#1110]
- Master should fire repartitioning after getting confirmation from nodes. [#1058]
- `SqlPredicate` does not Implement `equals/hashCode`. [#960]
- `DelegatingFuture.isDone` seems to always return false until the method `DelegatingFuture.get` is called. [#850]
- Predicate support for entry processor. [#826]

RC2 Fixes

- `ClientService.getConnectedClients` returns all end points [#1883].
- `MultiMap` is throwing `ConcurrentModificationExceptions` [#1882].
- `executorPoolSize` field of `ClientConfig` cannot be configured using XML [#1867].
- Partition processing cannot be postponed [#1856].
- Memory leak at client endpoints [#1842].
- Errors related to management center configuration on startup [#1821].
- XML parsing error by client [#1818].
- `ClientReAuthOperation` cannot return response without call ID [#1816].
- `MemberAttributeOperationType` should be introduced to remove the dependency to `MapOperationType` [#1811].
- Entry listener removal from `MultiMap` [#1810].
- Change `DefaultRecordStore#removeAll` to keep “key objects to delete” as a list, not a set [#1795].

RC1 Fixes

- `TransactionalMap` does not support `put(K,V,long,TimeUnit)` [#1718].
- Entry is not removed from backup store when it is deleted using entry processor [#1687].
- Possibility of losing data when `MapStore` takes a long time [#1644].
- When eviction tasks are cancelled, `scheduledExecutorService` should be cleaned [#1595].
- A fix related to `StoreAll` is needed in a write-behind scenario [#1592].
- Update problem at map statistics [#1576].
- Exception while executing script in OpenJDK 8 [#1518].
- `StackOverflowError` at `AndResultSet` [#1501].
- Near Cache using `InMemoryFormat.OBJECT` also for local node [#1438].
- Async entry processor is not deserializing the value before returning [#1433].
- Distributed Executor; `Future Cancel` is not working [#1394].
- `HazelcastInstanceFactory$InstanceFuture.get()` never returns when `newHazelcastInstance()` method fails/throws exception [#1253].
- Changes for `Vertex` on Openshift [#1176].
- Serialization should be performed after database interaction for `MapStore` [#1115].
- System property related to chunk size is passed over in Hazelcast 3.1 [#1110].
- Map backups lack eviction of some specific data [#1085].
- `DelegatingFuture.isDone` always returns false until `get` is called [#850].
- Predicate support for entry processor [#826].
- Full replication of Maps should be performed [#360].

2.1.4 Known Issues & Workarounds

Please see [here](#) for the known issues.

2.2 Upgrading from 2.x versions

In this section, we list the changes what users should take into account before upgrading to latest Hazelcast from earlier versions.

- **Removal of deprecated static methods:** The static methods of Hazelcast class reaching Hazelcast data components have been removed. The functionality of these methods can be reached from HazelcastInstance interface. Namely you should replace following:

```
Map<Integer, String> mapCustomers = Hazelcast.getMap("customers");
```

with

```
HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);
// or if you already started an instance
// HazelcastInstance instance = Hazelcast.getHazelcastInstanceByName("instance1");
Map<Integer, String> mapCustomers = instance.getMap("customers");
```

- **Removal of lite members:** With 3.0 there will be no member type as lite member. As 3.0 clients are smart client that they know in which node the data is located, you can replace your lite members with native clients.
- **Renaming “instance” to “distributed object”:** Before 3.0 there was a confusion for the term “instance”. It was used for both the cluster members and the distributed objects (map, queue, topic, etc. instances). Starting 3.0, the term instance will be only used for Hazelcast instances, namely cluster members. We will use the term “distributed object” for map, queue, etc. instances. So you should replace the related methods with the new renamed ones. As 3.0 clients are smart client that they know in which node the data is located, you can replace your lite members with native clients.

```
public static void main(String[] args) throws InterruptedException {
    Config cfg = new Config();
    HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
    IMap map = hz.getMap("test");
    Collection<Instance> instances = hz.getInstances();
    for (Instance instance : instances) {
        if(instance.getInstanceType() == Instance.InstanceType.MAP) {
            System.out.println("there is a map with name:"+instance.getId());
        }
    }
}
```

with

```
public static void main(String[] args) throws InterruptedException {
    Config cfg = new Config();
    HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
    IMap map = hz.getMap("test");
    Collection<DistributedObject> distributedObjects = hz.getDistributedObjects();
    for (DistributedObject distributedObject : distributedObjects) {
        if(distributedObject instanceof IMap)
            System.out.println("there is a map with name:"+distributedObject.getName());
    }
}
```

- **Package structure change:** PartitionService has been moved to package com.hazelcast.core from com.hazelcast.partition.
- **Listener API change:** Before 3.0, removeListener methods was taking the Listener object as parameter. But, it causes confusion as same listener object may be used as parameter for different listener registrations. So we have changed the listener API. addListener methods return you an unique ID and you can remove listener by using this ID. So you should do following replacement if needed:

```
IMap map = instance.getMap("map");
map.addEntryListener(listener, true);
map.removeEntryListener(listener);
```

with

```
IMap map = instance.getMap("map");
String listenerId = map.addEntryListener(listener, true);
map.removeEntryListener(listenerId);
```

- **IMap changes:** - `tryRemove(K key, long timeout, TimeUnit timeunit)` returns boolean indicating whether operation is successful.
- `'tryLockAndGet(K key, long time, TimeUnit timeunit)'` is removed.
- `'putAndUnlock(K key, V value)'` is removed.
- `'lockMap(long time, TimeUnit timeunit)'` and `'unlockMap()'` are removed
- `'getMapEntry(K key)'` is renamed as `'getEntryView(K key)'`. The returned object's type, `MapEntry` c
- There is no predefined names for merge policies. You just give the full class name of the merge

```
“‘xml
```

```
com.hazelcast.map.merge.PassThroughMergePolicy ““
```

Also MergePolicy interface has been renamed to MapMergePolicy and also returning null from the implemented `merge()` method causes the existing entry to be removed.

- **IQueue changes:** There is no change on IQueue API but there are changes on how IQueue is configured. With Hazelcast 3.0 there will not be backing map configuration for queue. Settings like backup count will be directly configured on queue config. For queue configuration details, please see Distributed Queue page.
- **Transaction API change:** In Hazelcast 3.0, transaction API is completely different. Please see [Distributed Transactions](#).
- **ExecutorService API change:** Classes `MultiTask` and `DistributedTask` have been removed. All the functionality is supported by the newly presented interface `IExecutorService`. Please see Distributed Execution.
- **LifeCycleService API** has been simplified. `pause()`, `resume()`, `restart()` methods have been removed.
- `AtomicNumber` class has been renamed to **IAAtomicLong**.
- **ICountDownLatch** `await()` operation has been removed. We expect users to use `await()` method with timeout parameters.
- **ISemaphore API** has been substantially changed. `attach()`, `detach()` methods have been removed.
- In 2.x releases, the default value for *max-size* eviction policy was **cluster_wide_map_size**. In 3.x releases, default is **PER_NODE**. After upgrading, the *max-size* should be set according to this new default, if it is not changed. Otherwise, it is likely that `OutOfMemory` exception may be thrown.

2.3 Document Revision History

Chapter	Section	Description
All		Chapters re-outlined.
Chapter 1 - Introduction	All	Sections enhanced. <i>Hazelcast Overview</i> , <i>Why Hazelcast?</i> ,
Chapter 2 - What's New in Hazelcast 3.2		Section <i>Upgrading from 2.x versions</i> updated by adding a
Chapter 3 - Distributed Data Structures	Persistence	- Information related to <code>MapStoreFactory</code> and <code>MapLoader</code>
	Topic, Set, List	Sections enhanced.
Chapter 6 - Distributed Query	Paging Predicate	Added as a new section explaining the order/limit support

Chapter	Section	Description
Chapter 10 - Clients	MapReduce	Added as a new section.
	Native Clients	Thread count explanation updated.
	Java Client	Improved by adding parameter explanations.
	C++ Client	Added as a new section.
	REST Client	Improved by adding more operation explanations.
	C# Client	Added as a new section.
Chapter 12 - Management	Management Center	Whole chapter updated by adding new screenshots and co
Chapter 13 - Security		<i>Socket Interceptor</i> , <i>Encryption</i> and <i>SSL</i> sections previousl
Chapter 17 - FAQ		Added as a new chapter.

Chapter 3

Distributed Data Structures

Common Features of all Hazelcast Data Structures:

- Data in the cluster is almost evenly distributed (partitioned) across all nodes. So each node carries $\sim (1/n * \text{total-data}) + \text{backups}$, n being the number of nodes in the cluster.
- If a member goes down, its backup replica that also holds the same data, will dynamically redistribute the data including the ownership and locks on them to remaining live nodes. As a result, no data will get lost.
- When a new node joins the cluster, new node takes ownership(responsibility) and load of -some- of the entire data in the cluster. Eventually the new node will carry almost $(1/n * \text{total-data}) + \text{backups}$ and becomes the new partition reducing the load on others.
- There is no single cluster master or something that can cause single point of failure. Every node in the cluster has equal rights and responsibilities. No-one is superior. And no dependency on external ‘server’ or ‘master’ kind of concept.

Here is how you can retrieve existing data structure instances (map, queue, set, lock, topic, etc.) and how you can listen for instance events to get notified when an instance is created or destroyed.

```
import java.util.Collection;
import com.hazelcast.config.Config;
import com.hazelcast.core.*;

public class Sample implements DistributedObjectListener {
    public static void main(String[] args) {
        Sample sample = new Sample();

        Config cfg = new Config();
        HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
        hz.addDistributedObjectListener(sample);

        Collection<DistributedObject> distributedObjects = hz.getDistributedObjects();

        for (DistributedObject distributedObject : distributedObjects) {
            System.out.println(distributedObject.getName() + "," + distributedObject.getId());
        }
    }

    @Override
    public void distributedObjectCreated(DistributedObjectEvent event) {
        DistributedObject instance = event.getDistributedObject();
        System.out.println("Created " + instance.getName() + "," + instance.getId());
    }
}
```

```

    }

    @Override
    public void distributedObjectDestroyed(DistributedObjectEvent event) {
        DistributedObject instance = event.getDistributedObject();
        System.out.println("Destroyed " + instance.getName() + "," + instance.getId());
    }
}

```

3.1 Map

Hazelcast will partition your map entries and almost evenly distribute onto all Hazelcast members. Distributed maps have 1 backup by default so that if a member goes down, you do not lose data. Backup operations are synchronous, so when a `map.put(key, value)` returns, it is guaranteed that the entry is replicated to one other node. For the reads, it is also guaranteed that `map.get(key)` returns the latest value of the entry. Consistency is strictly enforced.

```

import com.hazelcast.core.Hazelcast;
import java.util.Map;
import java.util.Collection;
import com.hazelcast.config.Config;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
Map<String, Customer> mapCustomers = hz.getMap("customers");
mapCustomers.put("1", new Customer("Joe", "Smith"));
mapCustomers.put("2", new Customer("Ali", "Selam"));
mapCustomers.put("3", new Customer("Avi", "Noyan"));

Collection<Customer> colCustomers = mapCustomers.values();
for (Customer customer : colCustomers) {
    // process customer
}

```

`HazelcastInstance.getMap()` actually returns `com.hazelcast.core.IMap` which extends `java.util.concurrent.ConcurrentMap` interface. So methods like `ConcurrentMap.putIfAbsent(key,value)` and `ConcurrentMap.replace(key,value)` can be used on distributed map as shown in the example below.

```

import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import java.util.concurrent.ConcurrentMap;

Config cfg = new Config();
HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);

Customer getCustomer (String id) {
    ConcurrentMap<String, Customer> map = instance.getMap("customers");
    Customer customer = map.get(id);
    if (customer == null) {
        customer = new Customer (id);
        customer = map.putIfAbsent(id, customer);
    }
    return customer;
}

```

```

public boolean updateCustomer (Customer customer) {
    ConcurrentMap<String, Customer> map = instance.getMap("customers");
    return (map.replace(customer.getId(), customer) != null);
}

public boolean removeCustomer (Customer customer) {
    ConcurrentMap<String, Customer> map = instance.getMap("customers");
    return map.remove(customer.getId(), customer) );
}

```

All `ConcurrentMap` operations such as `put` and `remove` might wait if the key is locked by another thread in the local or remote JVM. But, they will eventually return with success. `ConcurrentMap` operations never throw `java.util.ConcurrentModificationException`.

Also see:

- [Data Affinity](#).
- [Map Configuration with wildcards](#).

3.1.1 Backups

Hazelcast will distribute map entries onto multiple JVMs (cluster members). Each JVM holds some portion of the data but you do not want to lose data when a member JVM crashes. To provide data safety, Hazelcast allows you to specify the number of backup copies you want to have. That way, data on a JVM will be copied onto other JVM(s). Hazelcast supports both `sync` and `async` backups. `Sync` backups block operations until backups are successfully copied to backup nodes (or deleted from backup nodes in case of remove) and acknowledgements are received. In contrast, `async` backups do not block operations, they are fire & forget and do not require acknowledgements. By default, Hazelcast will have one sync backup copy. If backup count is more than 1, then each member will carry both owned entries and backup copies of other member(s). So for the `map.get(key)` call, it is possible that calling member has backup copy of that key but by default, `map.get(key)` will always read the value from the actual owner of the key for consistency. It is possible to enable backup reads by changing the configuration. Enabling backup reads will give you greater performance.

```

<hazelcast>
...
<map name="default">
  <!--
    Number of sync-backups. If 1 is set as the backup-count for example,
    then all entries of the map will be copied to another JVM for
    fail-safety. Valid numbers are 0 (no backup), 1, 2, 3.
  -->
  <backup-count>1</backup-count>

  <!--
    Number of async-backups. If 1 is set as the backup-count for example,
    then all entries of the map will be copied to another JVM for
    fail-safety. Valid numbers are 0 (no backup), 1, 2, 3.
  -->
  <async-backup-count>1</async-backup-count>

  <!--
    Can we read the local backup entries? Default value is false for
    strong consistency. Being able to read backup data will give you
    greater performance.
  -->
  <read-backup-data>>false</read-backup-data>

```

```

    ...
  </map>
</hazelcast>

```

3.1.2 Eviction

Hazelcast also supports policy based eviction for distributed map. Currently supported eviction policies are LRU (Least Recently Used) and LFU (Least Frequently Used). This feature enables Hazelcast to be used as a distributed cache. If `time-to-live-seconds` is not 0, entries older than `time-to-live-seconds` value will get evicted, regardless of the eviction policy set. Here is a sample configuration for eviction:

```

<hazelcast>
  ...
  <map name="default">
    <!--
      Number of backups. If 1 is set as the backup-count for example,
      then all entries of the map will be copied to another JVM for
      fail-safety. Valid numbers are 0 (no backup), 1, 2, 3.
    -->
    <backup-count>1</backup-count>

    <!--
      Maximum number of seconds for each entry to stay in the map. Entries that are
      older than <time-to-live-seconds> will get automatically evicted from the map.
      Any integer between 0 and Integer.MAX_VALUE. 0 means infinite. Default is 0.
    -->
    <time-to-live-seconds>0</time-to-live-seconds>

    <!--
      Maximum number of seconds for each entry to stay idle in the map. Entries that are
      idle(not touched) for more than <max-idle-seconds> will get
      automatically evicted from the map.
      Entry is touched if get, put or containsKey is called.
      Any integer between 0 and Integer.MAX_VALUE.
      0 means infinite. Default is 0.
    -->
    <max-idle-seconds>0</max-idle-seconds>

    <!--
      Valid values are:
      NONE (no extra eviction, <time-to-live-seconds> may still apply),
      LRU (Least Recently Used),
      LFU (Least Frequently Used).
      NONE is the default.
      Regardless of the eviction policy used, <time-to-live-seconds> will still apply.
    -->
    <eviction-policy>LRU</eviction-policy>

    <!--
      Maximum size of the map. When max size is reached,
      map is evicted based on the policy defined.
      Any integer between 0 and Integer.MAX_VALUE. 0 means
      Integer.MAX_VALUE. Default is 0.
    -->
    <max-size policy="PER_NODE">5000</max-size>

    <!--

```

When max. size is reached, specified percentage of the map will be evicted. Any integer between 0 and 100. If 25 is set for example, 25% of the entries will get evicted.

```
-->
<eviction-percentage>25</eviction-percentage>
</map>
</hazelcast>
```

max-size Policies

Below policies can be used in *max-size* configuration.

1. **PER_NODE**: Max map size per instance.

```
'''
<max-size policy="PER_NODE">5000</max-size>
'''
```

2. **PER_PARTITION**: Max map size per each partition.

```
'''
<max-size policy="PER_PARTITION">27100</max-size>
'''
```

3. **USED_HEAP_SIZE**: Max used heap size in MB (mega-bytes) per JVM.

```
'''
<max-size policy="USED_HEAP_SIZE">4096</max-size>
'''
```

4. **USED_HEAP_PERCENTAGE**: Max used heap size percentage per JVM.

```
'''
<max-size policy="USED_HEAP_PERCENTAGE">75</max-size>
'''
```

3.1.3 Persistence

Hazelcast allows you to load and store the distributed map entries from/to a persistent datastore such as relational database. If a loader implementation is provided, when `get(key)` is called, if the map entry does not exist in-memory, then Hazelcast will call your loader implementation to load the entry from a datastore. If a store implementation is provided, when `put(key,value)` is called, Hazelcast will call your store implementation to store the entry into a datastore. Hazelcast can call your implementation to store the entries synchronously (write-through) with no-delay or asynchronously (write-behind) with delay and it is defined by the `write-delay-seconds` value in the configuration.

If it is write-through, when the `map.put(key,value)` call returns, you can be sure that

- `MapStore.store(key,value)` is successfully called so the entry is persisted.
- In-Memory entry is updated
- In-Memory backup copies are successfully created on other JVMs (if `backup-count` is greater than 0)

If it is write-behind, when the `map.put(key,value)` call returns, you can be sure that

- In-Memory entry is updated
- In-Memory backup copies are successfully created on other JVMs (if `backup-count` is greater than 0)
- The entry is marked as `dirty` so that after `write-delay-seconds`, it can be persisted.

Same behavior goes for the `remove(key)` and `MapStore.delete(key)` methods. If `MapStore` throws an exception, then the exception will be propagated back to the original `put` or `remove` call in the form of `RuntimeException`. When write-through is used, Hazelcast will call `MapStore.store(key,value)` and `MapStore.delete(key)` for each entry update. When write-behind is used, Hazelcast will call `MapStore.store(map)`, and `MapStore.delete(collection)` to do all writes in a single call. Also, note that your `MapStore` or `MapLoader` implementation should not use Hazelcast `Map/Queue/MultiMap/List/Set` operations. Your implementation should only work with your data store. Otherwise, you may get into deadlock situations.

Here is a sample configuration:

```
<hazelcast>
...
<map name="default">
...
  <map-store enabled="true">
    <!--
      Name of the class implementing MapLoader and/or MapStore.
      The class should implement at least of these interfaces and
      contain no-argument constructor. Note that the inner classes are not supported.
    -->
    <class-name>com.hazelcast.examples.DummyStore</class-name>
    <!--
      Number of seconds to delay to call the MapStore.store(key, value).
      If the value is zero then it is write-through so MapStore.store(key, value)
      will be called as soon as the entry is updated.
      Otherwise it is write-behind so updates will be stored after write-delay-seconds
      value by calling Hazelcast.storeAll(map). Default value is 0.
    -->
    <write-delay-seconds>0</write-delay-seconds>
  </map-store>
</map>
</hazelcast>
```

As you know, a configuration can be applied to more than one map using wildcards (Please see [Wildcard Configuration](#)), meaning the configuration is shared among the maps. But, `MapStore` does not know which entries to be stored when there is one configuration applied to multiple maps. To overcome this, Hazelcast provides `MapStoreFactory` interface.

Using this factory, `MapStores` for each map can be created, when a wildcard configuration is used. A sample code is given below.

```
java final Config config = new Config(); final MapConfig mapConfig = config.getMapConfig("*");
final MapStoreConfig mapStoreConfig = mapConfig.getMapStoreConfig(); mapStoreConfig.setFactoryImplementation(
MapStoreFactory<Object, Object>() { @Override public MapLoader<Object, Object> newMapStore(String
mapName, Properties properties) { return null; } });
```

Moreover, if the configuration implements `MapLoaderLifecycleSupport` interface, then the user will have the control to initialize the `MapLoader` implementation with the given map name, configuration properties and the Hazelcast instance. See the below code portion.

```
public interface MapLoaderLifecycleSupport {

    /**
```

```

    * Initializes this MapLoader implementation. Hazelcast will call
    * this method when the map is first used on the
    * HazelcastInstance. Implementation can
    * initialize required resources for the implementing
    * mapLoader such as reading a config file and/or creating
    * database connection.
    */

    void init(HazelcastInstance hazelcastInstance, Properties properties, String mapName);

    /**
     * Hazelcast will call this method before shutting down.
     * This method can be overridden to cleanup the resources
     * held by this map loader implementation, such as closing the
     * database connections etc.
     */
    void destroy();
}

```

3.1.3.1 Initialization on startup

`MapLoader.loadAllKeys` API is used for pre-populating the in-memory map when the map is first touched/used. If `MapLoader.loadAllKeys` returns NULL then nothing will be loaded. Your `MapLoader.loadAllKeys` implementation can return all or some of the keys. You may select and return only the `hot` keys, for instance. Also note that this is the fastest way of pre-populating the map as Hazelcast will optimize the loading process by having each node loading owned portion of the entries.

Moreover, there is `InitialLoadMode` configuration parameter in the class `MapStoreConfig` class. This parameter has two values: `LAZY` and `EAGER`. If `InitialLoadMode` is set as `LAZY`, data is not loaded during the map creation. If it is set as `EAGER`, whole data is loaded while the map is being created and everything becomes ready to use. Also, if you add indices to your map by `MapIndexConfig` class or `addIndex` method, then `InitialLoadMode` is overridden and `MapStoreConfig` behaves as if `EAGER` mode is on.

Here is `MapLoader` initialization flow;

1. When `getMap()` is first called from any node, initialization will start depending on the the value of `InitialLoadMode`. If it is set as `EAGER`, initialization starts. If it is set as `LAZY`, initialization actually does not start but data is loaded at each time a partition loading is completed.
2. Hazelcast will call `MapLoader.loadAllKeys()` to get all your keys on each node
3. Each node will figure out the list of keys it owns
4. Each node will load all its owned keys by calling `MapLoader.loadAll(keys)`
5. Each node puts its owned entries into the map by calling `IMap.putTransient(key,value)`

Warning: If the load mode is `LAZY` and when `clear()`* method is called (which triggers `MapStore.deleteAll()`), Hazelcast will remove **ONLY** the loaded entries from your map and datastore. Since the whole data is not loaded for this case (`LAZY` mode), please note that there may be still entries in your datastore.*

3.1.3.2 Post Processing Map Store:

In some scenarios, you may need to modify the object after storing it into the map store. For example, you can get ID or version auto generated by your database and you need to modify your object stored in distributed map, not to break the sync between database and data grid. You can do that by implementing `PostProcessingMapStore` interface; so the modified object will be put to the distributed map. That will cause an extra step of **Serialization**, so use it just when needed (This explanation is only valid when using `write-through` map store configuration).

Here is an example of post processing map store:

```

class ProcessingStore extends MapStore<Integer, Employee> implements PostProcessingMapStore {
    @Override
    public void store(Integer key, Employee employee) {
        EmployeeId id = saveEmployee();
        employee.setId(id.getId());
    }
}

```

3.1.4 Interceptors

You can add intercept operations and execute your own business logic synchronously blocking the operation. You can change the returned value from a get operation, change the value to be put or cancel operations by throwing exception.

Interceptors are different from listeners as with listeners you just take an action after the operation has been completed. Interceptor actions are synchronous and you can alter the behaviour of operation, change the values or totally cancel it.

IMap API has two methods for adding and removing interceptor to the map:

```

/**
 * Adds an interceptor for this map. Added interceptor will intercept operations
 * and execute user defined methods and will cancel operations if user defined method throw exception.
 *
 *
 * @param interceptor map interceptor
 * @return id of registered interceptor
 */
String addInterceptor(MapInterceptor interceptor);

/**
 * Removes the given interceptor for this map. So it will not intercept operations anymore.
 *
 *
 * @param id registration id of map interceptor
 */
void removeInterceptor(String id);

```

Here is the MapInterceptor interface:

```

public interface MapInterceptor extends Serializable {

    /**
     * Intercept get operation before returning value.
     * Return another object to change the return value of get(..)
     * Returning null will cause the get(..) operation return original value, namely return null if you
     *
     *
     * @param value the original value to be returned as the result of get(..) operation
     * @return the new value that will be returned by get(..) operation
     */
    Object interceptGet(Object value);

    /**
     * Called after get(..) operation is completed.
     *
     *
     * @param value the value returned as the result of get(..) operation
     */
}

```



```

void afterGet(Object value);

/**
 * Intercept put operation before modifying map data.
 * Return the object to be put into the map.
 * Returning null will cause the put(..) operation to operate as expected, namely no interception.
 * Throwing an exception will cancel the put operation.
 *
 *
 * @param oldValue the value currently in map
 * @param newValue the new value to be put
 * @return new value after intercept operation
 */
Object interceptPut(Object oldValue, Object newValue);

/**
 * Called after put(..) operation is completed.
 *
 *
 * @param value the value returned as the result of put(..) operation
 */
void afterPut(Object value);

/**
 * Intercept remove operation before removing the data.
 * Return the object to be returned as the result of remove operation.
 * Throwing an exception will cancel the remove operation.
 *
 *
 * @param removedValue the existing value to be removed
 * @return the value to be returned as the result of remove operation
 */
Object interceptRemove(Object removedValue);

/**
 * Called after remove(..) operation is completed.
 *
 *
 * @param value the value returned as the result of remove(..) operation
 */
void afterRemove(Object value);
}

```

Example Usage:

```

public class InterceptorTest {
    final String mapName = "map";

    @Test
    public void testMapInterceptor() throws InterruptedException {
        Config cfg = new Config();
        HazelcastInstance instance1 = Hazelcast.newHazelcastInstance(cfg);
        HazelcastInstance instance2 = Hazelcast.newHazelcastInstance(cfg);
        final IMap<Object, Object> map = instance1.getMap("testMapInterceptor");
        SimpleInterceptor interceptor = new SimpleInterceptor();
        map.addInterceptor(interceptor);
        map.put(1, "New York");
    }
}

```

```

map.put(2, "Istanbul");
map.put(3, "Tokyo");
map.put(4, "London");
map.put(5, "Paris");
map.put(6, "Cairo");
map.put(7, "Hong Kong");

try {
    map.remove(1);
} catch (Exception ignore) {
}
try {
    map.remove(2);
} catch (Exception ignore) {
}

assertEquals(map.size(), 6);

assertEquals(map.get(1), null);
assertEquals(map.get(2), "ISTANBUL:");
assertEquals(map.get(3), "TOKYO:");
assertEquals(map.get(4), "LONDON:");
assertEquals(map.get(5), "PARIS:");
assertEquals(map.get(6), "CAIRO:");
assertEquals(map.get(7), "HONG KONG:");

map.removeInterceptor(interceptor);
map.put(8, "Moscow");

assertEquals(map.get(8), "Moscow");
assertEquals(map.get(1), null);
assertEquals(map.get(2), "ISTANBUL");
assertEquals(map.get(3), "TOKYO");
assertEquals(map.get(4), "LONDON");
assertEquals(map.get(5), "PARIS");
assertEquals(map.get(6), "CAIRO");
assertEquals(map.get(7), "HONG KONG");
}

static class SimpleInterceptor implements MapInterceptor, Serializable {

    @Override
    public Object interceptGet(Object value) {
        if(value == null)
            return null;
        return value + ":";
    }

    @Override
    public void afterGet(Object value) {
    }

    @Override
    public Object interceptPut(Object oldValue, Object newValue) {
        return newValue.toString().toUpperCase();
    }
}

```

```

@Override
public void afterPut(Object value) {
}

@Override
public Object interceptRemove(Object removedValue) {
    if(removedValue.equals("ISTANBUL"))
        throw new RuntimeException("you can not remove this");
    return removedValue;
}

@Override
public void afterRemove(Object value) {
    // do something
}
}
}

```

3.1.5 Near Cache

Map entries in Hazelcast are partitioned across the cluster. Imagine that you are reading key `k` so many times and `k` is owned by another member in your cluster. Each `map.get(k)` will be a remote operation, meaning lots of network trips. If you have a map that is read-mostly, then you should consider creating a Near Cache for the map so that reads can be much faster and consume less network traffic. All these benefits do not come free. When using near cache, you should consider the following issues:

- JVM will have to hold extra cached data so it will increase the memory consumption.
- If invalidation is turned on and entries are updated frequently, then invalidations will be costly.
- Near cache breaks the strong consistency guarantees; you might be reading stale data.

Near cache is highly recommended for the maps that are read-mostly. Here is a near cache configuration for a map:

```

<hazelcast>
...
<map name="my-read-mostly-map">
...
    <near-cache>
        <!--
            Maximum size of the near cache. When max size is reached,
            cache is evicted based on the policy defined.
            Any integer between 0 and Integer.MAX_VALUE. 0 means
            Integer.MAX_VALUE. Default is 0.
        -->
        <max-size>5000</max-size>
        <!--
            Maximum number of seconds for each entry to stay in the near cache. Entries that are
            older than <time-to-live-seconds> will get automatically evicted from the near cache.
            Any integer between 0 and Integer.MAX_VALUE. 0 means infinite. Default is 0.
        -->
        <time-to-live-seconds>0</time-to-live-seconds>
        <!--
            Maximum number of seconds each entry can stay in the near cache as untouched (not-read).
            Entries that are not read (touched) more than <max-idle-seconds> value will get removed
            from the near cache.
        -->
    </near-cache>
</map>
</hazelcast>

```

```

    Any integer between 0 and Integer.MAX_VALUE. 0 means
    Integer.MAX_VALUE. Default is 0.
-->
<max-idle-seconds>60</max-idle-seconds>

<!--
    Valid values are:
    NONE (no extra eviction, <time-to-live-seconds> may still apply),
    LRU (Least Recently Used),
    LFU (Least Frequently Used).
    NONE is the default.
    Regardless of the eviction policy used, <time-to-live-seconds> will still apply.
-->
<eviction-policy>LRU</eviction-policy>

<!--
    Should the cached entries get evicted if the entries are changed (updated or removed).
    true or false. Default is true.
-->
<invalidate-on-change>true</invalidate-on-change>

<!--
    You may want also local entries to be cached.
    This is useful when in memory format for near cache is different than the map's one.
    By default it is disabled.
-->
<cache-local-entries>>false</cache-local-entries>
</near-cache>
</map>
</hazelcast>

```

Note: Programmatically, near cache configuration is done by using the class [NearCacheConfig](#). And this class is used both in nodes and clients. To create a near cache in a client (native Java client), use the method `addNearCacheConfig` in the class `ClientConfig` (please see [Java Client](#) section). Please note that near cache configuration is specific to the node or client itself, a map in a node may not have near cache configured while the same map in a client may have.

3.1.6 Entry Statistics

Hazelcast keeps extra information about each map entry such as creation time, last update time, last access time, number of hits, version, and this information is exposed to the developer via `IMap.getEntryView(key)` call. Here is an example:

```

import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.EntryView;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
EntryView entry = hz.getMap("quotes").getEntryView("1");
System.out.println("size in memory   : " + entry.getCost());
System.out.println("creationTime    : " + entry.getCreationTime());
System.out.println("expirationTime  : " + entry.getExpirationTime());
System.out.println("number of hits   : " + entry.getHits());
System.out.println("lastAccessedTime: " + entry.getLastAccessTime());
System.out.println("lastUpdateTime  : " + entry.getLastUpdateTime());
System.out.println("version        : " + entry.getVersion());
System.out.println("key           : " + entry.getKey());
System.out.println("value         : " + entry.getValue());

```

3.1.7 In Memory Format

Distributed map has in-memory-format configuration option. By default, Hazelcast stores data into memory in binary (serialized) format. But sometimes, it can be efficient to store the entries in their objects form, especially in cases of local processing like entry processor and queries. Setting in-memory-format in map's configuration, you can decide how the data will be stored in memory. There are below options.

- **BINARY (default)**: This is the default option. The data will be stored in serialized binary format.
- **OBJECT**: The data will be stored in de-serialized form. This configuration is good for maps where entry processing and queries form the majority of all operations and the objects are complex ones, so serialization cost is respectively high. By storing objects, entry processing will not contain the de-serialization cost.

3.2 Queue

Hazelcast distributed queue is an implementation of `java.util.concurrent.BlockingQueue`.

```
import com.hazelcast.core.Hazelcast;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;
import com.hazelcast.config.Config;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
BlockingQueue<MyTask> q = hz.getQueue("tasks");
q.put(new MyTask());
MyTask task = q.take();

boolean offered = q.offer(new MyTask(), 10, TimeUnit.SECONDS);
task = q.poll(5, TimeUnit.SECONDS);
if (task != null) {
    //process task
}
```

FIFO ordering will apply to all queue operations cluster wide. User objects (such as `MyTask` in the example above), that are (en/de)queued have to be `Serializable`. By configuring `max-size` for queue, one can obtain a bounded queue.

There is no batching while iterating over Queue. All items will be copied to local and iteration will occur locally.

A sample configuration is shown below.

```
<hazelcast>
...
<queue name="tasks">
    <!--
        Maximum size of the queue. When queue size reaches the maximum,
        all put operations will get blocked until the queue size
        goes down below the maximum.
        Any integer between 0 and Integer.MAX_VALUE. 0 means Integer.MAX_VALUE. Default is 0.
    -->
    <max-size>10000</max-size>

    <!--
        Number of backups. If 1 is set as the backup-count for example,
        then all entries of the map will be copied to another JVM for
        fail-safety. Valid numbers are 0 (no backup), 1, 2 ... 6.
    -->
```

```

Default is 1.
-->
<backup-count>1</backup-count>

<!--
    Number of async backups. 0 means no backup.
-->
<async-backup-count>0</async-backup-count>

<!--
    QueueStore implementation to persist items.
    'binary' property indicates that storing items will be in binary format
    'memory-limit' property enables 'overflow to store' after reaching limit
    'bulk-load' property enables bulk-loading from store
-->
<queue-store>
  <class-name>com.hazelcast.QueueStore</class-name>
  <properties>
    <property name="binary">false</property>
    <property name="memory-limit">1000</property>
    <property name="bulk-load">250</property>
  </properties>
</queue-store>
</queue>
</hazelcast>

```

3.2.1 Persistence

Hazelcast allows you to load and store the distributed queue entries from/to a persistent datastore such as relational database via a queue-store. If queue store is enabled, each entry added to queue will also be stored at the configured queue store. When the number of items in queue exceeds the memory limit, items will only persisted to queue store, they will not stored in queue memory. Below are the queue store configuration options:

- **Binary:** By default, Hazelcast stores queue items in serialized form in memory and before inserting into datastore, deserializes them. But if you will not reach the queue store from an external application, you can prefer the items to be inserted in binary form. So you get rid of de-serialization step which is a performance optimization. Binary feature is disabled by default.
- **Memory Limit:** This is the number of items after which Hazelcast will just store items to datastore. For example, if memory limit is 1000, then 1001st item will be just put into datastore. This feature is useful when you want to avoid out-of-memory conditions. Default number for memory limit is 1000. If you want to always use memory, you can set it to `Integer.MAX_VALUE`.
- **Bulk Load:** At initialization of queue, items are loaded from QueueStore in bulks. Bulk load is the size of these bulks. By default it is 250.

Below is an example queue store configuration:

```

<queue-store>
  <class-name>com.hazelcast.QueueStoreImpl</class-name>
  <properties>
    <property name="binary">false</property>
    <property name="memory-limit">10000</property>
    <property name="bulk-load">500</property>
  </properties>
</queue-store>

```

3.3 MultiMap

MultiMap is a specialized map where you can associate a key with multiple values. Just like any other distributed data structure implementation in Hazelcast, MultiMap is distributed/partitioned and thread-safe.

```
import com.hazelcast.core.MultiMap;
import com.hazelcast.core.Hazelcast;
import java.util.Collection;
import com.hazelcast.config.Config;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);

// a multimap to hold <customerId, Order> pairs
MultiMap<String, Order> mmCustomerOrders = hz.getMultiMap("customerOrders");
mmCustomerOrders.put("1", new Order ("iPhone", 340));
mmCustomerOrders.put("1", new Order ("MacBook", 1200));
mmCustomerOrders.put("1", new Order ("iPod", 79));

// get orders of the customer with customerId 1.
Collection<Order> colOrders = mmCustomerOrders.get ("1");
for (Order order : colOrders) {
    // process order
}

// remove specific key/value pair
boolean removed = mmCustomerOrders.remove("1", new Order ("iPhone", 340));
```

3.4 Set

Hazelcast Set is distributed and concurrent implementation of `java.util.Set`.

- Hazelcast Set does not allow duplicate elements.
- Hazelcast Set does not preserve the order of elements.
- Hazelcast Set is non-partitioned data structure where values and each backup is represented by its own single partition.
- Hazelcast Set cannot be scaled beyond the capacity of a single machine.
- There is no batching while iterating over Set. All items will be copied to local and iteration will occur locally.
- Equals method implementation of Hazelcast Set uses serialized byte version of objects compared to `java.util.HashSet`.

3.4.1 Sample Set Code

```
import com.hazelcast.core.Hazelcast;
import java.util.Set;
import java.util.Iterator;
import com.hazelcast.config.Config;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);

java.util.Set set = hz.getSet("IBM-Quote-History");
set.add(new Price(10, time1));
set.add(new Price(11, time2));
set.add(new Price(12, time3));
```

```

set.add(new Price(11, time4));
//....
Iterator it = set.iterator();
while (it.hasNext()) {
    Price price = (Price) it.next();
    //analyze
}

```

3.4.2 Event Registration and Configuration

Hazelcast Set uses `ItemListener` to listen to events which occur when items are added and removed.

```

import java.util.Queue;
import java.util.Map;
import java.util.Set;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.ItemListener;
import com.hazelcast.core.EntryListener;
import com.hazelcast.core.EntryEvent;
import com.hazelcast.config.Config;

public class Sample implements ItemListener{

    public static void main(String[] args) {
        Sample sample = new Sample();
        Config cfg = new Config();
        HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
        ISet set = hz.getSet("default");
        set.addItemListener (sample, true);

        Price price = new Price(10, time1)
        set.add(price);
        set.remove(price);
    }

    public void itemAdded(Object item) {
        System.out.println("Item added = " + item);
    }

    public void itemRemoved(Object item) {
        System.out.println("Item removed = " + item);
    }
}

```

Related Information

Please refer to *Listener Configurations*.

3.5 List

Hazelcast List is very similar to Hazelcast Set but it allows duplicate elements.

- Besides allowing duplicate elements, Hazelcast List preserves the order of elements.
- Hazelcast List is non-partitioned data structure where values and each backup is represented by its own single partition.

- Hazelcast List cannot be scaled beyond the capacity of a single machine.
- There is no batching while iterating over List. All items will be copied to local and iteration will occur locally.

3.5.1 Sample List Code

```
import com.hazelcast.core.Hazelcast;
import java.util.List;
import java.util.Iterator;
import com.hazelcast.config.Config;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);

java.util.List list = hz.getList("IBM-Quote-Frequency");
list.add(new Price(10));
list.add(new Price(11));
list.add(new Price(12));
list.add(new Price(11));
list.add(new Price(12));

//....
Iterator it = list.iterator();
while (it.hasNext()) {
    Price price = (Price) it.next();
    //analyze
}
```

3.5.2 Event Registration and Configuration

Hazelcast List uses `ItemListener` to listen to events which occur when items are added and removed.

```
import java.util.Queue;
import java.util.Map;
import java.util.Set;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.ItemListener;
import com.hazelcast.core.EntryListener;
import com.hazelcast.core.EntryEvent;
import com.hazelcast.config.Config;

public class Sample implements ItemListener{

    public static void main(String[] args) {
        Sample sample = new Sample();
        Config cfg = new Config();
        HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
        IList list = hz.getList("default");
        list.addItemListener(sample, true);

        Price price = new Price(10, time1)
        list.add(price);
        list.remove(price);
    }

    public void itemAdded(Object item) {
        System.out.println("Item added = " + item);
    }
}
```

```

    }

    public void itemRemoved(Object item) {
        System.out.println("Item removed = " + item);
    }
}

```

Related Information

Please refer to [Listener Configurations](#).

3.6 Topic

Hazelcast provides distribution mechanism for publishing messages that are delivered to multiple subscribers which is also known as *publish/subscribe (pub/sub)* messaging model. Publishing and subscribing operations are cluster wide. When a member subscribes for a topic, it is actually registering for messages published by any member in the cluster, including the new members joined after you added the listener.

3.6.1 Statistics

Topic has two statistic variables that can be queried. These values are incremental and local to the member.

```

final HazelcastInstance instance = Hazelcast.newHazelcastInstance(config);
final ITopic<Object> myTopic = instance.getTopic("myTopicName");

myTopic.getLocalTopicStats().getPublishOperationCount();
myTopic.getLocalTopicStats().getReceiveOperationCount();

```

`getPublishOperationCount` and `getReceiveOperationCount` returns total number of publishes and received messages since the start of this node, respectively. Please note that, these values are not backed up and if the node goes down, they will be lost.

This feature can be disabled with topic configuration. Please see [Topic Configuration](#).

Related Information

These statistics values can be also viewed in Management Center. Please see [Topics](#).

3.6.2 Internals

Each node has the list of all registrations in the cluster. When a new node is registered for a topic, it will send a registration message to all members in the cluster. Also, when a new node joins the cluster, it will receive all registrations made so far in the cluster.

The behavior of topic varies depending on the value of configuration parameter `globalOrderEnabled`.

- If `globalOrderEnabled` is disabled:

Messages are ordered, i.e. listeners (subscribers) will process the messages in the order they are actually published. If cluster member *M* publishes messages *m1*, *m2*, *m3*, ..., *mn* to a topic **T**, then Hazelcast makes sure that all of the subscribers of topic **T** will receive and process *m1*, *m2*, *m3*, ..., *mn* in the given order.

Here is how it works. Let's say that we have three nodes (*node1*, *node2* and *node3*) and that *node1* and *node2* are registered to a topic named **news**. Notice that, all three nodes know that *node1* and *node2* registered to **news**.

In this example, *node1* publishes two messages: **a1** and **a2**. And, *node3* publishes two messages: **c1** and **c2**. When *node1* and *node3* publishes a message, they will check their local list for registered nodes. They discover that *node1* and *node2* are in the list. Then, it fires messages to those nodes. One of the possible order of messages received can be following.

Node1 -> c1, b1, a2, c2

Node2 -> c1, c2, a1, a2

- If `globalOrderEnabled` is enabled:

When enabled, it guarantees that all nodes listening the same topic will get messages in the same order.

Here is how it works. Let's say that again we have three nodes (*node1*, *node2* and *node3*) and that *node1* and *node2* are registered to a topic named **news**. Notice that all three nodes know that *node1* and *node2* registered to **news**.

In this example, *node1* publishes two messages: **a1** and **a2**. And, *node3* publishes two messages: **c1** and **c2**. When a node publishes messages over topic **news**, it first calculates which partition **news** ID corresponds to. Then, send an operation to owner of the partition for that node to publish messages. Let's assume that **news** corresponds to a partition that *node2* owns. Then, *node1* and *node3* first sends all messages to *node2*. Assume that the messages are published in the following order.

Node1 -> a1, c1, a2, c2

Then, *node2* publishes these messages by looking at registrations in its local list. It sends these messages to *node1* and *node2* (it will make a local dispatch for itself).

Node1 -> a1, c1, a2, c2

Node2 -> a1, c1, a2, c2

This way we guarantee that all nodes will see the events in same order.

In both cases, there is a **StripedExecutor** in **EventService** responsible for dispatching the received message. For all events in Hazelcast, the order that events are generated and the order they are published to the user are guaranteed to be the same via this **StripedExecutor**.

There are `hazelcast.event.thread.count` (default is 5) threads in **StripedExecutor**. For a specific event source (for topic, for a particular topic name), `hash of that source's name % 5` gives the ID of responsible thread. Note that, there can be another event source (**entryListener** of a map, **item listener** of a collection, etc.) corresponding to same thread. In order not to make other messages to block, heavy process should not be done in this thread. If there is a time consuming work needs to be done, the work should be handed over to another thread. Please see [Sample Topic Code](#).

3.6.3 Topic Configuration

- Declarative Configuration

```

'''xml
<hazelcast>
    ...

    <topic name="yourTopicName">
        <global-ordering-enabled>true</global-ordering-enabled>
        <statistics-enabled>true</statistics-enabled>
        <message-listeners>
            <message-listener>MessageListenerImpl</message-listener>
        </message-listeners>
    </topic>

    ...
</hazelcast>
'''

```

- Programmatic Configuration

```

'''java

final Config config = new Config();
final TopicConfig topicConfig = new TopicConfig();
topicConfig.setGlobalOrderingEnabled(true);
topicConfig.setStatisticsEnabled(true);
topicConfig.setName("yourTopicName");
final MessageListener<String> implementation = new MessageListener<String>() {
    @Override
    public void onMessage(Message<String> message) {
        // process the message
    }
};
topicConfig.addMessageListenerConfig(new ListenerConfig(implementation));
final HazelcastInstance instance = Hazelcast.newHazelcastInstance(config)'''

```

Default values are

- Global ordering is **false**, meaning there is no global order guarantee by default.
- Statistics are **true**, meaning statistics are calculated by default.

Topic related but not topic specific configuration parameters

- "hazelcast.event.queue.capacity" : default value is 1,000,000
- "hazelcast.event.queue.timeout.millis" : default value is 250
- "hazelcast.event.thread.count" : default value is 5

For these parameters see [Distributed Event Config](#)

3.6.4 Sample Topic Code

```

import com.hazelcast.core.Topic;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.MessageListener;
import com.hazelcast.config.Config;

public class Sample implements MessageListener<MyEvent> {

    public static void main(String[] args) {
        Sample sample = new Sample();
        Config cfg = new Config();
        HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
        ITopic topic = hz.getTopic("default");
        topic.addMessageListener(sample);
        topic.publish(new MyEvent());
    }

    public void onMessage(Message<MyEvent> message) {
        MyEvent myEvent = message.getMessageObject();
        System.out.println("Message received = " + myEvent.toString());
        if (myEvent.isHeavyweight()) {
            messageExecutor.execute(new Runnable() {
                public void run() {

```

```

        doHeavyweightStuff(myEvent);
    }
    });
}
}

// ...

private static final Executor messageExecutor = Executors.newSingleThreadExecutor();
}

```

3.7 Lock

ILock is the distributed implementation of `java.util.concurrent.locks.Lock`. Meaning if you lock on an ILock, the critical section that it guards is guaranteed to be executed by only one thread in entire cluster. Even though locks are great for synchronization, they can lead to problems if not used properly.

A few warnings when using locks:

- Always use lock with *try-catch* blocks. It will ensure that lock will be released if an exception is thrown from the code in critical section. Also note that lock method is outside *try-catch* block, because we do not want to unlock if lock operation itself fails.

```

import com.hazelcast.core.Hazelcast;
import com.hazelcast.config.Config;
import java.util.concurrent.locks.Lock;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
Lock lock = hz.getLock("myLock");
lock.lock();
try {
    // do something here
} finally {
    lock.unlock();
}

```

- If a lock is not released in the cluster, another thread that is trying to get the lock can wait forever. To avoid this, `tryLock` with a timeout value can be used. One can set a high value (normally should not take that long) for `tryLock`. Return value of `tryLock` can be checked as follows :

```

if (lock.tryLock (10, TimeUnit.SECONDS)) {
    try {
        // do some stuff here..
    }
    finally {
        lock.unlock();
    }
} else {
    // warning
}

```

- Another method to avoid ending up with indefinitely waiting threads is using lock with lease time. This will cause lock to be released in the given time. Lock can be unlocked before time expires safely. Note that the unlock operation can throw `IllegalMonitorStateException` if lock is released because of lease time expiration. If it is the case, it means that critical section guarantee is broken.

Please see the below example.

```

lock.lock (5, TimeUnit.SECONDS))
    try {
        // do some stuff here..
    } finally {
        try{
            lock.unlock();
        }catch(IllegalMonitorStateException ex ){
            // WARNING Critical section guarantee can be broken
        }

    }
} else{
    // warning
}

```

- Locks are fail-safe. If a member holds a lock and some other members go down, cluster will keep your locks safe and available. Moreover, when a member leaves the cluster, all the locks acquired by this dead member will be removed so that these locks can be available for live members immediately.
- Locks are re-entrant, meaning same thread can lock multiple times on the same lock. Note that for other threads to be able to require this lock, owner of the lock should call unlock as many times as it called lock.
- In split-brain scenario, cluster behaves as if there are two different clusters. Since two separate clusters are not aware of each other, two nodes from different clusters can acquire the same lock. For more information on places where split-brain can be handled, please see [Split Brain](#).

3.7.1 ICondition

ICondition is the distributed implementation of `notify`, `notifyAll` and `wait` operations on Java object . It can be used to synchronize threads across the cluster. More specifically, it is used when a thread's work depends on another thread's output. A good example can be producer/consumer methodology.

Please see the below code snippets for a sample producer/consumer implementation.

- Producer thread

```

HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
Lock lock = hz.getLock("myLockId");
ICondition condition = lock.newCondition("myConditionId");

lock.lock();

try {
    while (!shouldProduce()) {
        condition.await(); //freeds the lock and waits for signal
                           //when it wakes up it re-acquires the lock
                           //if available or waits for it to become
                           //available
    }
    produce()
    condition.signalAll();
} finally {
    lock.unlock();
}

```

- Consumer thread

```
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
Lock lock = hz.getLock("myLockId");
ICondition condition = lock.newCondition("myConditionId");

lock.lock();

try {
    while (!canConsume()) {
        condition.await(); //freeds the lock and waits for signal
                           //when it wakes up it re-acquires the lock if
                           //available or waits for it to become
                           //available
    }
    consume()
    condition.signalAll();
} finally {
    lock.unlock();
}
```


Chapter 4

Distributed Events

Hazelcast allows you to register for entry events to get notified when events occurred. Event Listeners are cluster-wide so when a listener is registered in one member of cluster, it is actually registering for events originated at any member in the cluster. When a new member joins, events originated at the new member will also be delivered.

An Event is created only if there is a listener registered. If there is no listener registered than no event will be created. If a predicate provided while registering the listener, predicate should pass before sending the event to the listener(node/client).

As a rule of thumb, event listener should not implement heavy processes in its event methods which block the thread for long time. If needed, **ExecutorService** can be used to transfer long running processes to another thread and offload current listener thread.

4.1 Event Listeners

- **MembershipListener** for cluster membership events
- **DistributedObjectListener** for distributed object creation and destroy events
- **MigrationListener** for partition migration start and complete events
- **LifecycleListener** for HazelcastInstance lifecycle events
- **EntryListener** for IMap and MultiMap entry events
- **ItemListener** for IQueue, ISet and IList item events (please refer to Event Registration and Configuration sections of [Set](#) and [List](#)).
- **MessageListener** for ITopic message events
- **ClientListener** for client connection events

4.2 Global Event Configuration

- `hazelcast.event.queue.capacity`: default value is 1000000
- `hazelcast.event.queue.timeout.millis`: default value is 250
- `hazelcast.event.thread.count`: default value is 5

There is a striped executor in each node to control and dispatch received events to user. This striped executor also guarantees the order. For all events in Hazelcast, the order that events are generated and the order they are published to the user are guaranteed for given keys. For map and multimap, order is preserved for the operations on same key of the entry. For list, set, topic and queue, order is preserved for events on that instance of the distributed data structure.

Order guarantee is achieved by making only one thread responsible for a particular set of events (entry events of a key in a map, item events of a collection, etc.) in **StripedExecutor**.

If event queue reaches the capacity (`hazelcast.event.queue.capacity`) and last item cannot be put to the event queue for timeout millis (`hazelcast.event.queue.timeout.millis`), these events will be dropped with a warning message like “EventQueue overloaded”.

If listeners are doing a computation that requires a long time, this can cause event queue to reach its maximum capacity and lost of events. For map and multimap, `hazelcast.event.thread.count` can be configured to a higher value so that less collision occurs for keys, therefore worker threads will not block each other in `StripedExecutor`. For list, set, topic and queue, heavy work should be offloaded to another thread. Notice that, in order to preserve order guarantee, the user should implement similar logic with `StripedExecutor` in offloaded thread pool.

Related Information

Please refer to *Listener Configurations* section on how to configure each listener.

Chapter 5

Distributed Computing

5.1 Executor Service

One of the coolest features of Java 1.5 is the Executor framework, which allows you to asynchronously execute your tasks, logical units of works, such as database query, complex calculation, image rendering, etc. So, one nice way of executing such tasks would be running them asynchronously and doing other things meanwhile. When ready, get the result and move on. If execution of the task takes longer than expected, you may consider canceling the task execution. In Java Executor framework, tasks are implemented as `java.util.concurrent.Callable` and `java.util Runnable`.

```
import java.util.concurrent.Callable;
import java.io.Serializable;

public class Echo implements Callable<String>, Serializable {
    String input = null;

    public Echo() {
    }

    public Echo(String input) {
        this.input = input;
    }

    public String call() {
        Config cfg = new Config();
        HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);
        return instance.getCluster().getLocalMember().toString() + ":" + input;
    }
}
```

Echo callable above, for instance, in its `call()` method, is returning the local member and the input passed in. Remember that `instance.getCluster().getLocalMember()` returns the local member and `toString()` returns the member's address (IP + port) in String form, just to see which member actually executed the code for our example. Of course, `call()` method can do and return anything you like. Executing a task by using executor framework is very straight forward. Simply obtain a `ExecutorService` instance, generally via `Executors` and submit the task which returns a `Future`. After executing task, you do not have to wait for execution to complete, you can process other things and when ready use the future object to retrieve the result as show in code below.

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<String> future = executorService.submit (new Echo("myinput"));
//while it is executing, do some useful stuff
//when ready, get the result of your execution
String result = future.get();
```

5.1.1 Execution

Distributed executor service is a distributed implementation of `java.util.concurrent.ExecutorService`. It allows you to execute your code in the cluster. In this chapter, all the code samples are based on the Echo class above. Please note that Echo class is `Serializable`. You can ask Hazelcast to execute your code (`Runnable`, `Callable`);

- on a specific cluster member you choose,
- on the member owning the key you choose,
- on the member Hazelcast will pick, and
- on all or subset of the cluster members.

```
“java import com.hazelcast.core.Member; import com.hazelcast.core.Hazelcast; import com.hazelcast.core.IExecutorService;
import java.util.concurrent.Callable; import java.util.concurrent.Future;
import java.util.Set; import com.hazelcast.config.Config;
```

```
public void echoOnTheMember(String input, Member member) throws Exception { Callable task = new
Echo(input); HazelcastInstance hz = Hazelcast.newHazelcastInstance(); IExecutorService executorService =
hz.getExecutorService(“default”); Future future = executorService.submitToMember(task, member); String
echoResult = future.get(); }
```

```
public void echoOnTheMemberOwningTheKey(String input, Object key) throws Exception { Callable task =
new Echo(input); HazelcastInstance hz = Hazelcast.newHazelcastInstance(); IExecutorService executorService =
hz.getExecutorService(“default”); Future future = executorService.submitToKeyOwner(task, key); String echoResult
= future.get(); }
```

```
public void echoOnSomewhere(String input) throws Exception { HazelcastInstance hz = Hazelcast.newHazelcastInstance();
IExecutorService executorService = hz.getExecutorService(“default”); Future future = executorService.submit(new
Echo(input)); String echoResult = future.get(); }
```

```
public void echoOnMembers(String input, Set members) throws Exception { HazelcastInstance hz = Hazel-
cast.newHazelcastInstance(); IExecutorService executorService = hz.getExecutorService(“default”); Map> futures
= executorService.submitToMembers(new Echo(input), members); for (Future future : futures.values()) { String
echoResult = future.get(); // ... } }“
```

Note that you can obtain the set of cluster members via `HazelcastInstance#getCluster().getMembers()` call.

5.1.2 Execution Cancellation

What if the code you execute in cluster takes longer than expected. If you cannot stop/cancel that task, it will keep eating your resources. Standard Java executor framework solves this problem with by introducing `cancel()` API and “encouraging” us to code and design for cancellations, which is highly ignored part of software development.

```
public class Fibonacci<Long> implements Callable<Long>, Serializable {
    int input = 0;

    public Fibonacci() {
    }

    public Fibonacci(int input) {
        this.input = input;
    }

    public Long call() {
        return calculate (input);
    }
}
```

```

private long calculate (int n) {
    if (Thread.currentThread().isInterrupted()) return 0;
    if (n <= 1) return n;
    else return calculate(n-1) + calculate(n-2);
}
}

```

The callable class above calculates the Fibonacci number for a given number. In the `calculate` method, we are checking to see if the current thread is interrupted so that code can be responsive to cancellations once the execution is started. Below `fib()` method submits the Fibonacci calculation task for number 'n' and waits maximum 3 seconds for result. If the execution does not completed in 3 seconds, `future.get()` will throw `TimeoutException` and upon catching, it we interruptibly cancel the execution for saving some CPU cycles.

```

long fib(int n) throws Exception {
    Config cfg = new Config();
    HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
    IExecutorService es = hz.getExecutorService();
    Future future = es.submit(new Fibonacci(n));
    try {
        return future.get(3, TimeUnit.SECONDS);
    } catch (TimeoutException e) {
        future.cancel(true);
    }
    return -1;
}

```

`fib(20)` will probably take less than 3 seconds but, `fib(50)` will take way longer. (This is not the example for writing better Fibonacci calculation code, but for showing how to cancel a running execution that takes too long). The method `future.cancel(false)` can only cancel execution before it is running (executing), but `future.cancel(true)` can interrupt running executions if your code is able to handle the interruption. So, if you are willing to be able to cancel already running task, then your task has to be designed to handle interruption. If `calculate (int n)` method did not have `(Thread.currentThread().isInterrupted())` line, then you would not be able to cancel the execution after it is started.

5.1.3 Execution Callback

`ExecutionCallback` allows you to asynchronously get notified when the execution is done. Below is a sample code.

```

public class Fibonacci<Long> implements Callable<Long>, Serializable {
    int input = 0;

    public Fibonacci() {
    }

    public Fibonacci(int input) {
        this.input = input;
    }

    public Long call() {
        return calculate (input);
    }

    private long calculate (int n) {
        if (n <= 1) return n;
        else return calculate(n-1) + calculate(n-2);
    }
}

```

```

import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.ExecutionCallback;
import com.hazelcast.core.IExecutorService;
import java.util.concurrent.Future;
import com.hazelcast.config.Config;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
IExecutorService es = hz.getExecutorService();
Callable<Long> task = new Fibonacci(10);

es.submit(task, new ExecutionCallback<Long> () {

    public void onResponse(Long response) {
        System.out.println("Fibonacci calculation result = " + response);
    }

    public void onFailure(Throwable t) {
        t.printStackTrace();
    }

});

```

5.2 Entry Processor

Hazelcast supports entry processing. The interface `EntryProcessor` gives you the ability to execute your code on an entry in an atomic way. You do not need any explicit lock on entry. Practically, Hazelcast locks the entry, runs the `EntryProcessor`, and then unlocks the entry. If entry processing is the major operation for a map and the map consists of complex objects, then using object type as `in-memory-format` is recommended to minimize serialization cost.

There are below methods in `IMap` interface for entry processing:

```

“java /** * Applies the user defined EntryProcessor to the entry mapped by the key. * Returns the the object
which is result of the process() method of EntryProcessor. */

```

```

Object executeOnKey(K key, EntryProcessor entryProcessor);

```

```

/** * Applies the user defined EntryProcessor to the entries mapped by the collection of keys. * the results mapped
by each key in the collection. */

```

```

Map<K, Object> executeOnKeys(Set<K> keys, EntryProcessor entryProcessor);

```

```

/** * Applies the user defined EntryProcessor to the entry mapped by the key with * specified ExecutionCallback
to listen event status and returns immediately. */

```

```

void submitToKey(K key, EntryProcessor entryProcessor, ExecutionCallback callback);

```

```

/** * Applies the user defined EntryProcessor to the all entries in the map. * Returns the results mapped by each
key in the map. */

```

```

Map<K, Object> executeOnEntries(EntryProcessor entryProcessor);

```

```

/** * Applies the user defined EntryProcessor to the entries in the map which satisfies provided predicate. *
Returns the results mapped by each key in the map. */

```

```
Map<K, Object> executeOnEntries(EntryProcessor entryProcessor, Predicate predicate);
'''
```

Using `executeOnEntries` method, if the number of entries is high and you do need the results, then returning null in `process(..)` method is a good practice.

Here is the `EntryProcessor` interface:

```
public interface EntryProcessor<K, V> extends Serializable {

    Object process(Map.Entry<K, V> entry);

    EntryBackupProcessor<K, V> getBackupProcessor();
}
```

If your code is modifying the data, then you should also provide a processor for backup entries:

```
public interface EntryBackupProcessor<K, V> extends Serializable {

    void processBackup(Map.Entry<K, V> entry);
}
```

Example Usage:

```
public class EntryProcessorTest {

    @Test
    public void testMapEntryProcessor() throws InterruptedException {
        Config cfg = new Config();
        cfg.getMapConfig("default").setInMemoryFormat(MapConfig.InMemoryFormat.OBJECT);
        HazelcastInstance instance1 = Hazelcast.newHazelcastInstance(cfg);
        HazelcastInstance instance2 = Hazelcast.newHazelcastInstance(cfg);
        IMap<Integer, Integer> map = instance1.getMap("testMapEntryProcessor");
        map.put(1, 1);
        EntryProcessor entryProcessor = new IncrementorEntryProcessor();
        map.executeOnKey(1, entryProcessor);
        assertEquals(map.get(1), (Object) 2);
        instance1.getLifecycleService().shutdown();
        instance2.getLifecycleService().shutdown();
    }

    @Test
    public void testMapEntryProcessorAllKeys() throws InterruptedException {
        StaticNodeFactory nodeFactory = new StaticNodeFactory(2);
        Config cfg = new Config();
        cfg.getMapConfig("default").setInMemoryFormat(MapConfig.InMemoryFormat.OBJECT);
        HazelcastInstance instance1 = nodeFactory.newHazelcastInstance(cfg);
        HazelcastInstance instance2 = nodeFactory.newHazelcastInstance(cfg);
        IMap<Integer, Integer> map = instance1.getMap("testMapEntryProcessorAllKeys");
        int size = 100;
        for (int i = 0; i < size; i++) {
            map.put(i, i);
        }
        EntryProcessor entryProcessor = new IncrementorEntryProcessor();
        Map<Integer, Object> res = map.executeOnEntries(entryProcessor);
        for (int i = 0; i < size; i++) {
            assertEquals(map.get(i), (Object) (i+1));
        }
    }
}
```

```
    }
    for (int i = 0; i < size; i++) {
        assertEquals(map.get(i)+1, res.get(i));
    }
    instance1.getLifecycleService().shutdown();
    instance2.getLifecycleService().shutdown();
}

static class IncrementorEntryProcessor implements EntryProcessor, EntryBackupProcessor, Serializable {
    public Object process(Map.Entry entry) {
        Integer value = (Integer) entry.getValue();
        entry.setValue(value + 1);
        return value + 1;
    }

    public EntryBackupProcessor getBackupProcessor() {
        return IncrementorEntryProcessor.this;
    }

    public void processBackup(Map.Entry entry) {
        entry.setValue((Integer) entry.getValue() + 1);
    }
}
}
```


Chapter 6

Distributed Query

6.1 Query

Hazelcast partitions your data and spreads across cluster of servers. You can surely iterate over the map entries and look for certain entries you are interested in but this is not very efficient as you will have to bring entire entry set and iterate locally. Instead, Hazelcast allows you to run distributed queries on your distributed map.

Assume that you have an “employee” map containing values of `Employee` objects:

```
import java.io.Serializable;

public class Employee implements Serializable {
    private String name;
    private int age;
    private boolean active;
    private double salary;

    public Employee(String name, int age, boolean live, double price) {
        this.name = name;
        this.age = age;
        this.active = live;
        this.salary = price;
    }

    public Employee() {
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public double getSalary() {
        return salary;
    }

    public boolean isActive() {
        return active;
    }
}
```

Now you are looking for the employees who are active and with age less than 30. Hazelcast allows you to find these entries in two different ways:

6.1.1 Distributed SQL Query

SqlPredicate takes regular SQL where clause. Here is an example:

```
import com.hazelcast.core.IMap;
import com.hazelcast.query.SqlPredicate;
```

```
Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
IMap map = hz.getMap("employee");
```

```
Set<Employee> employees = (Set<Employee>) map.values(new SqlPredicate("active AND age < 30"));
```

Supported SQL syntax:

- AND/OR
 - '`<expression> AND <expression> AND <expression>...`'
 - '`active AND age>30`'
 - '`active=false OR age = 45 OR name = 'Joe'`'
 - '`active AND (age >20 OR salary < 60000)`'
- =, !=, <, <=, >, >=
 - '`<expression> = value`'
 - '`age <= 30`'
 - '`name ="Joe"`'
 - '`salary != 50000`'
- BETWEEN
 - '`<attribute> [NOT] BETWEEN <value1> AND <value2>`'
 - '`age BETWEEN 20 AND 33 (same as age >=20 AND age<=33)`'
 - '`age NOT BETWEEN 30 AND 40 (same as age <30 OR age>40)`'
- LIKE
 - '`<attribute> [NOT] LIKE 'expression'`'
 - '%' (percentage sign) is placeholder for many characters, '_' (underscore) is placeholder for one character
 - '`name LIKE 'Jo%'`' (true for 'Joe', 'Josh', 'Joseph' etc.)
 - '`name LIKE 'Jo_'`' (true for 'Joe'; false for 'Josh')
 - '`name NOT LIKE 'Jo_'`' (true for 'Josh'; false for 'Joe')
 - '`name LIKE 'J_s%'`' (true for 'Josh', 'Joseph'; false 'John', 'Joe')

- IN
 - `<attribute> [NOT] IN (val1, val2,...)'`
 - `'age IN (20, 30, 40)'`
 - `'age NOT IN (60, 70)'`

Examples:

- `active AND (salary >= 50000 OR (age NOT BETWEEN 20 AND 30))`
- `age IN (20, 30, 40) AND salary BETWEEN (50000, 80000)`

6.1.2 Criteria API

If SQL is not enough or programmable queries are preferred, then JPA criteria like API can be used. Here is an example:

```
import com.hazelcast.core.IMap;
import com.hazelcast.query.Predicate;
import com.hazelcast.query.PredicateBuilder;
import com.hazelcast.query.EntryObject;
import com.hazelcast.config.Config;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
IMap map = hz.getMap("employee");

EntryObject e = new PredicateBuilder().getEntryObject();
Predicate predicate = e.is("active").and(e.get("age").lessThan(30));

Set<Employee> employees = (Set<Employee>) map.values(predicate);
```

6.1.3 Paging Predicate (Order & Limit)

Hazelcast provides paging for defined predicates. For this purpose, `PagingPredicate` class has been developed. You may want to get collection of keys, values or entries page by page, by filtering them with predicates and giving the size of pages. Also, you can sort the entries by specifying comparators.

Below is a sample code where the `greaterEqual` predicate is used to get values from “students” map. This predicate puts a filter such that the objects with value of “age” is greater than or equal to 18 will be retrieved. Then, a `PagingPredicate` is constructed in which the page size is 5. So, there will be 5 objects in each page.

The first time the values are called will constitute the first page. You can get the subsequent pages by using the `nextPage()` method of `PagingPredicate`.

```
final IMap<Integer, Student> map = instance.getMap("students");
final Predicate greaterEqual = Predicates.greaterEqual("age", 18);
final PagingPredicate pagingPredicate = new PagingPredicate(greaterEqual, 5);
Collection<Student> values = map.values(pagingPredicate); //First Page
...

pagingPredicate.nextPage();
values = map.values(pagingPredicate); //Second Page
...
```

Paging Predicate is not supported in Transactional Context.

Note: Please refer to [here](#) for all predicates.

6.1.4 Indexing

Hazelcast distributed queries will run on each member in parallel and only results will return the conn. When a query runs on a member, Hazelcast will iterate through the entire owned entries and find the matching ones. This can be made faster by indexing the mostly queried fields. Just like you would do for your database. Of course, indexing will add overhead for each `write` operation but queries will be a lot faster. If you are querying your map a lot, make sure to add indexes for most frequently queried fields. So, if your `active` and `age < 30` query, for example, is used a lot, make sure you add index for `active` and `age` fields. Here is how:

```
Config cfg = new Config();
HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);
IMap imap = instance.getMap("employees");
imap.addIndex("age", true);           // ordered, since we have ranged queries for this field
imap.addIndex("active", false);      // not ordered, because boolean field cannot have range
```

`IMap.addIndex(fieldName, ordered)` is used for adding index. For each indexed field, if you have ranged queries such as `age>30`, `age BETWEEN 40 AND 60`, then `ordered` parameter should be `true`. Otherwise, set it to `false`.

Also, you can define `IMap` indexes in configuration, a sample of which is shown below.

```
‘‘‘xml
<map name="default">
  ...
  <indexes>
    <index ordered="false">name</index>
    <index ordered="true">age</index>
  </indexes>
</map>‘‘‘
```

This sample in programmatic configuration looks like below.

```
‘‘‘java
mapConfig.addMapIndexConfig(new MapIndexConfig("name", false));
mapConfig.addMapIndexConfig(new MapIndexConfig("age", true));‘‘‘
```

And, the following is the Spring declarative configuration for the same sample.

```
‘‘‘xml
<hz:map name="default">
  <hz:indexes>
    <hz:index attribute="name"/>
    <hz:index attribute="age" ordered="true"/>
  </hz:indexes>
</hz:map>
```

““

6.2 MapReduce

You have heard about MapReduce ever since Google released its [research white paper](#) on this concept. With Hadoop as the most common and well known implementation, MapReduce gained a broad audience and made it into all kinds of business applications dominated by data warehouses.

From what we see at the white paper, MapReduce is a software framework for processing large amounts of data in a distributed way. Therefore, the processing is normally spread over several machines. The basic idea behind MapReduce is to map your source data into a collection of key-value pairs and reducing those pairs, grouped by key, in a second step towards the final result.

The main idea can be summarized with below 3 simple steps.

1. Read source data
2. Map data to one or multiple key-value pairs
3. Reduce all pairs with the same key

Use Cases

The best known examples for MapReduce algorithms are text processing tools like counting the word frequency in large texts or websites. Apart from that, there are more interesting example use cases as listed below.

- Log Analysis
- Data Querying
- Aggregation and summing
- Distributed Sort
- ETL (Extract Transform Load)
- Credit and Risk management
- Fraud detection
- and more...

6.2.1 MapReduce Essentials

This section will give a deeper insight on the MapReduce pattern and help to understand the semantics behind the different MapReduce phases and how they are implemented in Hazelcast.

In addition to this, there are hints in the sections which compare Hadoop and Hazelcast MapReduce implementation to help adopters with Hadoop background to quickly get familiar with their new target.

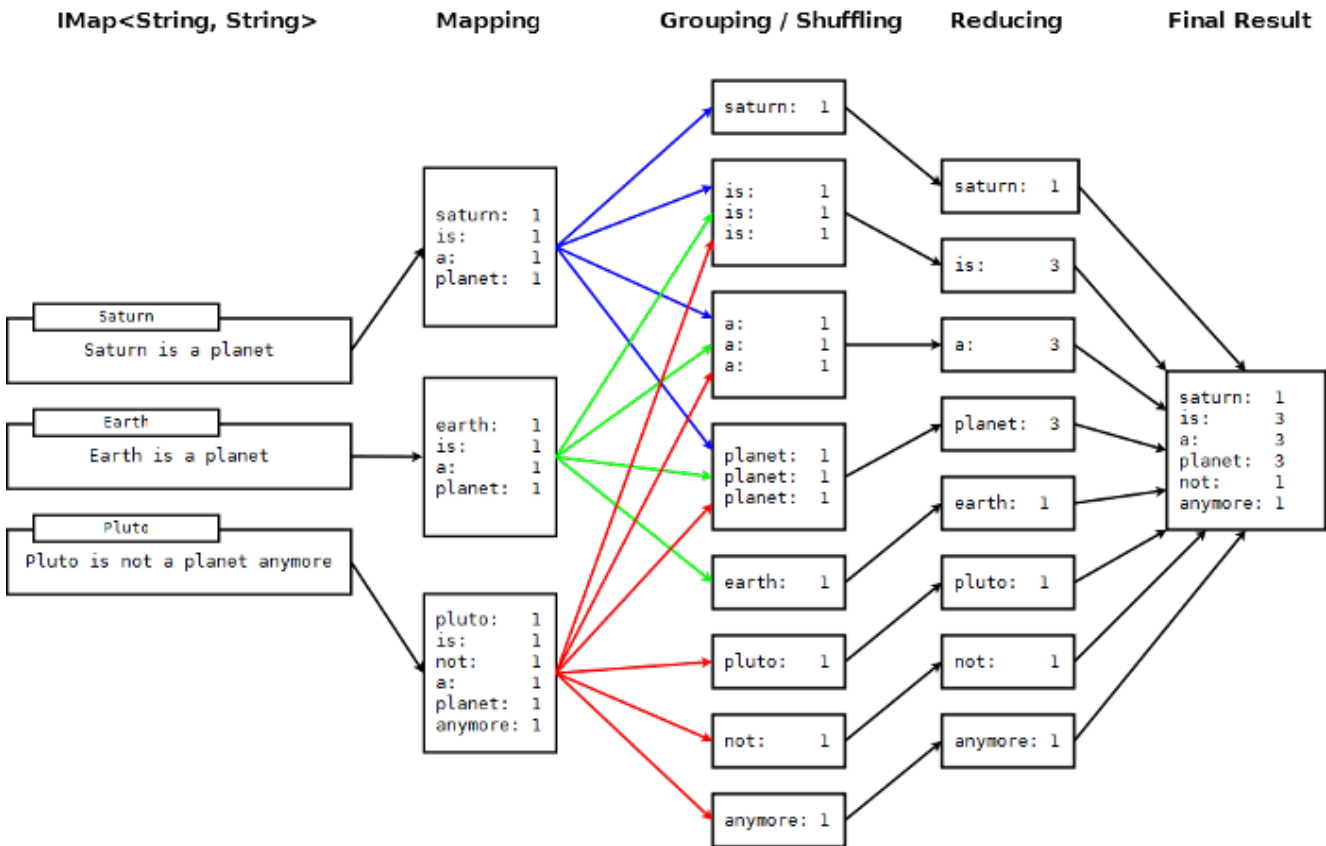
6.2.1.1 MapReduce Workflow Example

Below flowchart demonstrates a basic workflow of the already mentioned word count example (distributed occurrences analysis). From left to right, it iterates over all entries of a data structure (in this case an IMap). In the mapping phase, it splits the sentence in single words and emits a key-value pair per word with the word as a key and 1 as the value. In the next phase, values are collected (grouped) and transported to their corresponding reducers where they are eventually reduced to a single key-value pair with the value as the number of occurrences of the word. As the last step, the different reducer results are grouped up to the final result and returned to the requester.

In pseudo code, the corresponding map and reduce function would look like the following. Hazelcast code example will be shown in the next section.

```
map( key:String, document:String ):Void ->
    for each w:word in document:
        emit( w, 1 )

reduce( word:String, counts:List[Int] ):Int ->
    return sum( counts )
```



6.2.1.2 MapReduce Phases

As seen in the workflow example, a MapReduce process consists of multiple phases. The original MapReduce pattern describes two phases (map, reduce) and one optional phase (combine). In Hazelcast, these phases are either only existing virtually to explain the data flow or are executed in parallel during the real operation while the general idea is still persisting.

$(K \times V)^* \rightarrow (L \times W)^*$

$[(k1, v1), \dots, (kn, vn)] \rightarrow [(l1, w1), \dots, (lm, wm)]$

Mapping Phase

The mapping phase iterates all key-value pairs of any kind of legal input source. The mapper then analyzes the input pairs and emits zero or more new key-value pairs.

$K \times V \rightarrow (L \times W)^*$

$(k, v) \rightarrow [(l1, w1), \dots, (ln, wn)]$

Combine Phase

In the combine phase, multiple key-value pairs with the same key are collected and combined to an intermediate result before being sent to the reducers. **Combine phase is also optional in Hazelcast, but is highly recommended to use to lower the traffic.**

In terms of the word count example, this can be explained using the sentences “Saturn is a planet but the Earth is a planet, too”. As shown above, we would send two key-value pairs (planet, 1). The registered combiner now collects those two pairs and combines them to an intermediate result of (planet, 2). Instead of two key-value pairs sent through the wire, there is now only one for the key “planet”.

The pseudo code for a combiner is pretty the same as for the reducer.

```
combine( word:String, counts:List[Int] ):Void ->
    emit( word, sum( counts ) )
```

Grouping / Shuffling Phase

The grouping or shuffling phase only exists virtually in Hazelcast since it is not a real phase; emitted key-value pairs with the same key are always transferred to the same reducer in the same job. That way they are grouped together which is equivalent to the shuffling phase.

Reducing Phase

In the reducing phase, the collected intermediate key-value pairs are reduced by their keys to build the final by-key result. This value can be a sum of all the emitted values of the same key, an average value or something completely different depending on the use case.

A reduced representation of this phase:

$$L \times W^* \rightarrow X^*$$

$$(l, [w_1, \dots, w_n]) \rightarrow [x_1, \dots, x_n]$$

Producing the Final Result

This also is not a real MapReduce phase but is the final step in Hazelcast after all reducers notified that reducing has finished. The original job initiator then requests all reduced results and builds the final result.

6.2.1.3 Additional MapReduce Resources

The Internet is full of useful resources to find deeper information on MapReduce. Below is a short collection of some more introduction material. In addition, there are a lot of amazing books written about all kinds of MapReduce patterns and how to write a MapReduce function for your use case. To name them all is sadly out of scope of this documentation.

- <http://labs.google.com/papers/mapreduce.html>
- <http://en.wikipedia.org/wiki/MapReduce>
- http://hci.stanford.edu/courses/cs448g/a2/files/map_reduce_tutorial.pdf
- <http://ksat.me/map-reduce-a-really-simple-introduction-kloudo/>
- <http://www.slideshare.net/franbandov/an-introduction-to-mapreduce-6789635>

6.2.2 Introduction to MapReduce API

This section explains basics of the Hazelcast MapReduce framework. While walking through the different API classes, we will build the word count example that was discussed earlier and create it step by step.

The Hazelcast API for MapReduce operations consists of a fluent DSL like configuration syntax to build and submit jobs. JobTracker is the basic entry point to all MapReduce operations and is retrieved from `com.hazelcast.core.HazelcastInstance` by calling `getJobTracker` and supplying the name of the required JobTracker configuration. The configuration for JobTrackers will be discussed later, for now we focus on the API itself. In addition, the complete submission part of the API is built to support a fully reactive way of programming.

To give an easy introduction to people that are already used to Hadoop, we decided to create the class names as familiar as possible to their counterparts on Hadoop. That means while most users will recognize a lot of similar sounding classes, the way to configure the jobs is more fluent due to the already mentioned DSL like styled API.

While building the example, we will go through as much options as possible, e.g. we create a specialized JobTracker configuration (at the end). Special JobTracker configuration are not required, as for all other Hazelcast features you can use “default” as the configuration name, but special configurations offer better options to predict behavior of the framework while execution.

The full example is available [here](#) as a ready to run Maven project.

6.2.2.1 JobTracker

The JobTracker is used to create Job instances whereas every instance of `com.hazelcast.mapreduce.Job` defines a single MapReduce configuration. The same Job can be submitted multiple times, no matter if executed in parallel or after the previous execution is finished.

Note: *After retrieving the JobTracker, be aware of the fact that it should only be used with data structures derived from the same HazelcastInstance. Otherwise, unexpected behavior will happen.*

To retrieve a JobTracker from Hazelcast, we will start by using the “default” configuration for convenience reasons to show the basic way.

```
import com.hazelcast.core.*;
import com.hazelcast.mapreduce.*;
```

```
HazelcastInstance hazelcastInstance = getHazelcastInstance();
JobTracker jobTracker = hazelcastInstance.getJobTracker( "default" );
```

JobTracker is retrieved using the same kind of entry point as most of other Hazelcast features. After building the cluster connection, you use the created HazelcastInstance to request the configured (or default) JobTracker from Hazelcast.

Next step will be to create a new Job and configure it to execute our first MapReduce request against cluster data.

6.2.2.2 Job

As mentioned in the last section, a Job is created using the retrieved JobTracker instance. A Job defines exactly one configuration of a MapReduce task. Mapper, combiner and reducers will be defined per job but since the Job instance is only a configuration, it is possible to be submitted multiple times, no matter if executions happening in parallel or one after the other.

A submitted job is always identified using a unique combination of the JobTracker’s name and a, on submit-time generated, jobId. The way for retrieving the jobId will be shown in one of the later sections.

To create a Job, a second class `com.hazelcast.mapreduce.KeyValueSource` is necessary. We will have a deeper look at the KeyValueSource class in the next section, for now it is enough to know that it is used to wrap any kind of data or data structure into a well defined set of key-value pairs.

Below example code is a direct follow up of the example of the JobTracker section and reuses the already created HazelcastInstance and JobTracker instances.

We start by retrieving an instance of our data map and create the Job instance afterwards. Implementations used to configure the Job will be discussed while walking further through the API documentation, they are not yet discussed.

Note: *Since the Job class is highly depending on generics to support type safety, the generics change over time and may not be assignment compatible to old variable types. To create full potential of the fluent API, we recommend to use fluent method chaining as shown in this example to prevent the need of too much variables.*

```
IMap<String, String> map = hazelcastInstance.getMap( "articles" );
KeyValueSource<String, String> source = KeyValueSource.fromMap( map );
Job<String, String> job = jobTracker.newJob( source );
```

```
ICompletableFuture<Map<String, Long>> future = job
    .mapper( new TokenizerMapper() )
    .combiner( new WordCountCombinerFactory() )
    .reducer( new WordCountReducerFactory() )
    .submit();
```



```
// Attach a callback listener
future.andThen( buildCallback() );

// Wait and retrieve the result
Map<String, Long> result = future.get();
```

As seen above, we create the Job instance and define a mapper, combiner, reducer and eventually submit the request to the cluster. The `submit` method returns an `ICompletableFuture` that can be used to attach our callbacks or just to wait for the result to be processed in a blocking fashion.

There are more options available for job configuration like defining a general chunk size or on what keys the operation will be operate. For more information, please consolidate the Javadoc matching your used Hazelcast version.

6.2.2.3 KeyValueSource

The `KeyValueSource` is able to either wrap Hazelcast data structures (like `IMap`, `MultiMap`, `IList`, `ISet`) into key-value pair input sources or to build your own custom key-value input source. The latter option makes it possible to feed Hazelcast MapReduce with all kind of data like just-in-time downloaded web page contents or data files. People familiar with Hadoop will recognize similarities with the `Input` class.

You can imagine a `KeyValueSource` as a bigger `java.util.Iterator` implementation. Whereas most methods are required to be implemented, `getAllKeys` is optional to implement. If implementation is able to gather all keys upfront, it should be implemented and `isAllKeysSupported` must return true, that way Job configured `KeyPredicates` are able to be evaluate keys upfront before sending them to the cluster. Otherwise, they are serialized and transfered as well to be evaluated at execution time.

As shown in the example above, the abstract `KeyValueSource` class provides a number of static methods to easily wrap Hazelcast data structures into `KeyValueSource` implementations already provided by Hazelcast. The data structures' generics are inherited into the resulting `KeyValueSource` instance. For data structures like `IList` or `ISet`, the key type is always `String`. While mapping, the key is the data structure's name whereas the value type and value itself are inherited from the `IList` or `ISet` itself.

```
// KeyValueSource from com.hazelcast.core.IMap
IMap<String, String> map = hazelcastInstance.getMap( "my-map" );
KeyValueSource<String, String> source = KeyValueSource.fromMap( map );

// KeyValueSource from com.hazelcast.core.MultiMap
MultiMap<String, String> multiMap = hazelcastInstance.getMultiMap( "my-multimap" );
KeyValueSource<String, String> source = KeyValueSource.fromMultiMap( multiMap );

// KeyValueSource from com.hazelcast.core.IList
IList<String> list = hazelcastInstance.getList( "my-list" );
KeyValueSource<String, String> source = KeyValueSource.fromList( list );

// KeyValueSource from com.hazelcast.core.ISet
ISet<String> set = hazelcastInstance.getSet( "my-set" );
KeyValueSource<String, String> source = KeyValueSource.fromSet( set );
```

PartitionIdAware

The `com.hazelcast.mapreduce.PartitionIdAware` interface can be implemented by the `KeyValueSource` implementation if the underlying data set is aware of the Hazelcast partitioning schema (as it is for all internal data structures). If this interface is implemented, the same `KeyValueSource` instance is reused multiple times for all partitions on the cluster node. As a consequence, the `close` and `open` methods are also executed multiple times but once per `partitionId`.

6.2.2.4 Mapper

Using the Mapper interface, you will implement the mapping logic. Mappers can transform, split, calculate, aggregate data from data sources. In Hazelcast, it is also possible to integrate data from more than the KeyValueSource data source by implementing `com.hazelcast.core.HazelcastInstanceAware` and requesting additional maps, multimaps, list, sets.

The mappers `map` function is called once per available entry in the data structure. If you work on distributed data structures that operate in a partition based fashion, then multiple mappers work in parallel on the different cluster nodes, on the nodes' assigned partitions. Mappers then prepare and maybe transform the input key-value pair and emit zero or more key-value pairs for reducing phase.

For our word count example, we retrieve an input document (a text document) and we transform it by splitting the text into the available words. After that, as discussed in the pseudo code, we emit every single word with a key-value pair of the word itself as key and 1 as the value.

A common implementation of that Mapper might look like the following example:

```
public class TokenizerMapper implements Mapper<String, String, String, Long> {
    private static final Long ONE = Long.valueOf( 1L );

    @Override
    public void map(String key, String document, Context<String, Long> context) {
        StringTokenizer tokenizer = new StringTokenizer( document.toLowerCase() );
        while ( tokenizer.hasMoreTokens() ) {
            context.emit( tokenizer.nextToken(), ONE );
        }
    }
}
```

The code is pretty basic and just splits the mapped texts into their tokens and iterate over the tokenizer as long as there are more tokens and emits a pair per word. What is to note, we're not yet collecting multiple occurrences of the same word but just fire every word on its own.

LifecycleMapper / LifecycleMapperAdapter

The LifecycleMapper interface or its adapter class LifecycleMapperAdapter can be used to make the Mapper implementation lifecycle aware. That means it will be notified when mapping of a partition or set of data begins and when the last entry was mapped.

Only special algorithms might have a need for those additional lifecycle events to perform preparation, cleanup or emit additional values.

6.2.2.5 Combiner / CombinerFactory

As stated in the introduction, a Combiner is used to minimize traffic between the different cluster nodes when transmitting mapped values from mappers to the reducers by aggregating multiple values for the same emitted key. This is a fully optional operation but is highly recommended to be used.

Combiners can be seen as an intermediate reducer. The calculated value is always assigned back to the key for which the combiner initially was created. Since combiners are created per emitted key, not the Combiner implementation itself is defined in the jobs configuration but a CombinerFactory that is able to create the expected Combiner instance.

Due to the fact that Hazelcast MapReduce is executing mapping and reducing phase in parallel, the Combiner implementation must be able to deal with chunked data. That means, it is required to reset its internal state whenever `finalizeChunk` is called. Calling that method creates a chunk of intermediate data to be grouped (shuffled) and sent to the reducers.

Combiners can override `beginCombine` and `finalizeCombine` to perform preparation or cleanup work.

For our word count example, we are going to have a simple CombinerFactory and Combiner implementation similar to the following one:

```

public class WordCountCombinerFactory implements CombinerFactory<String, Long, Long> {

    @Override
    public Combiner<String, Long, Long> newCombiner( String key ) {
        return new WordCountCombiner();
    }

    private class WordCountCombiner extends Combiner<String, Long, Long> {
        private long sum = 0;

        @Override
        public void combine( String key, Long value ) {
            sum++;
        }

        @Override
        public Long finalizeChunk() {
            long chunk = sum;
            sum = 0;
            return chunk;
        }
    }
}

```

As mentioned before, the Combiner must be able to return its current value as a chunk and reset the internal state by setting sum back to 0. Since combiners are always called from a single thread, no synchronization or volatility of the variables is necessary.

6.2.2.6 Reducer / ReducerFactory

Reducers doing the last bit of algorithm work. This can be aggregating values, calculating averages or anything else that is expected by the algorithm to work.

Since values arrive in chunks, the reduce method is called multiple times for every emitted value of the creation key. This also can happen multiple times per chunk if no Combiner implementation was configured for a job configuration.

In difference of the combiners, a reducers `finalizeReduce` method is only called once per reducer (which means once per key). So, a reducer does not need to be able to reset its internal state at any time.

Reducers can override `beginReduce` to perform preparation work.

Again for our word count example, the implementation will look similar to the following code snippet:

```

public class WordCountReducerFactory implements ReducerFactory<String, Long, Long> {

    @Override
    public Reducer<String, Long, Long> newReducer( String key ) {
        return new WordCountReducer();
    }

    private class WordCountReducer extends Reducer<String, Long, Long> {

        private volatile long sum = 0;

        @Override
        public void reduce( Long value ) {
            sum += value.longValue();
        }
    }
}

```

```

        @Override
        public Long finalizeReduce() {
            return sum;
        }
    }
}

```

Different from combiners, reducer tends to switch threads if running out of data to prevent blocking threads from the JobTracker configuration. They are rescheduled at a later point when new data to be processed arrives but unlikely to be executed on the same thread as before. Due to this fact, some volatility of the internal state might be necessary.

6.2.2.7 Collator

A Collator is an optional operation that is executed on the job emitting node and is able to modify the finally reduced result before returned to the user's codebase. Only special use cases are likely to make use of collators.

For an imaginary use case, we might want to know how many words were all over in the documents we analyzed and for this case, a Collator implementation can be given to the `submit` method of the Job instance.

A collator would look like the following snippet:

```

public class WordCountCollator implements Collator<Map.Entry<String, Long>, Long> {

    @Override
    public Long collate( Iterable<Map.Entry<String, Long>> values ) {
        long sum = 0;

        for ( Map.Entry<String, Long> entry : values ) {
            sum += entry.getValue().longValue();
        }
        return sum;
    }
}

```

The definition of the input type is a bit strange but due to the fact that Combiner and Reducer implementations are optional, the input type heavily depends on the state of the data. As stated above, collators are non-typical use cases and the generics of the framework always help in finding the correct signature.

6.2.2.8 KeyPredicate

A KeyPredicate can be used to pre-select if a key should be selected for mapping in the mapping phase. If the KeyValueSource implementation is able to know all keys upfront to execution, the keys are filtered before the operations are divided to the different cluster nodes.

It is also possible to be used to select only a special range of data (e.g. a time-frame) or similar use cases.

A basic KeyPredicate implementation to only map keys containing the word "hazelcast" might look like the following code class:

```

public class WordCountKeyPredicate implements KeyPredicate<String> {

    @Override
    public boolean evaluate( String s ) {
        return s != null && s.toLowerCase().contains( "hazelcast" );
    }
}

```

6.2.2.9 TrackableJob and Job Monitoring

A `TrackableJob` instance can be retrieved after submitting a job. It is requested from the `JobTracker` using the, per `JobTracker`, unique `jobId`. It can be used to get runtime statistics of the job. At the moment, the information available are limited to the number of processed (mapped) records and the processing state of the different partitions or nodes (if `KeyValueSource` is not `PartitionIdAware`).

To retrieve the `jobId` after submission of the job, use `com.hazelcast.mapreduce.JobCompletableFuture` instead of the `com.hazelcast.core.ICompletableFuture` as variable type for the returned future.

Below snippet will give a quick introduction on how to retrieve the instance and the runtime data. For more information, please have a look at the Javadoc corresponding your running Hazelcast version.

```
IMap<String, String> map = hazelcastInstance.getMap( "articles" );
KeyValueSource<String, String> source = KeyValueSource.fromMap( map );
Job<String, String> job = jobTracker.newJob( source );

JobCompletableFuture<Map<String, Long>> future = job
    .mapper( new TokenizerMapper() )
    .combiner( new WordCountCombinerFactory() )
    .reducer( new WordCountReducerFactory() )
    .submit();

String jobId = future.getJobId();
TrackableJob trackableJob = jobTracker.getTrackableJob(jobId);

JobProcessInformation stats = trackableJob.getJobProcessInformation();
int processedRecords = stats.getProcessedRecords();
log( "ProcessedRecords: " + processedRecords );

JobPartitionState[] partitionStates = stats.getPartitionStates();
for ( JobPartitionState partitionState : partitionStates ) {
    log( "PartitionOwner: " + partitionState.getOwner()
        + ", Processing state: " + partitionState.getState().name() );
}
```

Note: Caching of the `JobProcessInformation` does not work on Java native clients since current values are retrieved while retrieving the instance to minimize traffic between executing node and client.

6.2.2.10 JobTracker Configuration

The `JobTracker` configuration is used to setup behavior of the Hazelcast MapReduce framework.

Every `JobTracker` is capable of running multiple MapReduce jobs at once and so one configuration is meant as a shared resource for all jobs created by the same `JobTracker`. The configuration gives full control over the expected load behavior and thread counts to be used.

The following snippet shows a typical `JobTracker` configuration. We will discuss the configuration properties one by one:

```
<jobtracker name="default">
  <max-thread-size>0</max-thread-size>
  <!-- Queue size 0 means number of partitions * 2 -->
  <queue-size>0</queue-size>
  <retry-count>0</retry-count>
  <chunk-size>1000</chunk-size>
  <communicate-stats>true</communicate-stats>
  <topology-changed-strategy>CANCEL_RUNNING_OPERATION</topology-changed-strategy>
</jobtracker>
```

- **max-thread-size:** Configures the maximum thread pool size of the JobTracker.
- **queue-size:** Defines the maximum number of tasks that are able to wait to be processed. A value of 0 means unbounded queue. Very low numbers can prevent successful execution since job might not be correctly scheduled or intermediate chunks are lost.
- **retry-count:** Currently not used but reserved for later use where the framework will automatically try to restart / retry operations from a available save point.
- **chunk-size:** Defines the number of emitted values before a chunk is sent to the reducers. If your emitted values are big or you want to better balance your work, you might want to change this to a lower or higher value. A value of 0 means immediate transmission but remember that low values mean higher traffic costs. A very high value might cause an `OutOfMemoryError` to occur if emitted values not fit into heap memory before being sent to reducers. To prevent this, you might want to use a combiner to pre-reduce values on mapping nodes.
- **communicate-stats:** Defines if statistics (for example about processed entries) are transmitted to the job emitter. This might be used to show any kind of progress to a user inside of an UI system but produces additional traffic. If not needed, you might want to deactivate this.
- **topology-changed-strategy:** Defines how the MapReduce framework will react on topology changes while executing a job. Currently, only `CANCEL_RUNNING_OPERATION` is fully supported which throws an exception to the job emitter (will throw a `com.hazelcast.mapreduce.TopologyChangedException`).

6.2.3 Hazelcast MapReduce Architecture

6.2.3.1 Node Interoperation Example

To understand the following technical internals, we first will have a short look at what happens in terms of an example workflow.

To make the understanding simple, we think of an `IMap<String, Integer>` and emitted keys to have the same types. Imagine you have a three node cluster and initiate the MapReduce job on the first node. After you requested the JobTracker from your running / connected Hazelcast, we submit the task and retrieve the `ICompletableFuture` which gives us a chance of waiting for the result to be calculated or adding a callback to go a more reactive way.

The example expects that the chunk size is 0 or 1 so an emitted value is directly sent to the reducers. Internally, the job is prepared, started and executed on all nodes as shown below whereas the first node acts as the job owner (job emitter):

```
Node1 starts MapReduce job
Node1 emits key=Foo, value=1
Node1 does PartitionService::getKeyOwner(Foo) => results in Node3

Node2 emits key=Foo, value=14
Node2 asks jobOwner (Node1) for keyOwner of Foo => results in Node3

Node1 sends chunk for key=Foo to Node3

Node3 receives chunk for key=Foo and looks if there is already a Reducer,
      if not creates one for key=Foo
Node3 processes chunk for key=Foo

Node2 sends chunk for key=Foo to Node3

Node3 receives chunk for key=Foo and looks if there is already a Reducer and uses
      the previous one
Node3 processes chunk for key=Foo

Node1 send LastChunk information to Node3 because processing local values finished

Node2 emits key=Foo, value=27
Node2 has cached keyOwner of Foo => results in Node3
```

Node2 sends chunk for key=Foo to Node3

Node3 receives chunk for key=Foo and looks if there is already a Reducer and uses the previous one

Node3 processes chunk for key=Foo

Node2 send LastChunk information to Node3 because processing local values finished

Node3 finishes reducing for key=Foo

Node1 registers its local partitions are processed

Node2 registers its local partitions are processed

Node1 sees all partitions processed and requests reducing from all nodes

Node1 merges all reduced results together in a final structure and returns it

As you can see, the flow is quite complex but extremely powerful since everything is executed in parallel. Reducers do not wait until all values are emitted but immediately begin to reduce (when first chunk for an emitted key arrives).

6.2.3.2 Internal Architecture

Beginning with the package level, there is one basic package: `com.hazelcast.mapreduce`. This includes the external API and the **impl** package which itself contains the internal implementation.

- The **impl** package contains all the default KeyValueSource implementations and abstract base and support classes for exposed API.
- The **client** package contains all classes that are needed on client and server (node) side when a MapReduce job is offered from a client.
- The **notification** package contains all “notification” or event classes that are used to notify other members about progress on operations.
- The **operation** package contains all operations that are used by the workers or job owner to coordinate work and sync partition or reducer processing.
- The **task** package contains all classes that execute the actual MapReduce operation. It features the supervisor, mapping phase implementation and mapping and reducing tasks.

And now to the technical walk-through: As stated above, a MapReduce Job is always retrieved from a named JobTracker which in case is implemented in NodeJobTracker (extends AbstractJobTracker) and is configured using the configuration DSL. All of the internal implementation is completely ICompletableFuture driven and mostly non-blocking in design.

On submit, the Job creates a unique UUID which afterwards acts as a jobId and is combined with the JobTracker’s name to be uniquely identifiable inside the cluster. Then, the preparation is sent around the cluster and every member prepares its execution by creating a JobSupervisor, MapCombineTask and ReducerTask. The job emitting JobSupervisor gains special capabilities to synchronize and control JobSupervisors on other nodes for the same job.

If preparation is finished on all nodes, the job itself is started by executing a StartProcessingJobOperation on every node. This initiates a MappingPhase implementation (defaults to KeyValueSourceMappingPhase) and starts the actual mapping on the nodes.

The mapping process is currently a single threaded operation per node, but will be extended to run in parallel on multiple partitions (configurable per Job) in future versions. The Mapper is now called on every available value on the partition and eventually emits values. For every emitted value, either a configured CombinerFactory is called to create a Combiner or a cached one is used (or the default CollectingCombinerFactory is used to create Combiners). When the chunk limit is reached on a node, a IntermediateChunkNotification is prepared by collecting emitted keys to their corresponding nodes. This is either done by asking the job owner to assign members or by an already cached assignment. In later versions, a PartitionStrategy might be configurable, too.

The `IntermediateChunkNotification` is then sent to the reducers (containing only values for this node) and is offered to the `ReducerTask`. On every offer, the `ReducerTask` checks if it is already running and if not, it submits itself to the configured `ExecutorService` (from the `JobTracker` configuration).

If reducer queue runs out of work, the `ReducerTask` is removed from the `ExecutorService` to not block threads but eventually will be resubmitted on next chunk of work.

On every phase, the partition state is changed to keep track of the currently running operations. A `JobPartitionState` can be in one of the following states with self-explanatory titles: `[WAITING, MAPPING, REDUCING, PROCESSED, CANCELLED]`. On deeper interest of the states, look at the Javadoc.

- Node asks for new partition to process: `WAITING => MAPPING`
- Node emits first chunk to a reducer: `MAPPING => REDUCING`
- All nodes signal that they finished mapping phase and reducing is finished, too: `REDUCING => PROCESSED`

Eventually (or hopefully), all `JobPartitionStates` are reached to the state `PROCESSED`. Then, the job emitter's `JobSupervisor` asks all nodes for their reduced results and executes a potentially offered `Collator`. With this `Collator`, the overall result is calculated before it removes itself from the `JobTracker`, doing some final cleanup and returning the result to the requester (using the internal `TrackableJobFuture`).

If a job is cancelled while execution, all partitions are immediately set to `CANCELLED` state and a `CancelJobSupervisorOperation` is executed on all nodes to kill the running processes.

While the operation is running in addition to the default operations, some more like `ProcessStatsUpdateOperation` (updates processed records statistics) or `NotifyRemoteExceptionOperation` (notifies the nodes that the sending node encountered an unrecoverable situation and the Job needs to be cancelled - e.g. `NullPointerException` inside of a `Mapper`) are executed against the job owner to keep track of the process.

6.3 Continuous Query

You can listen map entry events providing a predicate and so, event will be fired for each entry validated by your query. `IMap` has a single method for listening map providing query.

```
/**
 * Adds an continuous entry listener for this map. Listener will get notified
 * for map add/remove/update/evict events filtered by given predicate.
 *
 * @param listener entry listener
 * @param predicate predicate for filtering entries
 */
void addEntryListener(EntryListener<K, V> listener, Predicate<K, V> predicate, K key, boolean includeVal
```


Chapter 7

Transactions

7.1 Transaction Interface

Hazelcast can be used in transactional context. Basically, create a `TransactionContext` which can be used to begin, commit, and rollback a transaction. Obtain transaction aware instances of queues, maps, sets, lists, multimaps via `TransactionContext`, work with them and commit/rollback in one shot. Hazelcast supports LOCAL (One Phase) and TWO_PHASE transactions. Default behavior is TWO_PHASE.

```
import java.util.Queue;
import java.util.Map;
import java.util.Set;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.Transaction;
```

```
Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
```

```
TransactionOptions options = new TransactionOptions().setTransactionType(TransactionType.LOCAL);
TransactionContext context = hz.newTransactionContext(options);
context.beginTransaction();
```

```
TransactionalQueue queue = context.getQueue("myqueue");
TransactionalMap map      = context.getMap  ("mymap");
TransactionalSet set       = context.getSet  ("myset");
```

```
try {
    Object obj = queue.poll();
    //process obj
    map.put ("1", "value1");
    set.add ("value");
    //do other things..
    context.commitTransaction();
} catch (Throwable t) {
    context.rollbackTransaction();
}
```

Isolation is always REPEATABLE_READ . If you are in a transaction, you can read the data in your transaction and the data that is already committed. If you are not in a transaction, you can only read the committed data.

Implementation is different for queue and map/set. For queue operations (offer, poll), offered and/or polled objects are copied to the owner member in order to safely commit/rollback. For map/set, Hazelcast first acquires the locks for the write operations (put, remove) and holds the differences (what is added/removed/updated) locally for each

transaction. When transaction is set to commit, Hazelcast will release the locks and apply the differences. When rolling back, Hazelcast will simply releases the locks and discard the differences.

7.2 J2EE Integration

Hazelcast can be integrated into J2EE containers via Hazelcast Resource Adapter (`hazelcast-ra-version.rar`). After proper configuration, Hazelcast can participate in standard J2EE transactions.

```
<%@page import="javax.resource.ResourceException" %>
<%@page import="javax.transaction.*" %>
<%@page import="javax.naming.*" %>
<%@page import="javax.resource.cci.*" %>
<%@page import="java.util.*" %>
<%@page import="com.hazelcast.core.*" %>
<%@page import="com.hazelcast.jca.*" %>

<%
UserTransaction txn = null;
HazelcastConnection conn = null;
Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);

try {
    Context context = new InitialContext();
    txn = (UserTransaction) context.lookup("java:comp/UserTransaction");
    txn.begin();

    HazelcastConnectionFactory cf = (HazelcastConnectionFactory) context.lookup ("java:comp/env/Hazelcas
    conn = cf.getConnection();

    TransactionalMap<String, String> txMap = conn.getTransactionMap("default");
    txMap.put("key", "value");

    txn.commit();
} catch (Throwable e) {
    if (txn != null) {
        try {
            txn.rollback();
        } catch (Exception ix) {ix.printStackTrace();};
    }
    e.printStackTrace();
} finally {
    if (conn != null) {
        try {
            conn.close();
        } catch (Exception ignored) {};
    }
}
%>
```

7.2.1 Resource Adapter Configuration

Deploying and configuring Hazelcast resource adapter is no different than any other resource adapter since it is a standard JCA resource adapter. However, resource adapter installation and configuration is container specific, so please consult your J2EE vendor documentation for details. Most common steps are:

1. Add the `hazelcast-version.jar` to container's classpath. Usually there is a `lib` directory that is loaded automatically by the container on startup.
2. Deploy `hazelcast-ra-version.rar`. Usually there is some kind of a `deploy` directory. Name of the directory varies by container.
3. Make container specific configurations when/after deploying `hazelcast-ra-version.rar`. Besides container specific configurations, JNDI name for Hazelcast resource is set.
4. Configure your application to use the Hazelcast resource. Update `web.xml` and/or `ejb-jar.xml` to let container know that your application will use the Hazelcast resource and define the resource reference.
5. Make container specific application configuration to specify JNDI name used for the resource in the application.

7.2.2 Sample Glassfish v3 Web Application Configuration

1. Place the `hazelcast-version.jar` into `GLASSFISH_HOME/glassfish/domains/domain1/lib/ext/` directory.
2. Place the `hazelcast-ra-version.rar` into `GLASSFISH_HOME/glassfish/domains/domain1/autodeploy/` directory.
3. Add the following lines to the `web.xml` file.

```
<resource-ref>
  <res-ref-name>HazelcastCF</res-ref-name>
  <res-type>com.hazelcast.jca.ConnectionFactoryImpl</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Notice that, we did not have to put `sun-ra.xml` into the RAR file since it comes with the `hazelcast-ra-version.rar` file already.

If Hazelcast resource is used from EJBs, you should configure `ejb-jar.xml` for resource reference and JNDI definitions, just like we did for `web.xml`.

7.2.3 Sample JBoss Web Application Configuration

- Place the `hazelcast-version.jar` into `JBOSS_HOME/server/deploy/default/lib` directory.
- Place the `hazelcast-ra-version.rar` into `JBOSS_HOME/server/deploy/default/deploy` directory
- Create a `hazelcast-ds.xml` file at `JBOSS_HOME/server/deploy/default/deploy` directory containing below content. Make sure to set the `rar-name` element to `hazelcast-ra-version.rar`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE connection-factories
  PUBLIC "-//JBoss//DTD JBoss JCA Config 1.5//EN"
  "http://www.jboss.org/j2ee/dtd/jboss-ds_1_5.dtd">

<connection-factories>
  <tx-connection-factory>
    <local-transaction/>
    <track-connection-by-tx>true</track-connection-by-tx>
    <jndi-name>HazelcastCF</jndi-name>
    <rar-name>hazelcast-ra-<version>.rar</rar-name>
    <connection-definition>
      javax.resource.cci.ConnectionFactory
    </connection-definition>
  </tx-connection-factory>
</connection-factories>
```

- Add the following lines to the `web.xml` file.

```
<resource-ref>
  <res-ref-name>HazelcastCF</res-ref-name>
  <res-type>com.hazelcast.jca.ConnectionFactoryImpl</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

- Add the following lines to the `jboss-web.xml` file.

```
<resource-ref>
  <res-ref-name>HazelcastCF</res-ref-name>
  <jndi-name>java:HazelcastCF</jndi-name>
</resource-ref>
```

If Hazelcast resource is used from EJBs, you should configure `ejb-jar.xml` and `jboss.xml` for resource reference and JNDI definitions.

Chapter 8

Integrated Clustering

8.1 Hibernate Second Level Cache

Hazelcast provides distributed second level cache for your Hibernate entities, collections and queries. Hazelcast has two implementations of Hibernate 2nd level cache, one for *hibernate-pre-3.3* and one for *hibernate-3.3.x* versions. In your Hibernate configuration file (e.g. `hibernate.cfg.xml`), add these properties:

- To enable use of second level cache

```
<property name="hibernate.cache.use_second_level_cache">true</property>
```

- To enable use of query cache

```
<property name="hibernate.cache.use_query_cache">true</property>
```

- And to force minimal puts into cache

```
<property name="hibernate.cache.use_minimal_puts">true</property>
```

- To configure Hazelcast for Hibernate, it is enough to put configuration file named `hazelcast.xml` into root of your classpath. If Hazelcast cannot find `hazelcast.xml`, then it will use default configuration from `hazelcast.jar`.
- You can define custom named Hazelcast configuration XML file with one of these Hibernate configuration properties.

```
– <property name="hibernate.cache.provider_configuration_file_resource_path">  
    hazelcast-custom-config.xml  
</property>
```

or

```
– <property name="hibernate.cache.hazelcast.configuration_file_path">  
    hazelcast-custom-config.xml  
</property>
```

- You can set up Hazelcast to connect cluster as Native Client. Native client is not a member; it connects to one of the cluster members and delegates all cluster wide operations to it. When the relied cluster member dies, client will transparently switch to another live member.

```
<property name="hibernate.cache.hazelcast.use_native_client">true</property>
```

To setup Native Client properly, you should add Hazelcast **group-name**, **group-password** and **cluster member address** properties. Native Client will connect to defined member and will get addresses of all members in the cluster. If the connected member will die or leave the cluster, client will automatically switch to another member in the cluster.

```
<property name="hibernate.cache.hazelcast.native_client_address">10.34.22.15</property>
<property name="hibernate.cache.hazelcast.native_client_group">dev</property>
<property name="hibernate.cache.hazelcast.native_client_password">dev-pass</property>
```

Note: To use Native Client you should add *hazelcast-client-<version>.jar* into your classpath. Refer to Native Clients for more information.

- To define Hibernate RegionFactory, add following property.

```
<property name="hibernate.cache.region.factory_class">
    com.hazelcast.hibernate.HazelcastCacheRegionFactory
</property>
```

Or, as an alternative you can use *HazelcastLocalCacheRegionFactory* which stores data in local node and sends invalidation messages when an entry is updated on local.

```
<property name="hibernate.cache.region.factory_class">
    com.hazelcast.hibernate.HazelcastLocalCacheRegionFactory
</property>
```

Hazelcast creates a separate distributed map for each Hibernate cache region. So, these regions can be configured easily via Hazelcast map configuration. You can define **backup**, **eviction**, **TTL** and **Near Cache** properties.

- Backup Configuration
- Eviction And TTL Configuration
- Near Cache Configuration

Hibernate has four cache concurrency strategies: *read-only*, *read-write*, *nonstrict-read-write* and *transactional*. But, Hibernate does not force cache providers to support all strategies. Hazelcast supports first three (**read-only**, **read-write**, **nonstrict-read-write**) of these strategies. It has no support for *transactional* strategy yet.

- If you are using XML based class configurations, you should add a *cache* element into your configuration with *usage* attribute with one of *read-only*, *read-write*, *nonstrict-read-write*.

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>

<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

- If you are using Hibernate-Annotations, then you can add *class-cache* or *collection-cache* element into your Hibernate configuration file with *usage* attribute with one of *read only*, *read/write*, *nonstrict read/write*.

```
<class-cache usage="read-only" class="eg.Immutable"/>
<class-cache usage="read-write" class="eg.Cat"/>
<collection-cache collection="eg.Cat.kittens" usage="read-write"/>
```

OR

- Alternatively, you can put Hibernate Annotation's `@Cache` annotation on your entities and collections.

```
Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Cat implements Serializable {
    ...
}
```

The last thing you should be aware of is to drop `hazelcast-hibernate-version.jar` into your classpath.

Accessing underlying HazelcastInstance

Using `com.hazelcast.hibernate.instance.HazelcastAccessor` you can access the underlying `HazelcastInstance` used by `Hibernate SessionFactory`.

```
SessionFactory sessionFactory = ...;
HazelcastInstance hazelcastInstance = HazelcastAccessor.getHazelcastInstance(sessionFactory);
```

Changing/setting lock timeout value of *read-write* strategy

Lock timeout value can be set using `hibernate.cache.hazelcast.lock_timeout_in_seconds` Hibernate property. Value should be in seconds and default value is 300 seconds.

Using named HazelcastInstance

Instead of creating a new `HazelcastInstance` for each `SessionFactory`, an existing instance can be used by setting `hibernate.cache.hazelcast.instance_name` Hibernate property to `HazelcastInstance`'s name. For more information see `Named HazelcastInstance`.

Disabling shutdown during SessionFactory.close()

Shutting down `HazelcastInstance` can be disabled during `SessionFactory.close()` by setting `hibernate.cache.hazelcast` Hibernate property to `false`. (*In this case Hazelcast property `hazelcast.shutdownhook.enabled` should not be set to `false`.*) Default value is `true`.

8.2 HTTP Session Clustering with Hazelcast WM

Assume that you have more than one web servers (A, B, C) with a load balancer in front of them. If server A goes down, your users on that server will be directed to one of the live servers (B or C), but their sessions will be lost!

So we have to have all these sessions backed up somewhere if we do not want to lose the sessions upon server crashes. Hazelcast WM allows you to cluster user HTTP sessions automatically. The following are required for enabling Hazelcast Session Clustering:

- Target application or web server should support Java 1.5 or higher
- Target application or web server should support Servlet 2.4 or higher spec
- Session objects that need to be clustered have to be `Serializable`

Here are the steps to setup Hazelcast Session Clustering:

- Put the `hazelcast` and `hazelcast-wm` jars in your `WEB-INF/lib` directory. Optionally, if you wish to connect to a cluster as a client, add `hazelcast-client` as well.

- Put the following XML into `web.xml` file. Make sure Hazelcast filter is placed before all the other filters if any; put it at the top for example.

```
<filter>
  <filter-name>hazelcast-filter</filter-name>
  <filter-class>com.hazelcast.web.WebFilter</filter-class>
  <!--
    Name of the distributed map storing
    your web session objects
  -->
  <init-param>
    <param-name>map-name</param-name>
    <param-value>my-sessions</param-value>
  </init-param>
  <!--
    How is your load-balancer configured?
    stick-session means all requests of a session
    is routed to the node where the session is first created.
    This is excellent for performance.
    If sticky-session is set to false, when a session is updated
    on a node, entry for this session on all other nodes is invalidated.
    You have to know how your load-balancer is configured before
    setting this parameter. Default is true.
  -->
  <init-param>
    <param-name>sticky-session</param-name>
    <param-value>true</param-value>
  </init-param>
  <!--
    Name of session id cookie
  -->
  <init-param>
    <param-name>cookie-name</param-name>
    <param-value>hazelcast.sessionId</param-value>
  </init-param>
  <!--
    Domain of session id cookie. Default is based on incoming request.
  -->
  <init-param>
    <param-name>cookie-domain</param-name>
    <param-value>.mywebsite.com</param-value>
  </init-param>
  <!--
    Should cookie only be sent using a secure protocol? Default is false.
  -->
  <init-param>
    <param-name>cookie-secure</param-name>
    <param-value>false</param-value>
  </init-param>
  <!--
    Should HttpOnly attribute be set on cookie ? Default is false.
  -->
  <init-param>
    <param-name>cookie-http-only</param-name>
    <param-value>false</param-value>
  </init-param>
  <!--
    Are you debugging? Default is false.
```



```

-->
<init-param>
  <param-name>debug</param-name>
  <param-value>true</param-value>
</init-param>
<!--
  Configuration xml location;
  * as servlet resource OR
  * as classpath resource OR
  * as URL
  Default is one of hazelcast-default.xml
  or hazelcast.xml in classpath.
-->
<init-param>
  <param-name>config-location</param-name>
  <param-value>/WEB-INF/hazelcast.xml</param-value>
</init-param>
<!--
  Do you want to use an existing HazelcastInstance?
  Default is null.
-->
<init-param>
  <param-name>instance-name</param-name>
  <param-value>default</param-value>
</init-param>
<!--
  Do you want to connect as a client to an existing cluster?
  Default is false.
-->
<init-param>
  <param-name>use-client</param-name>
  <param-value>>false</param-value>
</init-param>
<!--
  Client configuration location;
  * as servlet resource OR
  * as classpath resource OR
  * as URL
  Default is null.
-->
<init-param>
  <param-name>client-config-location</param-name>
  <param-value>/WEB-INF/hazelcast-client.properties</param-value>
</init-param>
<!--
  Do you want to shutdown HazelcastInstance during
  web application undeploy process?
  Default is true.
-->
<init-param>
  <param-name>shutdown-on-destroy</param-name>
  <param-value>true</param-value>
</init-param>
</filter>
<filter-mapping>
  <filter-name>hazelcast-filter</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>FORWARD</dispatcher>

```

```

    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>

<listener>
    <listener-class>com.hazelcast.web.SessionListener</listener-class>
</listener>

```

- Package and deploy your war file as you would normally do.

It is that easy! All HTTP requests will go through Hazelcast `WebFilter` and it will put the session objects into Hazelcast distributed map if needed.

Information about sticky-sessions:

Hazelcast holds whole session attributes in a distributed map and in local HTTP session. Local session is required for fast access to data and distributed map is needed for fail-safety.

- *If sticky-session is not used, whenever a session attribute is updated in a node (in both node local session and clustered cache), that attribute should be invalidated in all other nodes' local sessions, because now they have dirty value. So, when a request arrives to one of those other nodes, that attribute value is fetched from clustered cache.*
- *To overcome performance penalty of sending invalidation messages during updates, sticky-sessions can be used. If Hazelcast knows sessions are sticky, invalidation will not be send, because Hazelcast assumes there is no other local session at the moment. When a server is down, requests belonging to a session hold in that server will routed to other one and that server will fetch session data from clustered cache. That means, using sticky-sessions, one will not suffer performance penalty of accessing clustered data and can benefit recover from a server failure.*

8.3 Spring Integration

8.3.1 Configuration

Note: Hazelcast-Spring integration requires either `hazelcast-spring-version.jar*` or `hazelcast-all-version.jar` in the classpath.*

You can declare Hazelcast beans for Spring context using `beans` namespace (default Spring `beans` namespace) as well to declare Hazelcast maps, queues and others.

```

<bean id="instance" class="com.hazelcast.core.Hazelcast" factory-method="newHazelcastInstance">
    <constructor-arg>
        <bean class="com.hazelcast.config.Config">
            <property name="groupConfig">
                <bean class="com.hazelcast.config.GroupConfig">
                    <property name="name" value="dev"/>
                    <property name="password" value="pwd"/>
                </bean>
            </property>
            <!-- and so on ... -->
        </bean>
    </constructor-arg>
</bean>

<bean id="map" factory-bean="instance" factory-method="getMap">
    <constructor-arg value="map"/>
</bean>

```

Hazelcast has Spring integration (requires version 2.5 or greater) since 1.9.1 using *hazelcast* namespace.

- Add namespace `xmlns:hz="http://www.hazelcast.com/schema/spring"` to `beans` tag in context file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:hz="http://www.hazelcast.com/schema/spring"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.hazelcast.com/schema/spring
    http://www.hazelcast.com/schema/spring/hazelcast-spring-3.0.xsd">
```

- Use *hz* namespace shortcuts to declare cluster, its items and so on.

After that you can configure Hazelcast instance as shown below.

```
<hz:hazelcast id="instance">
  <hz:config>
    <hz:group name="dev" password="password"/>
    <hz:network port="5701" port-auto-increment="false">
      <hz:join>
        <hz:multicast enabled="false"
          multicast-group="224.2.2.3"
          multicast-port="54327"/>
        <hz:tcp-ip enabled="true">
          <hz:members>10.10.1.2, 10.10.1.3</hz:members>
        </hz:tcp-ip>
      </hz:join>
    </hz:network>
    <hz:map name="map"
      backup-count="2"
      max-size="0"
      eviction-percentage="30"
      read-backup-data="true"
      eviction-policy="NONE"
      merge-policy="com.hazelcast.map.merge.PassThroughMergePolicy"/>
  </hz:config>
</hz:hazelcast>
```

You can easily configure `map-store` and `near-cache`, too. For `map-store`, you should set either *class-name* or *implementation* attribute.)

```
<hz:config>
  <hz:map name="map1">
    <hz:near-cache time-to-live-seconds="0" max-idle-seconds="60"
      eviction-policy="LRU" max-size="5000" invalidate-on-change="true"/>

    <hz:map-store enabled="true" class-name="com.foo.DummyStore"
      write-delay-seconds="0"/>
  </hz:map>

  <hz:map name="map2">
    <hz:map-store enabled="true" implementation="dummyMapStore"
      write-delay-seconds="0"/>
  </hz:map>

  <bean id="dummyMapStore" class="com.foo.DummyStore" />
</hz:config>
```

It is possible to use placeholders instead of concrete values. For instance, use property file *app-default.properties* for group configuration:

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <list>
      <value>classpath:/app-default.properties</value>
    </list>
  </property>
</bean>

<hz:hazelcast id="instance">
  <hz:config>
    <hz:group
      name="${cluster.group.name}"
      password="${cluster.group.password}"/>
    <!-- ... -->
  </hz:config>
</hz:hazelcast>
```

Similar for client:

```
<hz:client id="client">
  <hz:group name="${cluster.group.name}" password="${cluster.group.password}" />
  <hz:network connection-attempt-limit="3"
    connection-attempt-period="3000"
    connection-timeout="1000"
    redo-operation="true"
    smart-routing="true">
    <hz:member>10.10.1.2:5701</hz:member>
    <hz:member>10.10.1.3:5701</hz:member>
  </hz:network>
</hz:client>
```

Hazelcast also supports *lazy-init*, *scope* and *depends-on* bean attributes.

```
<hz:hazelcast id="instance" lazy-init="true" scope="singleton">
  ...
</hz:hazelcast>

<hz:client id="client" scope="prototype" depends-on="instance">
  ...
</hz:client>
```

You can declare beans for the following Hazelcast objects:

- map
- multiMap
- queue
- topic
- set
- list
- executorService
- idGenerator
- atomicLong
- semaphore

- `countDownLatch`
- `lock`

Example:

```
<hz:map id="map" instance-ref="client" name="map" lazy-init="true" />
<hz:multiMap id="multiMap" instance-ref="instance" name="multiMap" lazy-init="false" />
<hz:queue id="queue" instance-ref="client" name="queue" lazy-init="true" depends-on="instance"/>
<hz:topic id="topic" instance-ref="instance" name="topic" depends-on="instance, client"/>
<hz:set id="set" instance-ref="instance" name="set" />
<hz:list id="list" instance-ref="instance" name="list"/>
<hz:executorService id="executorService" instance-ref="client" name="executorService"/>
<hz:idGenerator id="idGenerator" instance-ref="instance" name="idGenerator"/>
<hz:atomicLong id="atomicLong" instance-ref="instance" name="atomicLong"/>
<hz:semaphore id="semaphore" instance-ref="instance" name="semaphore"/>
<hz:countDownLatch id="countDownLatch" instance-ref="instance" name="countDownLatch"/>
<hz:lock id="lock" instance-ref="instance" name="lock"/>
```

Spring tries to create a new `Map/Collection` instance and fill the new instance by iterating and converting values of the original `Map/Collection` (`IMap`, `IQueue`, etc.) to required types when generic type parameters of the original `Map/Collection` and the target property/attribute do not match.

Since Hazelcast `Maps/Collections` are designed to hold very large data which a single machine cannot carry, iterating through whole values can cause out of memory errors.

To avoid this issue, either target property/attribute can be declared as un-typed `Map/Collection` as shown below:

```
public class SomeBean {
    @Autowired
    IMap map; // instead of IMap<K, V> map

    @Autowired
    IQueue queue; // instead of IQueue<E> queue

    ...
}
```

Or, parameters of injection methods (constructor, setter) can be un-typed as shown below:

```
public class SomeBean {

    IMap<K, V> map;

    IQueue<E> queue;

    public SomeBean(IMap map) { // instead of IMap<K, V> map
        this.map = map;
    }

    ...

    public void setQueue(IQueue queue) { // instead of IQueue<E> queue
        this.queue = queue;
    }

    ...
}
```

For more information please see [Spring issue-3407](#).

8.3.2 Spring Managed Context

It is often desired to access Spring managed beans, to apply bean properties or to apply factory callbacks such as `ApplicationContextAware`, `BeanNameAware` or to apply bean post-processing such as `InitializingBean`, `@PostConstruct` like annotations while using Hazelcast distributed `ExecutorService` or more generally any Hazelcast managed object. Achieving those features are as simple as adding `@SpringAware` annotation to your distributed object types. Once you have configured `HazelcastInstance` as explained in Spring Configuration section, just mark any distributed type with `@SpringAware` annotation.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:hz="http://www.hazelcast.com/schema/spring"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.hazelcast.com/schema/spring
        http://www.hazelcast.com/schema/spring/hazelcast-spring-3.2.xsd">

  <context:annotation-config />

  <hz:hazelcast id="instance">
    <hz:config>
      <hz:group name="dev" password="password"/>
      <hz:network port="5701" port-auto-increment="false">
        <hz:join>
          <hz:multicast enabled="false" />
          <hz:tcp-ip enabled="true">
            <hz:members>10.10.1.2, 10.10.1.3</hz:members>
          </hz:tcp-ip>
        </hz:join>
      </hz:network>
      ...
    </hz:config>
  </hz:hazelcast>

  <bean id="someBean" class="com.hazelcast.examples.spring.SomeBean" scope="singleton" />
  ...
</beans>
```

ExecutorService example:

```
@SpringAware
public class SomeTask implements Callable<Long>, ApplicationContextAware, Serializable {

  private transient ApplicationContext context;

  private transient SomeBean someBean;

  public Long call() throws Exception {
    return someBean.value;
  }

  public void setApplicationContext(final ApplicationContext applicationContext)
    throws BeansException {
    context = applicationContext;
  }
}
```

```

    @Autowired
    public void setSomeBean(final SomeBean someBean) {
        this.someBean = someBean;
    }
}

HazelcastInstance hazelcast = (HazelcastInstance) context.getBean("hazelcast");
SomeBean bean = (SomeBean) context.getBean("someBean");

Future<Long> f = hazelcast.getExecutorService().submit(new SomeTask());
Assert.assertEquals(bean.value, f.get().longValue());

// choose a member
Member member = hazelcast.getCluster().getMembers().iterator().next();

Future<Long> f2 = (Future<Long>) hazelcast.getExecutorService()
    .submitToMember(new SomeTask(), member);
Assert.assertEquals(bean.value, f2.get().longValue());

```

Distributed Map value example:

```

@SpringAware
@Component("someValue")
@Scope("prototype")
public class SomeValue implements Serializable, ApplicationContextAware {

    transient ApplicationContext context;

    transient SomeBean someBean;

    transient boolean init = false;

    public void setApplicationContext(final ApplicationContext applicationContext)
        throws BeansException {
        context = applicationContext;
    }

    @Autowired
    public void setSomeBean(final SomeBean someBean) {
        this.someBean = someBean;
    }

    @PostConstruct
    public void init() {
        someBean.doSomethingUseful();
        init = true;
    }

    ...
}

```

On Node-1;

```

HazelcastInstance hazelcast = (HazelcastInstance) context.getBean("hazelcast");
SomeValue value = (SomeValue) context.getBean("someValue")
IMap<String, SomeValue> map = hazelcast.getMap("values");
map.put("key", value);

```

On Node-2;

```
HazelcastInstance hazelcast = (HazelcastInstance) context.getBean("hazelcast");
IMap<String, SomeValue> map = hazelcast.getMap("values");
SomeValue value = map.get("key");
Assert.assertTrue(value.init);
```

Note that, Spring managed properties/fields are marked as `transient`.

8.3.3 Spring Cache

As of version 3.1, Spring Framework provides support for adding caching into an existing Spring application. To use Hazelcast as Spring cache provider, you should just define a `com.hazelcast.spring.cache.HazelcastCacheManager` bean and register it as Spring cache manager.

```
<cache:annotation-driven cache-manager="cacheManager" />

<hz:hazelcast id="hazelcast">
    ...
</hz:hazelcast>

<bean id="cacheManager" class="com.hazelcast.spring.cache.HazelcastCacheManager">
    <constructor-arg ref="instance"/>
</bean>
```

For more information please see [Spring Cache Abstraction](#).

8.3.4 Hibernate 2nd Level Cache Config

If you are using Hibernate with Hazelcast as 2nd level cache provider, you can easily create `RegionFactory` instances within Spring configuration (by Spring version 3.1). That way, it is possible to use same `HazelcastInstance` as Hibernate L2 cache instance.

```
<hz:hibernate-region-factory id="regionFactory" instance-ref="instance" />
...
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
    scope="singleton">
    <property name="dataSource" ref="dataSource"/>
    <property name="cacheRegionFactory" ref="regionFactory" />
    ...
</bean>
```

8.3.5 Spring Data - JPA

Hazelcast supports JPA persistence integrated with [Spring Data-JPA](#) module. Your POJOs are mapped and persisted to your relational database. To use JPA persistence, first you should create a Repository interface extending `CrudRepository` class with object type that you want to persist.

```
package com.hazelcast.jpa.repository;

import com.hazelcast.jpa.Product;
import org.springframework.data.repository.CrudRepository;
```



```
public interface ProductRepository extends CrudRepository<Product, Long> {
}
```

Then you should add your data source and repository definition to your Spring configuration, as shown below.

```
<jpa:repositories
    base-package="com.hazelcast.jpa.repository" />

<bean class="com.hazelcast.jpa.SpringJPAMapStore" id="jpamapstore">
    <property name="crudRepository" ref="productRepository" />
</bean>

<bean class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close" id="dataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/YOUR_DB"/>
    <property name="username" value="YOUR_USERNAME"/>
    <property name="password" value="YOUR_PASSWORD"/>
</bean>

<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
            <property name="generateDdl" value="true" />
            <property name="database" value="MYSQL" />
        </bean>
    </property>
    <property name="persistenceUnitName" value="jpa.sample" />
</bean>

<bean class="org.springframework.orm.jpa.JpaTransactionManager"
    id="transactionManager">
    <property name="entityManagerFactory"
        ref="entityManagerFactory" />
    <property name="jpaDialect">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect" />
    </property>
</bean>
```

In the example configuration above, Hibernate and MySQL is configured. You change them according to your ORM and database selection. Also, you should define your persistence unit with `persistence.xml` under `META-INF` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="jpa.sample" />
</persistence>
```

By default, the key is expected to be the same with ID of the JPA object. You can change this behavior and customize MapStore implementation extending `SpringJPAMapStore` class.

Related Information

For more information please see [Spring Data JPA Reference](#).

8.3.6 Spring Data - MongoDB

Hazelcast supports MongoDB persistence integrated with [Spring Data-MongoDB](#) module. Spring MongoDB module maps your objects to equivalent MongoDB objects. To persist your objects into MongoDB, you should define MongoDB mapstore in your Spring configuration as follows:

```
<mongo:mongo id="mongo" host="localhost" port="27017"/>

<bean id="mongoTemplate"
      class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg ref="mongo"/>
    <constructor-arg name="databaseName" value="test"/>
</bean>

<bean class="com.hazelcast.spring.mongodb.MongoMapStore" id="mongomapstore">
    <property name="mongoTemplate" ref="mongoTemplate" />
</bean>
```

Then, you can set this as map store for maps that you want to persist into MongoDB.

```
<hz:map name="user">
    <hz:map-store enabled="true" implementation="mongomapstore"
                  write-delay-seconds="0">
    </hz:map-store>
</hz:map>
```

By default, the key is set as id of the MongoDB object. You can override `MongoMapStore` class for you custom needs.

Related Information

For more information please see [Spring Data MongoDB Reference](#).

Chapter 9

Storage

9.1 Elastic Memory

Enterprise Only

By default, Hazelcast stores your distributed data (map entries, queue items) into Java heap which is subject to garbage collection (GC). As your heap gets bigger, garbage collection might cause your application to pause tens of seconds, badly affecting your application performance and response times. Elastic Memory is Hazelcast with off-heap (direct) memory storage to avoid GC pauses. Even if you have terabytes of cache in-memory with lots of updates, GC will have almost no effect; resulting in more predictable latency and throughput.

Here are the steps to enable Elastic Memory:

- Set the maximum direct memory JVM can allocate, e.g. `java -XX:MaxDirectMemorySize=60G`
- Enable Elastic Memory by setting `hazelcast.elastic.memory.enabled` Hazelcast configuration property to `true`.
- Set the total direct memory size for `HazelcastInstance` by setting `hazelcast.elastic.memory.total.size` Hazelcast configuration property. Size can be in MB or GB and abbreviation can be used, such as 60G and 500M.
- Set the chunk size by setting `hazelcast.elastic.memory.chunk.size` Hazelcast configuration property. Hazelcast will partition the entire off-heap memory into chunks. Default chunk size is 1K.
- You can enable `sun.misc.Unsafe` based off-heap storage implementation instead of `java.nio.DirectByteBuffer` based one, by setting `hazelcast.elastic.memory.unsafe.enabled` property to `true`. Default value is `false`.
- Configure maps that will use Elastic Memory by setting `InMemoryFormat` to **OFFHEAP**. Default value is **BINARY**.

Below is the declarative configuration.

```
xml <hazelcast> ... <map name="default"> ... <in-memory-format>OFFHEAP</in-memory-format>
</map> </hazelcast>
```

And, the programmatic configuration:

```
MapConfig mapConfig = new MapConfig();
mapConfig.setInMemoryFormat(InMemoryFormat.OFFHEAP);
```


Chapter 10

Clients

There are currently three ways to connect to a running Hazelcast cluster:

- Native Clients
- Memcache Clients
- REST Client

10.1 Native Clients

Native Clients enable you to perform almost all Hazelcast operations without being a member of the cluster. It connects to one of the cluster members and delegates all cluster wide operations to it (*dummy client*) or connects to all of them and delegate operations smartly (*smart client*). When the relied cluster member dies, client will transparently switch to another live member.

There can be hundreds, even thousands of clients connected to the cluster. But, by default there are **core count * 10** threads on the server side that will handle all the requests (e.g. if the server has 4 cores, it will be 40).

Imagine a trading application where all the trading data stored and managed in a Hazelcast cluster with tens of nodes. Swing/Web applications at traders' desktops can use Native Clients to access and modify the data in the Hazelcast cluster.

Currently, Hazelcast has Native Java, C++ and C# Clients available.

10.1.1 Java Client

You can perform almost all Hazelcast operations with Java Client. It already implements the same interface. You must include `hazelcast.jar` and `hazelcast-client.jar` into your classpath. A sample code is shown below.

```
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.client.HazelcastClient;
```

```
import java.util.Map;
import java.util.Collection;
```

```
ClientConfig clientConfig = new ClientConfig();
clientConfig.getGroupConfig().setName("dev").setPassword("dev-pass");
clientConfig.getNetworkConfig().addAddress("10.90.0.1", "10.90.0.2:5702");
```

```
HazelcastInstance client = HazelcastClient.newHazelcastClient(clientConfig);
```

```
//All cluster operations that you can do with ordinary HazelcastInstance
Map<String, Customer> mapCustomers = client.getMap("customers");
mapCustomers.put("1", new Customer("Joe", "Smith"));
mapCustomers.put("2", new Customer("Ali", "Selam"));
mapCustomers.put("3", new Customer("Avi", "Noyan"));

Collection<Customer> colCustomers = mapCustomers.values();
for (Customer customer : colCustomers) {
    // process customer
}
```

Name and Password parameters seen above can be used to create a secure connection between the client and cluster. Same parameter values should be set at the node side, so that the client will connect to those nodes that have the same **GroupConfig** credentials, forming a separate cluster.

In the cases where the security established with **GroupConfig** is not enough and you want your clients connecting securely to the cluster, **ClientSecurityConfig** can be used. This configuration has a **credentials** parameter with which IP address and UID are set (please see [ClientSecurityConfig.java](#)).

To configure the other parameters of client-cluster connection, **ClientNetworkConfig** is used. In this class, below parameters are set:

- **addressList**: Includes the list of addresses to which the client will connect. Client uses this list to find an alive node. Although it may be enough to give only one address of a node in the cluster (since all nodes communicate with each other), it is recommended to give all nodes' addresses.
- **smartRouting**: This parameter determines whether the client is smart or dummy. A dummy client connects to one node specified in **addressList** and stays connected to that node. If that node goes down, it chooses and connects another node. In the case of a dummy client, all operations that will be performed by the client are distributed to the cluster over the connected node. A smart client, on the other hand, connects to all nodes in the cluster and for example if the client will perform a “put” operation, it finds the node that is the key owner and performs that operation on that node.
- **redoOperation**: Client may lost its connection to a cluster due to network issues or a node being down. In this case, we cannot know whether the operations that were being performed are completed or not. This boolean parameter determines if those operations will be retried or not. Setting this parameter to *true* for idempotent operations (e.g. “put” on a map) does not give a harm. But for operations that are not idempotent (e.g. “offer” on a queue), retrying them may cause undesirable effects.
- **connectionTimeout**: This parameter is the timeout in milliseconds for the heartbeat messages sent by the client to the cluster. If there is no response from a node for this timeout period, client deems the connection as down and closes it.
- **connectionAttemptLimit** and **connectionAttemptPeriod**: Assume that the client starts to connect to the cluster whose all nodes may not be up. First parameter is the count of connection attempts by the client and the second one is the time between those attempts (in milliseconds). These two parameters should be used together (if one of them is set, other should be set, too). Furthermore, assume that the client is connected to the cluster and everything was fine, but for a reason the whole cluster goes down. Then, the client will try to re-connect to the cluster using the values defined by these two parameters. If, for example, **connectionAttemptLimit** is set as *Integer.MAX_VALUE*, it will try to re-connect forever.
- **socketInterceptorConfig**: When a connection between the client and cluster is established (i.e. a socket is opened) and if a socket interceptor is defined, this socket is handed to the interceptor. Interceptor can use this socket, for example, to log the connection or to handshake with the cluster. There are some cases where a socket interceptor should also be defined at the cluster side, for example, in the case of client-cluster handshaking. This can be used as a security feature, since the clients that do not have interceptors will not handshake with the cluster.
- **sslConfig**: If SSL is desired to be enabled for the client-cluster connection, this parameter should be set. Once set, the connection (socket) is established out of an SSL factory defined either by a factory class name or factory implementation (please see [SSLConfig.java](#)).
- **loadBalancer**: This parameter is used to distribute operations to multiple endpoints. It is meaningful to use it when the operation in question is not a key specific one but is a cluster wide operation (e.g. calculating the size of a map, adding a listener). Default load balancer is Round Robin. The developer can write his/her own load balancer using the [LoadBalancer](#) interface.

- **executorPoolSize:** Hazelcast has an internal executor service (different from the data structure *Executor Service*) that has threads and queues to perform internal operations such as handling responses. This parameter specifies the size of the pool of threads which perform these operations laying in the executor's queue. If not configured, this parameter has the value as **5 * core size of the client** (i.e. it is 20 for a machine that has 4 cores).

10.1.2 C++ Client

Enterprise Only

You can use Native C++ Client to connect to Hazelcast nodes and perform almost all operations that a node can perform. Different from nodes, clients do not hold data. It is by default a smart client, i.e. it knows where the data is and asks directly to the correct node. This feature can be disabled (using `ClientConfig::setSmart` method) if you do not want the clients to connect every node.

Features of C++ Clients are:

- Access to distributed data structures (IMap, IQueue, MultiMap, ITopic, etc.).
- Access to transactional distributed data structures (TransactionalMap, TransactionalQueue, etc.).
- Ability to add cluster listeners to a cluster and entry/item listeners to distributed data structures.
- Distributed synchronization mechanisms with ILock, ISemaphore and ICountDownLatch.

10.1.2.1 How to Setup

Hazelcast C++ Client is shipped with 32/64 bit, shared and static libraries. Compiled static libraries of dependencies are also available in the release. Dependencies are **zlib** and **shared_ptr** from the boost libraries.

Downloaded release folder consists of:

- Mac_64/
- Windows_32/
- Windows_64/
- Linux_32/
- Linux_64/
- docs/ (*HTML Doxygen documents are here*)

And each of the folders above contains the following:

- examples/
 - testApp.exe => example command line client tool to connect hazelcast servers.
 - TestApp.cpp => code of the example command line tool.
- hazelcast/
 - lib/ => Contains both shared and static library of hazelcast.
 - include/ => Contains headers of client
- external/
 - lib/ => Contains compiled static libraries of zlib.
 - include/ => Contains headers of dependencies.(zlib and boost::shared_ptr)

10.1.2.2 Platform Specific Installation Guides

C++ Client is tested on Linux 32/64, Mac 64 and Windows 32/64 bit machines. For each of the headers above, it is assumed that you are in the correct folder for your platform. Folders are Mac_64, Windows_32, Windows_64, Linux_32 or Linux_64.

10.1.2.2.1 Linux For Linux, there are two distributions; 32 bit and 64 bit.

Sample script to build with static library:

```
g++ main.cpp -pthread -I./external/include -I./hazelcast/include ./hazelcast/lib/libHazelcastClientStatic.a
./external/lib/libz.a
```

Sample script to build with shared library:

```
g++ main.cpp -lpthread -Wl,-no-as-needed -lrt -I./external/include -I./hazelcast/include
-L./hazelcast/lib -lHazelcastClientShared_64 ./external/lib/libz.a
```

10.1.2.2.2 Mac For Mac, there is only one distribution which is 64 bit.

Sample script to build with static library:

```
g++ main.cpp -I./external/include -I./hazelcast/include ./hazelcast/lib/libHazelcastClientStatic_64.a
./external/lib/darwin/libz.a
```

Sample script to build with shared library:

```
g++ main.cpp -I./external/include -I./hazelcast/include -L./hazelcast/lib -lHazelcastClientShared_64
./external/lib/darwin/libz.a
```

10.1.2.2.3 Windows For Windows, there are two distributions; 32 bit and 64 bit. Current release have only Visual Studio 2010 compatible libraries. For others, please contact with support@hazelcast.com.

10.1.2.3 Code Examples

A Hazelcast node should be running to make below sample codes work.

10.1.2.3.1 Map Example

```
#include <hazelcast/client/HazelcastAll.h>
#include <iostream>

using namespace hazelcast::client;

int main(){
    ClientConfig clientConfig;
    Address address("localhost", 5701);
    clientConfig.addAddress(address);

    HazelcastClient hazelcastClient(clientConfig);

    IMap<int,int> myMap = hazelcastClient.getMap<int ,int>("myIntMap");
    myMap.put(1,3);
    boost::shared_ptr<int> v = myMap.get(1);
    if(v.get() != NULL){
        //process the item
    }

    return 0;
}
```

10.1.2.3.2 Queue Example


```

#include <hazelcast/client/HazelcastAll.h>
#include <iostream>
#include <string>

using namespace hazelcast::client;

int main(){
    ClientConfig clientConfig;
    Address address("localhost", 5701);
    clientConfig.addAddress(address);

    HazelcastClient hazelcastClient(clientConfig);

    IQueue<std::string> q = hazelcastClient.getQueue<std::string>("q");
    q.offer("sample");
    boost::shared_ptr<std::string> v = q.poll();
    if(v.get() != NULL){
        //process the item
    }
    return 0;
}

```

10.1.2.3.3 Entry Listener Example

```

#include "hazelcast/client/ClientConfig.h"
#include "hazelcast/client/EntryEvent.h"
#include "hazelcast/client/IMap.h"
#include "hazelcast/client/Address.h"
#include "hazelcast/client/HazelcastClient.h"
#include <iostream>
#include <string>

using namespace hazelcast::client;

class SampleEntryListener {
public:

    void entryAdded(EntryEvent<std::string, std::string> &event) {
        std::cout << "entry added " << event.getKey() << " " << event.getValue() << std::endl;
    };

    void entryRemoved(EntryEvent<std::string, std::string> &event) {
        std::cout << "entry added " << event.getKey() << " " << event.getValue() << std::endl;
    }

    void entryUpdated(EntryEvent<std::string, std::string> &event) {
        std::cout << "entry added " << event.getKey() << " " << event.getValue() << std::endl;
    }

    void entryEvicted(EntryEvent<std::string, std::string> &event) {
        std::cout << "entry added " << event.getKey() << " " << event.getValue() << std::endl;
    }
};

int main(int argc, char **argv) {

    ClientConfig clientConfig;

```

```

Address address("localhost", 5701);
clientConfig.addAddress(address);

HazelcastClient hazelcastClient(clientConfig);

IMap<std::string, std::string> myMap = hazelcastClient.getMap<std::string, std::string>("myIntMap");
SampleEntryListener * listener = new SampleEntryListener();

std::string id = myMap.addEntryListener(*listener, true);
myMap.put("key1", "value1"); //prints entryAdded
myMap.put("key1", "value2"); //prints updated
myMap.remove("key1"); //prints entryRemoved
myMap.put("key2", "value2", 1000); //prints entryEvicted after 1 second

myMap.removeEntryListener(id); //WARNING: deleting listener before removing it from hazelcast leads to
delete listener;                //delete listener after remove it from hazelcast.
return 0;
};

```

10.1.2.3.4 Serialization Example Assume that you have the following two classes in Java and you want to use it with C++ client.

```

class Foo implements Serializable{
    private int age;
    private String name;
}

class Bar implements Serializable{
    private float x;
    private float y;
}

```

First, let them implement `Portable` or `IdentifiedDataSerializable` as shown below.

```

class Foo implements Portable {
    private int age;
    private String name;

    public int getFactoryId() {
        return 666;    // a positive id that you choose
    }

    public int getClassId() {
        return 2;      // a positive id that you choose
    }

    public void writePortable(PortableWriter writer) throws IOException {
        writer.writeUTF("n", name);
        writer.writeInt("a", age);
    }

    public void readPortable(PortableReader reader) throws IOException {
        name = reader.readUTF("n");
        age = reader.readInt("a");
    }
}

```

```

class Bar implements IdentifiedDataSerializable {
    private float x;
    private float y;

    public int getFactoryId() {
        return 4;    // a positive id that you choose
    }

    public int getId() {
        return 5;    // a positive id that you choose
    }

    public void writeData(ObjectDataOutput out) throws IOException {
        out.writeFloat(x);
        out.writeFloat(y);
    }

    public void readData(ObjectDataInput in) throws IOException {
        x = in.readFloat();
        y = in.readFloat();
    }
}

```

Then, implement the corresponding classes in C++ with same factory and class ID as shown below:

```

class Foo : public Portable {
public:
    int getFactoryId() const{
        return 666;
    };

    int getClassId() const{
        return 2;
    };

    void writePortable(serialization::PortableWriter &writer) const{
        writer.writeUTF("n", name);
        writer.writeInt("a", age);
    };

    void readPortable(serialization::PortableReader &reader){
        name = reader.readUTF("n");
        age = reader.readInt("a");
    };

private:
    int age;
    std::string name;
};

class Bar : public IdentifiedDataSerializable {
public:
    int getFactoryId() const{
        return 4;
    };

    int getClassId() const{
        return 2;
    };
}

```

```

};

void writeData(serialization::ObjectDataOutput& out) const{
    out.writeFloat(x);
    out.writeFloat(y);
};

void readData(serialization::ObjectDataInput& in){
    x = in.readFloat();
    y = in.readFloat();
};
private:
    float x;
    float y;
};

```

Now, you can use class `Foo` and `Bar` in distributed structures. For example as `Key` or `Value` of `IMap` or as an `Item` in `IQueue`.

10.1.3 C# Client

Enterprise Only

You can use native C# client to connect to Hazelcast nodes. All you need is to add `HazelcastClient3x.dll` into your C# project references. The API is very similar to Java native client. Sample code is shown below.

```

using Hazelcast.Config;
using Hazelcast.Client;
using Hazelcast.Core;
using Hazelcast.IO.Serialization;

using System.Collections.Generic;

namespace Hazelcast.Client.Example
{
    public class SimpleExample
    {
        public static void Test()
        {
            var clientConfig = new ClientConfig();
            clientConfig.GetNetworkConfig().AddAddress("10.0.0.1");
            clientConfig.GetNetworkConfig().AddAddress("10.0.0.2:5702");

            //Portable Serialization setup up for Customer Class
            clientConfig.GetSerializationConfig().AddPortableFactory(MyPortableFactory.FactoryId, new MyPortableFactory());

            IHazelcastInstance client = HazelcastClient.NewHazelcastClient(clientConfig);
            //All cluster operations that you can do with ordinary HazelcastInstance
            IMap<string, Customer> mapCustomers = client.GetMap<string, Customer>("customers");
            mapCustomers.Put("1", new Customer("Joe", "Smith"));
            mapCustomers.Put("2", new Customer("Ali", "Selam"));
            mapCustomers.Put("3", new Customer("Avi", "Noyan"));

            ICollection<Customer> customers = mapCustomers.Values();
            foreach (var customer in customers)

```

```

        {
            //process customer
        }
    }
}

public class MyPortableFactory : IPortableFactory
{
    public const int FactoryId = 1;

    public IPortable Create(int classId) {
        if (Customer.Id == classId)
            return new Customer();
        else return null;
    }
}

public class Customer: IPortable
{
    private string name;
    private string surname;

    public const int Id = 5;

    public Customer(string name, string surname)
    {
        this.name = name;
        this.surname = surname;
    }

    public Customer(){}

    public int GetFactoryId()
    {
        return MyPortableFactory.FactoryId;
    }

    public int GetClassId()
    {
        return Id;
    }

    public void WritePortable(IPortableWriter writer)
    {
        writer.WriteUTF("n", name);
        writer.WriteUTF("s", surname);
    }

    public void ReadPortable(IPortableReader reader)
    {
        name = reader.ReadUTF("n");
        surname = reader.ReadUTF("s");
    }
}
}

```

10.1.3.1 Client Configuration

Hazelcast C# client can be configured via API or XML. To start the client, a configuration can be passed or can be left empty to use default values.

Note: C# and Java clients are similar in terms of configuration. Therefore, you can refer to *Java Client* section for configuration aspects. For information on C# API documentation, please refer to the API document provided along with the Hazelcast Enterprise license.

10.1.3.2 Client Startup

After configuration, one can obtain a client using one of the static methods of Hazelcast like as shown below.

```
IHazelcastInstance client = HazelcastClient.NewHazelcastClient(clientConfig);
```

```
...
```

```
IHazelcastInstance defaultClient = HazelcastClient.NewHazelcastClient();
```

```
...
```

```
IHazelcastInstance xmlConfClient = Hazelcast.NewHazelcastClient(@"..\Hazelcast.Net\Resources\hazelcast-c
```

IHazelcastInstance interface is the starting point where all distributed objects can be obtained using it.

```
var map = client.GetMap<int,string>("mapName");
```

```
...
```

```
var lock= client.GetLock("thelock");
```

C# Client has following distributed objects:

- IMap<K,V>
- IMultiMap<K,V>
- IQueue<E>
- ITopic<E>
- IHList<E>
- IHSet<E>
- IIdGenerator
- ILock
- ISemaphore
- ICountDownLatch
- IAtomicLong
- ITransactionContext

ITransactionContext can be used to obtain;

- ITransactionalMap<K,V>
- ITransactionalMultiMap<K,V>
- ITransactionalList<E>
- ITransactionalSet<E>

10.2 REST Client

Hazelcast provides REST interface, i.e. it provides an HTTP service in each node so that your `map` and `queue` can be accessed using HTTP protocol. Assuming `mapName` and `queueName` are already configured in your Hazelcast, its structure is shown below:

```
http://node IP address:port/hazelcast/rest/maps/mapName/key
```

```
http://node IP address:port/hazelcast/rest/queues/queueName
```

For the operations to be performed, standard REST conventions for HTTP calls are used.

Assume that your cluster members are as shown below.

```
Members [5] {
  Member [10.20.17.1:5701]
  Member [10.20.17.2:5701]
  Member [10.20.17.4:5701]
  Member [10.20.17.3:5701]
  Member [10.20.17.5:5701]
}
```

Note: All of the requests below can return one of the following two in case of a failure

- If HTTP request syntax is not known, the following will be returned as response.

```
HTTP/1.1 400 Bad Request
Content-Length: 0
```

- In case of an unexpected exception, it will return:

```
< HTTP/1.1 500 Internal Server Error
< Content-Length: 0
```

Creating/Updating Entries in a Map

You can put a new `key1/value1` entry into a map by using POST call to `http://10.20.17.1:5701/hazelcast/rest/maps/mapName/key1` URL. This call's content body should contain the value of the key. Also, if the call contains the MIME type, Hazelcast stores this information, too.

A sample POST call is shown below.

```
$ curl -v -X POST -H "Content-Type: text/plain" -d "bar" http://10.20.17.1:5701/hazelcast/rest/maps/mapName/key1
```

It will return the following if successful:

```
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 0
```

Retrieving Entries from a Map

If you want to retrieve an entry, you can use GET call to `http://10.20.17.1:5701/hazelcast/rest/maps/mapName/key1`. You can also retrieve this entry from another member of your cluster such as `http://10.20.17.3:5701/hazelcast/rest/maps/mapName/key1`.

A sample GET call is shown below.

```
$ curl -X GET http://10.20.17.3:5701/hazelcast/rest/maps/mapName/foo
```

It will return the following if there is a corresponding value:

```
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 3
bar
```

As you can see, GET call returned value, its length and also the MIME type (**text/plain**) since POST call sample shown above included the MIME type.

It will return the following if there is no mapping for the given key:

```
< HTTP/1.1 204 No Content
< Content-Length: 0
```

Removing Entries from a Map

You can use DELETE call to remove an entry. A sample DELETE call is shown below with its returns.

```
$ curl -v -X DELETE http://10.20.17.1:5701/hazelcast/rest/maps/mapName/foo

< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 0
```

If you leave key empty as follows, it will delete all entries from map.

```
$ curl -v -X DELETE http://10.20.17.1:5701/hazelcast/rest/maps/mapName

< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 0
```

Offering Items on a Queue

You can use POST call to create an item on the queue. A sample is shown below.

```
$ curl -v -X POST -H "Content-Type: text/plain" -d "foo" http://10.20.17.1:5701/hazelcast/rest/queues/myQueue
```

Above call is equivalent to `HazelcastInstance#getQueue("myEvents").offer("foo");`.

It will return the following if successful:

```
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 0
```

It will return the following if queue is full and item is not be able to offered to queue:

```
< HTTP/1.1 503 Service Unavailable
< Content-Length: 0
```


Retrieving Items from a Queue

DELETE call can be used for retrieving. Note that, poll timeout should be stated while polling for queue events by an extra path parameter.

A sample is shown below (**10** being the timeout value).

```
$ curl -v -X DELETE \http://10.20.17.1:5701/hazelcast/rest/queues/myEvents/10
```

Above call is equivalent to `HazelcastInstance#getQueue("myEvents").poll(10, SECONDS);`. Below is the returns of above call.

```
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 3
foo
```

When the timeout is reached, the return will be `No Content` success, i.e. there is no item on the queue to be returned.

```
< HTTP/1.1 204 No Content
< Content-Length: 0
```

Getting the size of the queue

```
$ curl -v -X GET \http://10.20.17.1:5701/hazelcast/rest/queues/myEvents/size
```

Above call is equivalent to `HazelcastInstance#getQueue("myEvents").size();`. Below is a sample return of above call.

```
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 1
5
```

RESTful access is provided through any member of your cluster. So you can even put an HTTP load-balancer in front of your cluster members for load balancing and fault tolerance.

Note: *You need to handle the failures on REST polls as there is no transactional guarantee.*

10.3 Memcache Client

A Memcache client written in any language can talk directly to Hazelcast cluster. No additional configuration is required. Assume that your cluster members are as below.

```
Members [5] {
  Member [10.20.17.1:5701]
  Member [10.20.17.2:5701]
  Member [10.20.17.4:5701]
  Member [10.20.17.3:5701]
  Member [10.20.17.5:5701]
}
```

And you have a PHP application that uses PHP Memcache client to cache things in Hazelcast. All you need to do is have your PHP Memcache client connect to one of these members. It does not matter which member the client connects to because Hazelcast cluster looks as one giant machine (Single System Image). PHP client code sample:

```
<?php
    $memcache = new Memcache;
    $memcache->connect('10.20.17.1', 5701) or die ("Could not connect");
    $memcache->set('key1','value1',0,3600);
    $get_result = $memcache->get('key1'); //retrieve your data
    var_dump($get_result); //show it
?>
```

Notice that Memcache client is connecting to 10.20.17.1 and using port 5701. Java client code sample with SpyMemcached client:

```
MemcachedClient client = new MemcachedClient(AddrUtil.getAddresses("10.20.17.1:5701 10.20.17.2:5701"));
client.set("key1", 3600, "value1");
System.out.println(client.get("key1"));
```

If you want your data to be stored in different maps (e.g. to utilize per map configuration), you can do that with a map name prefix as following:

```
MemcachedClient client = new MemcachedClient(AddrUtil.getAddresses("10.20.17.1:5701 10.20.17.2:5701"));
client.set("map1:key1", 3600, "value1"); // store to *hz_memcache_map1
client.set("map2:key1", 3600, "value1"); // store to hz_memcache_map2
System.out.println(client.get("key1")); //get from hz_memcache_map1
System.out.println(client.get("key2")); //get from hz_memcache_map2
```

hz_memcache prefix is to separate Memcache maps from Hazelcast maps. If no map name is given, it will be stored in default map named as *hz_memcache_default*.

An entry written with a Memcache client can be read by another Memcache client written in another language.

10.3.1 Unsupported Operations

- CAS operations are not supported. In operations getting CAS parameters like append, CAS values are ignored.
- Only a subset of statistics are supported. Below is the list of supported statistic values.

```
- cmd_set
- cmd_get
- incr_hits
- incr_misses
- decr_hits
- decr_misses
```

Chapter 11

Serialization

All your distributed objects such as your key and value objects, objects you offer into distributed queue and your distributed callable/runnable objects have to be **Serializable**.

Hazelcast serializes all your objects into an instance of `com.hazelcast.nio.serialization.Data`. Data is the binary representation of an object.

When Hazelcast serializes an object into Data, it first checks whether the object is an instance of `com.hazelcast.nio.serialization.DataSerializable`, if not it checks if it is an instance of `com.hazelcast.nio.serialization.Serializable` and serializes it accordingly.

Hazelcast optimizes the serialization for the below types, and the user cannot override this behavior:

- Byte
- Boolean
- Character
- Short
- Integer
- Long
- Float
- Double
- byte[]
- char[]
- short[]
- int[]
- long[]
- float[]
- double[]
- String

Hazelcast also optimizes the following types. However, you can override them by creating a custom serializer and registering it. See [Custom Serialization](#) for more information.

- Date
- BigInteger
- BigDecimal
- Class
- Externalizable
- Serializable

Note that, if the object is not an instance of any explicit type, Hazelcast uses Java Serialization for Serializable and Externalizable objects. The default behavior can be changed using a [Custom Serialization](#).

11.1 Data Serialization

For a faster serialization of objects, Hazelcast recommends to implement `com.hazelcast.nio.serialization.IdentifiedData` which is slightly better version of `com.hazelcast.nio.serialization.DataSerializable`.

Here is an example of a class implementing `com.hazelcast.nio.serialization.DataSerializable` interface.

```
public class Address implements com.hazelcast.nio.serialization.DataSerializable {
    private String street;
    private int zipCode;
    private String city;
    private String state;

    public Address() {}

    //getters setters..

    public void writeData(ObjectDataOutput out) throws IOException {
        out.writeUTF(street);
        out.writeInt(zipCode);
        out.writeUTF(city);
        out.writeUTF(state);
    }

    public void readData(ObjectDataInput in) throws IOException {
        street    = in.readUTF();
        zipCode   = in.readInt();
        city      = in.readUTF();
        state     = in.readUTF();
    }
}
```

Let's take a look at another example which is encapsulating a `DataSerializable` field.

```
public class Employee implements com.hazelcast.nio.serialization.DataSerializable {
    private String firstName;
    private String lastName;
    private int age;
    private double salary;
    private Address address; //address itself is DataSerializable

    public Employee() {}

    //getters setters..

    public void writeData(ObjectDataOutput out) throws IOException {
        out.writeUTF(firstName);
        out.writeUTF(lastName);
        out.writeInt(age);
        out.writeDouble(salary);
        address.writeData(out);
    }

    public void readData (ObjectDataInput in) throws IOException {
        firstName = in.readUTF();
        lastName  = in.readUTF();
        age       = in.readInt();
        salary     = in.readDouble();
    }
}
```

```

        address    = new Address();
        // since Address is DataSerializable let it read its own internal state
        address.readData (in);
    }
}

```

As you can see, since `address` field itself is `DataSerializable`, it is calling `address.writeData(out)` when writing and `address.readData(in)` when reading. Also note that, the order of writing and reading fields should be the same. While Hazelcast serializes a `DataSerializable`, it writes the `className` first and when de-serializes it, `className` is used to instantiate the object using reflection.

11.1.1 IdentifiedDataSerializable

To avoid the reflection and long class names, `IdentifiedDataSerializable` can be used instead of `DataSerializable`. Note that, `IdentifiedDataSerializable` extends `DataSerializable` and introduces two new methods.

- `int getId();`
- `int getFactoryId();`

`IdentifiedDataSerializable` uses `getId()` instead of class name and uses `getFactoryId()` to load the class given the Id. To complete the implementation, a `com.hazelcast.nio.serialization.DataSerializableFactory` should also be implemented and registered into `SerializationConfig` which can be accessed from `Config.getSerializationConfig()`. Factory's responsibility is to return an instance of the right `IdentifiedDataSerializable` object, given the Id. So far this is the most efficient way of Serialization that Hazelcast supports off the shelf.

11.2 Portable Serialization

As an alternative to the existing serialization methods, Hazelcast offers a Portable serialization that have the following advantages:

- Support multiversion of the same object type.
- Fetching individual fields without having to rely on reflection.
- Querying and indexing support without de-serialization and/or reflection.

In order to support these features, a serialized Portable object is offered containing meta information like the version and the concrete location of the each field in the binary data. This way Hazelcast is able to navigate in the byte[] and de-serialize only the required field without actually de-serializing the whole object which improves the Query performance.

With multiversion support, you can have two nodes where each of them having different versions of the same object and Hazelcast will store both meta information and use the correct one to serialize and de-serialize Portable objects depending on the node. This is very helpful when you are doing a rolling upgrade without shutting down the cluster.

Also note that, Portable serialization is totally language independent and is used as the binary protocol between Hazelcast server and clients.

A sample Portable implementation of a Foo class would look like the following.

```

public class Foo implements Portable{
    final static int ID = 5;

    private String foo;
}

```

```

    public String getFoo() {
        return foo;
    }

    public void setFoo(String foo) {
        this.foo = foo;
    }

    @Override
    public int getFactoryId() {
        return 1;
    }

    @Override
    public int getClassId() {
        return ID;
    }

    @Override
    public void writePortable(PortableWriter writer) throws IOException {
        writer.writeUTF("foo", foo);
    }

    @Override
    public void readPortable(PortableReader reader) throws IOException {
        foo = reader.readUTF("foo");
    }
}

```

Similar to `IdentifiedDataSerializable`, a `Portable` Class must provide `classId` and `factoryId`. The `Factory` object will be used to create the `Portable` object given the `classId`.

A sample `Factory` could be implemented as following:

```

public class MyPortableFactory implements PortableFactory {

    @Override
    public Portable create(int classId) {
        if (Foo.ID == classId)
            return new Foo();
        else return null;
    }
}

```

The last step is to register the `Factory` to the `SerializationConfig`. Below are the programmatic and declarative configurations for this step in order.

```

Config config = new Config();
config.getSerializationConfig().addPortableFactory(1, new MyPortableFactory());

```

```

<hazelcast>
  <serialization>
    <portable-version>0</portable-version>
    <portable-factories>
      <portable-factory factory-id="1">com.hazelcast.nio.serialization.MyPortableFactory</portable-factory>
    </portable-factories>
  </serialization>
</hazelcast>

```

Note that the id that is passed to the `SerializationConfig` is same as the `factoryId` that `Foo` class returns.

11.3 Custom Serialization

Hazelcast lets you plug a custom serializer to be used for serialization of objects.

Assume that you have a class `Foo` and you would like to customize the serialization. The reasons could be `Foo` is not `Serializable` or you are not happy with the default serialization.

```
public class Foo {
    private String foo;
    public String getFoo() {
        return foo;
    }
    public void setFoo(String foo) {
        this.foo = foo;
    }
}
```

Assume that our custom serialization will serialize `Foo` into XML. First we need to implement a `com.hazelcast.nio.serialization.StreamSerializer`. A very simple one that uses `XMLEncoder` and `XMLDecoder`, would look like the following:

```
public static class FooXmlSerializer implements StreamSerializer<Foo> {

    @Override
    public int getTypeId() {
        return 10;
    }

    @Override
    public void write(ObjectDataOutput out, Foo object) throws IOException {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        XMLEncoder encoder = new XMLEncoder(bos);
        encoder.writeObject(object);
        encoder.close();
        out.write(bos.toByteArray());
    }

    @Override
    public Foo read(ObjectDataInput in) throws IOException {
        final InputStream inputStream = (InputStream) in;
        XMLDecoder decoder = new XMLDecoder(inputStream);
        return (Foo) decoder.readObject();
    }

    @Override
    public void destroy() {
    }
}
```

Note that `typeId` must be unique as Hazelcast will use it to lookup the `StreamSerializer` while it de-serializes the object. Now, the last required step is to register the `StreamSerializer` to the Configuration. Below are the programmatic and declarative configurations for this step in order.

```
SerializerConfig sc = new SerializerConfig().
    setImplementation(new FooXmlSerializer()).
    setTypeClass(Foo.class);
Config config = new Config();
config.getSerializationConfig().addSerializerConfig(sc);
```

```
<hazelcast>
  <serialization>
    <serializers>
      <serializer type-class="com.www.Foo">com.www.FooXmlSerializer</serializer>
    </serializers>
  </serialization>
</hazelcast>
```

From now on, Hazelcast will use `FooXmlSerializer` to serialize `Foo` objects. This way one can write an adapter (`StreamSerializer`) for any Serialization framework and plug it into Hazelcast.

Chapter 12

Management

12.1 Monitoring with JMX

- Add the following system properties to enable [JMX agent](#):
 - `-Dcom.sun.management.jmxremote`
 - `-Dcom.sun.management.jmxremote.port=_portNo_` (to specify JMX port) (*optional*)
 - `-Dcom.sun.management.jmxremote.authenticate=false` (to disable JMX auth) (*optional*)
- Enable Hazelcast property `hazelcast.jmx` (please refer to [Advanced Configuration Properties](#)):
 - using Hazelcast configuration (API, XML, Spring)
 - or set system property `-Dhazelcast.jmx=true`
- Use `jconsole`, `jvisualvm` (with `mbean` plugin) or another JMX compliant monitoring tool.

Following attributes can be monitored:

- Cluster
 - configuration
 - group name
 - count of members and their addresses (*host:port*)
 - operations: cluster restart, shutdown
- Member
 - inet address
 - port
- Statistics

- count of instances
- number of instances created/destroyed since startup
- maximum instances created/destroyed per second
- AtomicLong
 - name
 - actual value
 - operations: add, set, compareAndSet, reset
- List, Set
 - name
 - size
 - items (as strings)
 - operations: clear, reset statistics
- Map
 - name
 - size
 - operations: clear
- Queue
 - name
 - size
 - received and served items
 - operations: clear, reset statistics
- Topic
 - name
 - number of messages dispatched since creation, in last second
 - maximum messages dispatched per second

12.2 Cluster Utilities

12.2.1 Cluster Interface

Hazelcast allows you to register for membership events to get notified when members added or removed. You can also get the set of cluster members.

```

import com.hazelcast.core.*;
import com.hazelcast.config.Config;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
Cluster cluster = hz.getCluster();
cluster.addMembershipListener(new MembershipListener(){
    public void memberAdded(MembershipEvent membershipEvent) {
        System.out.println("MemberAdded " + membershipEvent);
    }

    public void memberRemoved(MembershipEvent membershipEvent) {
        System.out.println("MemberRemoved " + membershipEvent);
    }
});

Member localMember = cluster.getLocalMember();
System.out.println ("my inetAddress= " + localMember.getInetAddress());

Set setMembers = cluster.getMembers();
for (Member member : setMembers) {
    System.out.println ("isLocalMember " + member.isLocalMember());
    System.out.println ("member.inetAddress " + member.getInetAddress());
    System.out.println ("member.port " + member.getPort());
}

```

12.2.2 Cluster Wide ID Generator

Hazelcast `IdGenerator` creates cluster wide unique IDs. Generated IDs are long type primitive values between 0 and `Long.MAX_VALUE`. ID generation occurs almost at the speed of `AtomicLong.incrementAndGet()`. Generated IDs are unique during the life cycle of the cluster. If the entire cluster is restarted, IDs start from 0 again or you can initialize to a value.

```

import com.hazelcast.core.IdGenerator;
import com.hazelcast.core.Hazelcast;

Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
IdGenerator idGenerator = hz.getIdGenerator("customer-ids");
idGenerator.init(123L); //Optional
long id = idGenerator.newId();

```

12.3 Management Center

12.3.1 Introduction

Hazelcast Management Center enables you to monitor and manage your nodes running Hazelcast. In addition to monitoring overall state of your clusters, you can also analyze and browse your data structures in detail, update map configurations and take thread dump from nodes. With its scripting and console module, you can run scripts (JavaScript, Groovy, etc.) and commands on your nodes.

12.3.1.1 Installation

Basically you will deploy `mancenter-version.war` application into your Java web server and then tell Hazelcast nodes to talk to that web application. That means, your Hazelcast nodes should know the URL of `mancenter` application before they start.

Here are the steps:

- Download the latest Hazelcast ZIP from hazelcast.org.
- ZIP contains `mancenter-version.war` file. Deploy it to your web server (Tomcat, Jetty, etc.). Let us say it is running at `http://localhost:8080/mancenter`.
- Start your web server and make sure `http://localhost:8080/mancenter` is up.
- Configure your Hazelcast nodes by adding the URL of your web app to your `hazelcast.xml`. Hazelcast nodes will send their states to this URL.

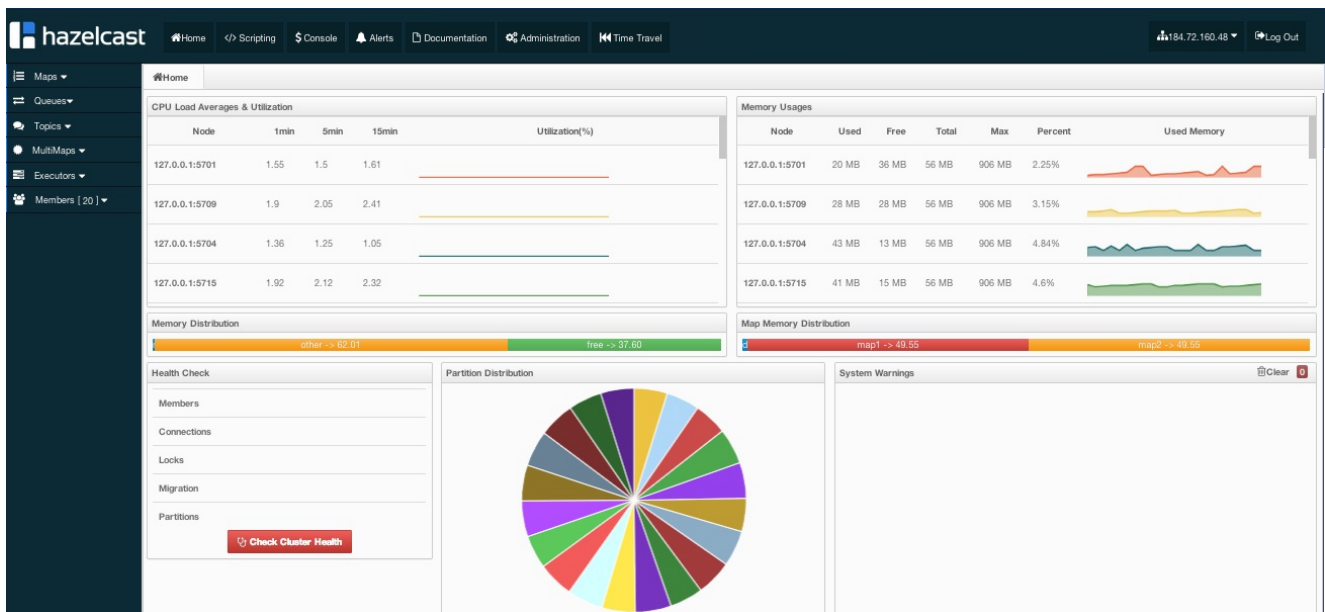
```
<management-center enabled="true">http://localhost:8080/mancenter</management-center>
```

- Start your Hazelcast cluster.
- Browse to `http://localhost:8080/mancenter` and login. Initial login username/password is `admin/admin`

Management Center creates a directory with name “mancenter” under your “user/home” directory to save data files. You can change the data directory by setting `hazelcast.mancenter.home` system property.

12.3.2 Tool Overview

Once the page is loaded after selecting a cluster, tool’s home page appears as shown below.



This page provides the fundamental properties of the selected cluster which are explained in Home Page section. It also has a toolbar on the top and a menu on the left.

12.3.2.1 Toolbar

Toolbar has the following buttons:

- **Home:** When pressed, loads the home page shown above. Please see Home Page.
- **Scripting:** When pressed, loads the page used to write and execute user’s own scripts on the cluster. Please see [Scripting](#).

- **Console:** When pressed, loads the page used to execute commands on the cluster. Please see [Console](#).
- **Alerts:** It is used to create alerts by specifying filters. Please see [Alerts](#).
- **Documentation:** It is used to open the documentation of Management Center in a window inside the tool. Please see [Documentation](#).
- **Administration:** It is used by the admin users to manage users in the system. Please see [Administration](#).
- **Time Travel:** It is used to see the cluster's situation at a time in the past. Please see [Time Travel](#).
- **Cluster Selector:** It is used to switch between clusters. When the mouse is moved onto this item, a dropdown list of clusters appears.

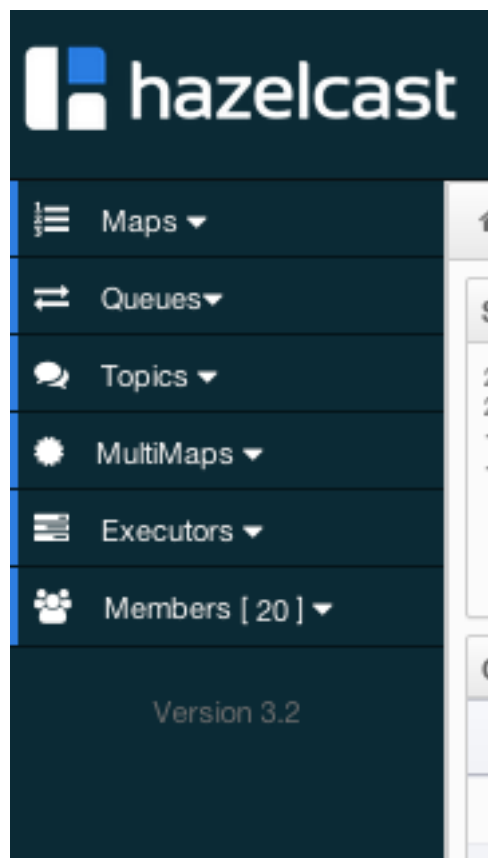
The user can select any cluster and once selected, the page immediately loads with the selected cluster.

- **Logout:** It is used to close the current user's session.

Note: Not all of the above listed toolbar items are visible to the users who are not admin or have **read-only** permission. Also, some of the operations explained in the later sections cannot be performed by users with read-only permission. Please see [Administration](#) for details.

12.3.2.2 Menu

Home page includes a menu on the left which lists the distributed data structures in the cluster and also all cluster members (nodes), as shown below.

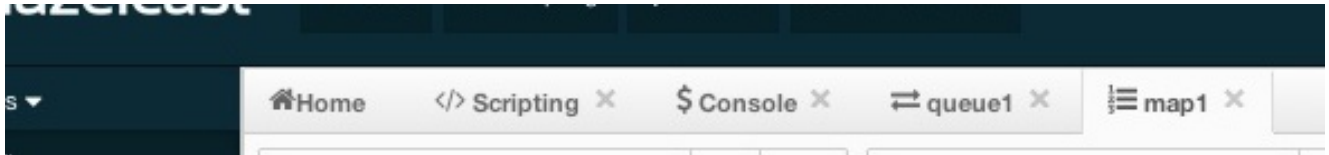



Menu items can be expanded/collapsed by clicking on them. Below is the list of menu items with the links to their explanations.

- [Maps](#)
- [Queues](#)
- [Topics](#)
- MultiMaps
- [Executors](#)
- [Members](#)

12.3.2.3 Tabbed View

Each time an item from the toolbar or menu is selected, it is added to main view as a tab, as shown below.



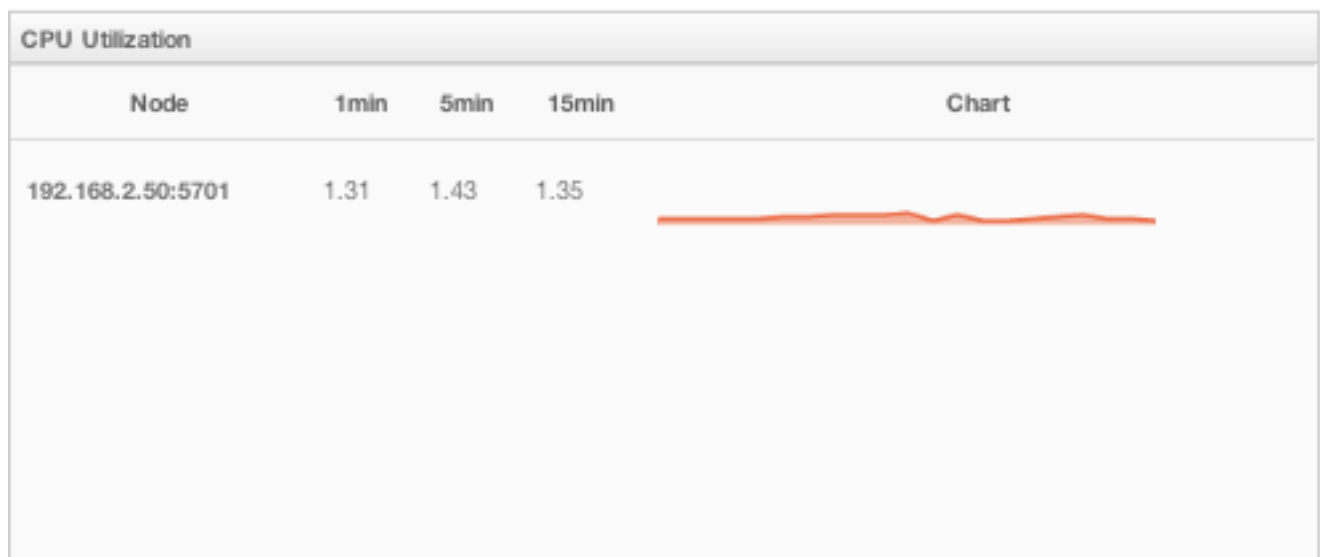
In the above example, *Home*, *Scripting*, *Console*, *queue1* and *map1* windows can be seen as tabs. Windows can be closed using the  icon on each tab (except the Home Page; it cannot be closed).

12.3.3 Home Page

This is the first page appearing after logging in. It gives an overview of the cluster connected. Below subsections describe each portion of the page.

12.3.3.1 CPU Utilization

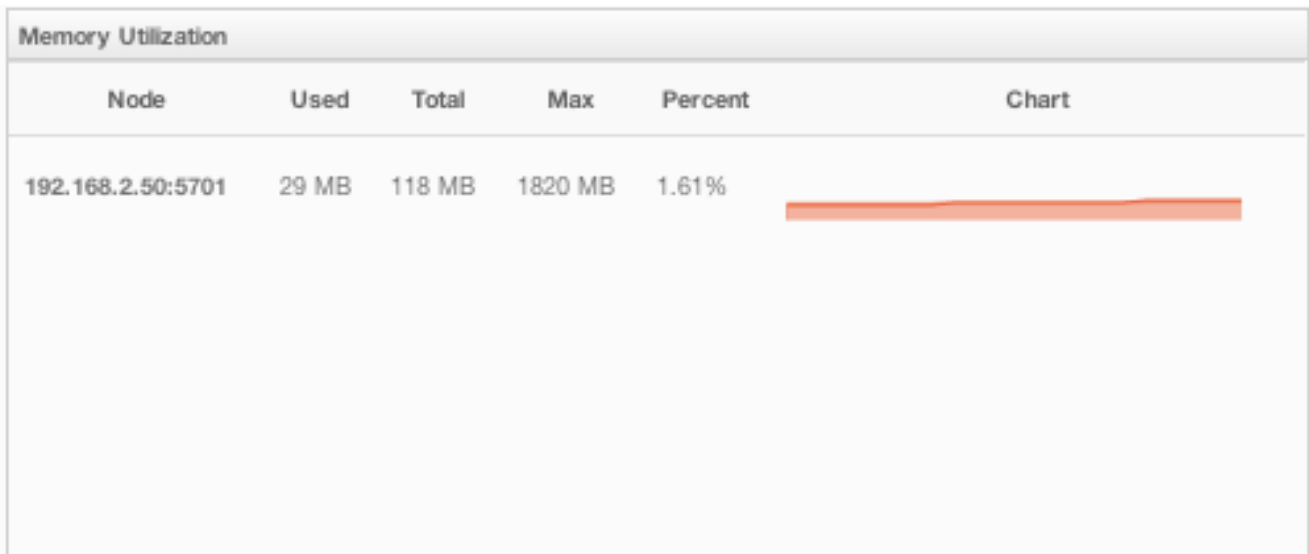
This part of the page provides information related to load and utilization of CPUs for each node, as shown below.



First column lists the nodes with their IPs and ports. Then, the loads on each CPU for the last 1, 5 and 15 minutes are listed. The last column (**Chart**) shows the utilization of CPUs graphically. When you move the mouse cursor on a desired graph, you can see the CPU utilization at the time to which cursor corresponds. Graphs under this column shows the CPU utilizations approximately for the last 2 minutes.

12.3.3.2 Memory Utilization

This part of the page provides information related to memory usages for each node, as shown below.



First column lists the nodes with their IPs and ports. Then, used and free memories out of the total memory reserved for Hazelcast usage are shown, in real-time. **Max** column lists the maximum memory capacity of each node and **Percent** column lists the percentage value of used memory out of the maximum memory. The last column (**Chart**) shows the memory usage of nodes graphically. When you move the mouse cursor on a desired graph, you can see the memory usage at the time to which cursor corresponds. Graphs under this column shows the memory usages approximately for the last 2 minutes.

12.3.3.3 Memory Distribution

This part of the page graphically provides the cluster wise breakdown of memory, as shown below. Blue area is the memory used by maps, dark yellow area is the memory used by non-Hazelcast entities and green area is the free memory (out of whole cluster's memory capacity).



In the above example, you can see 0.32% of the total memory is used by Hazelcast maps (it can be seen by moving the mouse cursor on it), 58.75% is used by non-Hazelcast entities and 40.85% of the total memory is free.

12.3.3.4 Map Memory Distribution

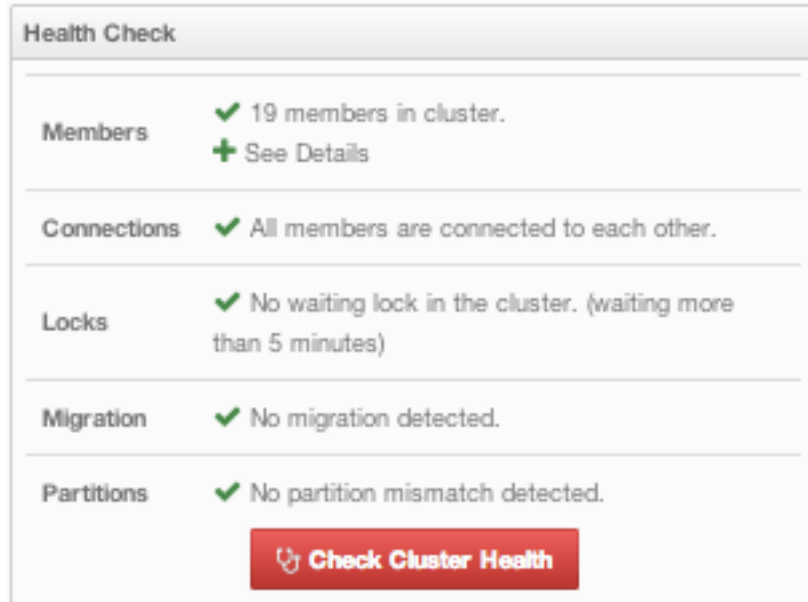
This part is actually the breakdown of the blue area shown in **Memory Distribution** graph explained above. It provides the percentage values of the memories used by each map, out of the total cluster memory reserved for all Hazelcast maps.



In the above example, you can see 49.55% of the total map memory is used by **map1** and 49.55% is used by **map2**.

12.3.3.5 Health Check

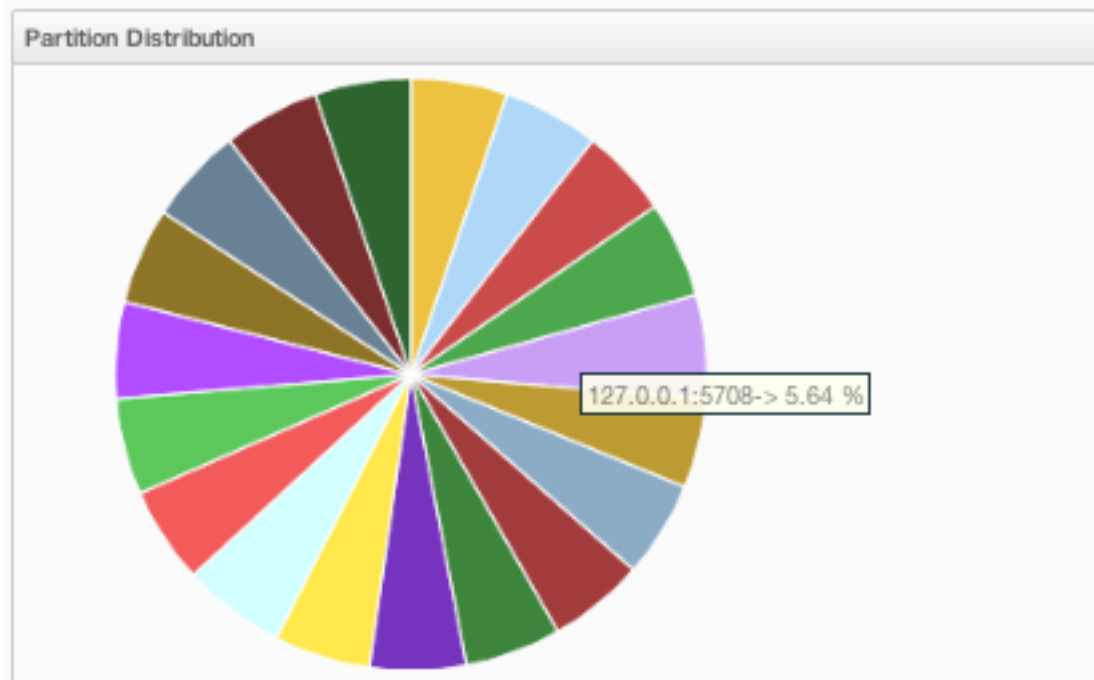
This part is useful to check how the cluster in general behaves. It lists the nodes (cluster members), locks and partition mismatches along with the information related to migrations and node interconnections. To see these, just click on **Check Cluster Health** button. A sample is shown below.



You can see each node's IP address and port by clicking on the plus sign at the **Members**.

12.3.3.6 Partition Distribution

This pie chart shows what percentage of partitions each node has, as shown below.



You can see each node's partition percentages by moving the mouse cursor on the chart. In the above example, you can see the node "127.0.0.1:5708" has 5.64% of the total partition count (which is 271 by default and configurable, please see [Advanced Configuration Properties](#)).

12.3.3.7 System Warnings

This part of the page shows informative warnings in situations like shutting down a node, as shown below.

System Warnings		Clear 26
Node 127.0.0.1:5701 is shutdown Node 127.0.0.1:5701 is shutdown...		2 hours ago
Node 127.0.0.1:5701 is shutdown Node 127.0.0.1:5701 is shutdown...		2 hours ago
Node 127.0.0.1:5701 is shutdown Node 127.0.0.1:5701 is shutdown...		2 hours ago
Node 127.0.0.1:5701 is shutdown Node 127.0.0.1:5701 is shutdown...		2 hours ago
Node 127.0.0.1:5701 is shutdown Node 127.0.0.1:5701 is shutdown...		2 hours ago
Node 127.0.0.1:5701 is shutdown Node 127.0.0.1:5701 is shutdown...		2 hours ago

Warnings can be cleared by clicking on the **Clear** link placed at top right of the window.

12.3.4 Maps

Map instances are listed under the **Maps** menu item on the left. When you click on a map, a new tab for monitoring that map instance is opened on the right, as shown below. In this tab, you can monitor metrics and also re-configure the selected map.

Below subsections explain the portions of this window.

12.3.4.1 Map Browser

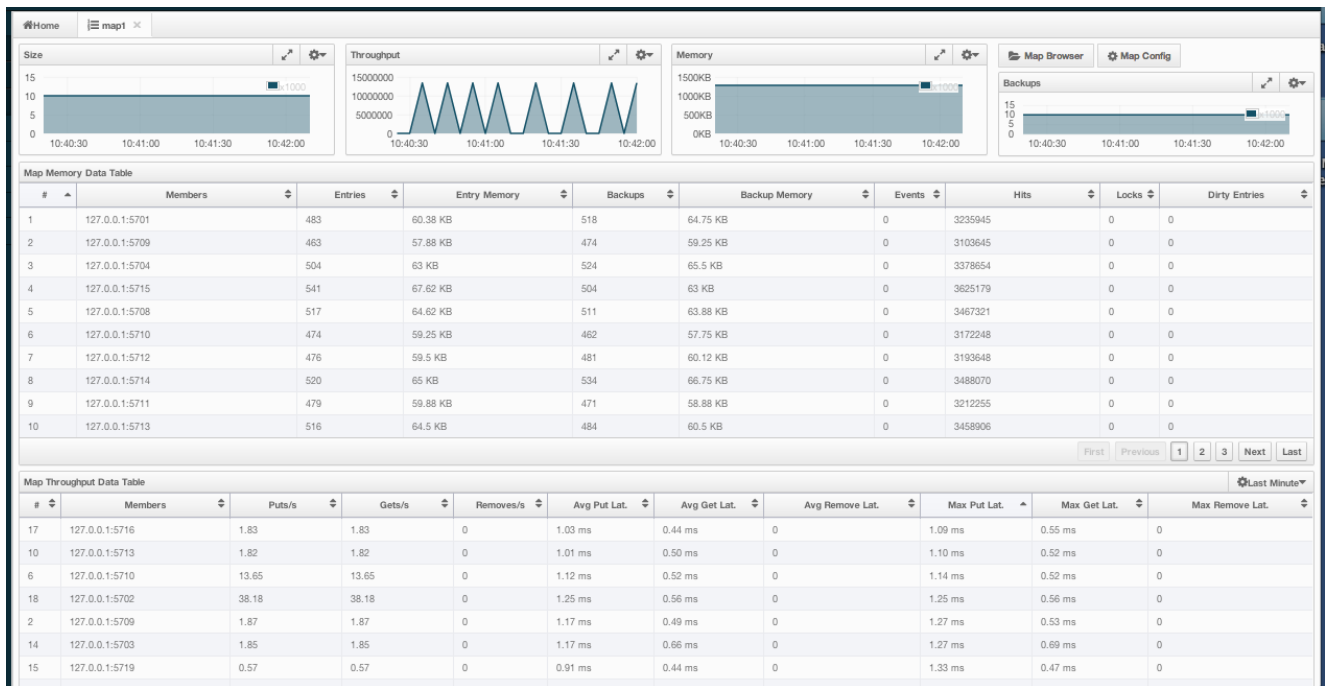
Map Browser is a tool used to retrieve properties of the entries stored in the selected map. It can be opened by clicking on the **Map Browser** button, located at top right of the window. Once opened, the tool appears as a dialog, as shown below.

Once the key and key's type is specified and **Browse** button is clicked, key's properties along with its value is listed.

12.3.4.2 Map Config

By using Map Config tool, you can set selected map's attributes like the backup count, TTL, and eviction policy. It can be opened by clicking on the **Map Config** button, located at top right of the window. Once opened, the tool appears as a dialog, as shown below.

Change any attribute as required and click **Update** button to save changes.



Map Browser

2 Integer

Value: 2 **Class:** java.lang.Integer

Cost: 0.12 KB **Creation Time:** Fri Feb 21 15:17:58 UTC 2014

Expiration Time: Thu Jan 01 00:00:00 UTC 1970 **Hits:** 6689

Access Time: Mon Mar 03 09:07:51 UTC 2014 **Update Time:** Mon Mar 03 09:07:51 UTC 2014

Version: 3335 **Valid:**

Map Config

Name: default

Max Size: 2147483647

Backup Count: 1

Async Backup Count: 0

Max Idle(seconds): 0

TTL (seconds): 0

Eviction Policy: None


Eviction Percentage (%): 25

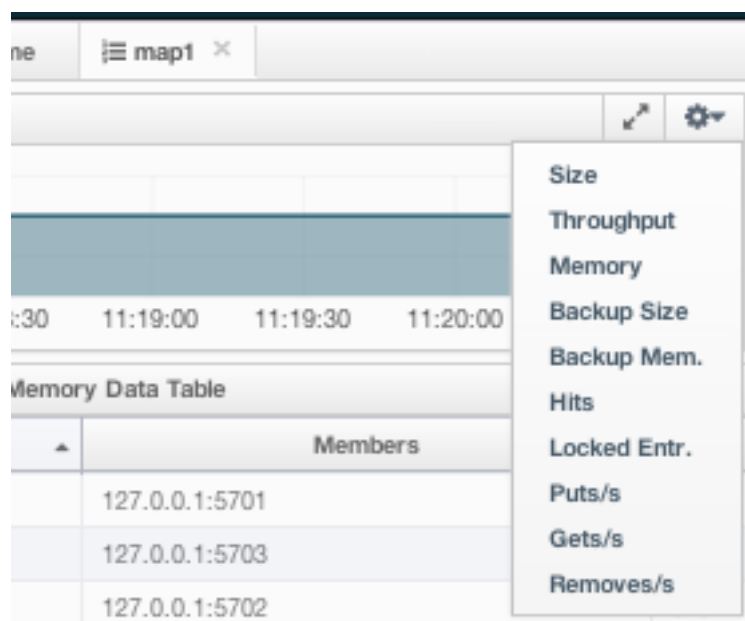
Read Backup Data: False


Update

12.3.4.3 Map Monitoring

Besides Map Browser and Map Config tools, this page has many monitoring options explained below. All of these perform real-time monitoring.

On top of the page, there are small charts to monitor the size, throughput, memory usage, backup size, etc. of the selected map in real-time. All charts' X-axis shows the current system time. Other small monitoring charts can be selected using  button placed at top right of each chart. When it is clicked, the whole list of monitoring options are listed, as shown below.



When you click on a desired monitoring, the chart is loaded with the selected option. Also, a chart can be opened as a separate dialog by clicking on the  button placed at top right of each chart. Below monitoring charts are available:

- **Size:** Monitors the size of the map. Y-axis is the entry count (should be multiplied by 1000).
- **Throughput:** Monitors get, put and remove operations performed on the map. Y-axis is the operation count.
- **Memory:** Monitors the memory usage on the map. Y-axis is the memory count.
- **Backups:** It is the chart loaded when “Backup Size” is selected. Monitors the size of the backups in the map. Y-axis is the backup entry count (should be multiplied by 1000).
- **Backup Memory:** It is the chart loaded when “Backup Mem.” is selected. Monitors the memory usage of the backups. Y-axis is the memory count.
- **Hits:** Monitors the hit count of the map.
- **Puts/s, Gets/s, Removes/s:** These three charts monitor the put, get and remove operations (per second) performed on the selected map.

Under these charts, there are **Map Memory** and **Map Throughput** data tables. Map Memory data table provides memory metrics distributed over nodes, as shown below.

# ▲	Members ▼	Entries ▼	Entry Memory ▼	Backups ▼	Backup Memory ▼	Event📈	Hits ▼	Lock🔒	Dirty Entries ▼
1	127.0.0.1:5701	515	64.38 KB	519	64.88 KB	0	73765	0	0
2	127.0.0.1:5703	498	62.25 KB	488	61 KB	0	71604	0	0
3	127.0.0.1:5702	525	65.62 KB	539	67.38 KB	0	75729	0	0
4	127.0.0.1:5708	542	67.75 KB	540	67.5 KB	0	77484	0	0
5	127.0.0.1:5707	489	61.12 KB	459	57.38 KB	0	70175	0	0
6	127.0.0.1:5706	494	61.75 KB	490	61.25 KB	0	71020	0	0
7	127.0.0.1:5709	486	60.75 KB	496	62 KB	0	70392	0	0
8	127.0.0.1:5704	516	64.5 KB	501	62.62 KB	0	74064	0	0
9	127.0.0.1:5713	511	63.88 KB	497	62.12 KB	0	73329	0	0
10	127.0.0.1:5716	468	58.5 KB	493	61.62 KB	0	67414	0	0

First
Previous
1
2
3
Next
Last

From left to right, this table lists the IP address and port, entry counts, memory used by entries, backup entry counts, memory used by backup entries, events, hits, locks and dirty entries (in the cases where *MapStore* is enabled, these are the entries that are put to/removed from the map but not written to/removed from a database yet) of each node in the map. You can navigate through the pages using the buttons placed at the bottom right of the table (**First**, **Previous**, **Next**, **Last**). The order of the listings in each column can be ascended or descended by clicking on column headings.

Map Throughput data table provides information about the operations (get, put, remove) performed on each node in the map, as shown below.

# ↕	Members ▼	Puts/s📈	Gets/s📈	Removes/s📈	Avg Put Lat. ▼	Avg Get Lat. ↕	Avg Remove Lat. ▼	Max Put Lat. ↕	Max Get Lat. ↕	Max Remove Lat. ▼
8	127.0.0.1:5704	2.30	2.30	0	2.03 ms	0.69 ms	0	2.10 ms	0.85 ms	0
17	127.0.0.1:5714	2.30	2.30	0	2.01 ms	0.62 ms	0	3.49 ms	1.36 ms	0
7	127.0.0.1:5709	2.30	2.30	0	1.99 ms	0.66 ms	0	2.33 ms	0.82 ms	0
9	127.0.0.1:5713	2.27	2.27	0	1.97 ms	0.61 ms	0	2.01 ms	0.64 ms	0
13	127.0.0.1:5711	2.30	2.30	0	1.90 ms	0.65 ms	0	2.47 ms	0.93 ms	0
1	127.0.0.1:5701	2.27	2.27	0	1.87 ms	0.86 ms	0	2.24 ms	1.20 ms	0
18	127.0.0.1:5718	2.28	2.28	0	1.84 ms	0.60 ms	0	3.24 ms	0.67 ms	0
20	127.0.0.1:5720	2.30	2.30	0	1.80 ms	0.62 ms	0	1.88 ms	0.66 ms	0
5	127.0.0.1:5707	2.27	2.27	0	1.79 ms	0.63 ms	0	2.48 ms	0.79 ms	0
6	127.0.0.1:5706	2.30	2.30	0	1.78 ms	0.62 ms	0	3.91 ms	1.00 ms	0

First
Previous
1
2
Next
Last

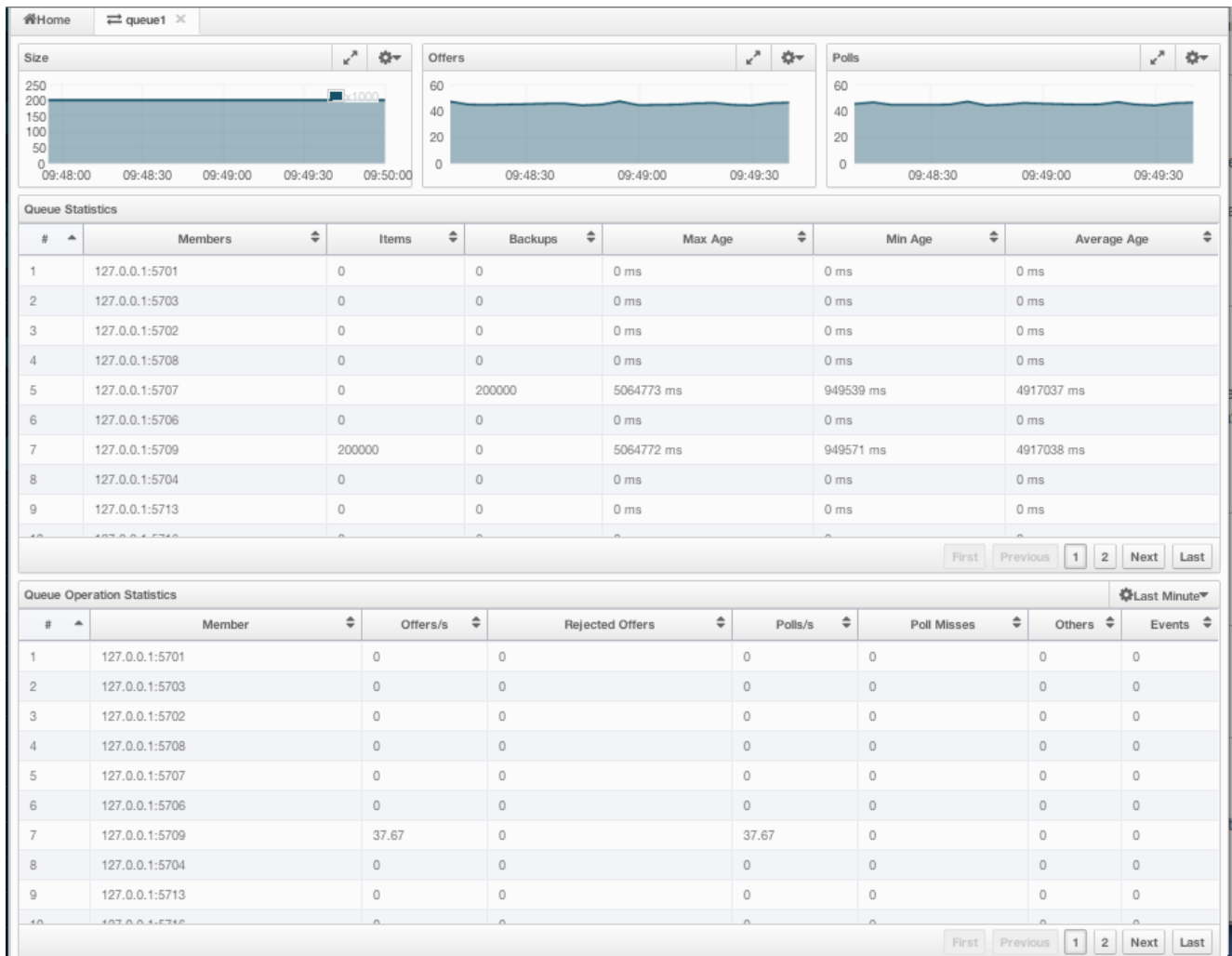
From left to right, this table lists the IP address and port of each node, put, get and remove operations on each node, average put, get, remove latencies and maximum put, get, remove latencies on each node.


You can select the period in the combo box placed at top right corner of the window, for which the table data will be shown. Available values are **Since Beginning**, **Last Minute**, **Last 10 Minutes** and **Last 1 Hour**.

You can navigate through the pages using the buttons placed at the bottom right of the table (**First**, **Previous**, **Next**, **Last**). The order of the listings in each column can be ascended or descended by clicking on column headings.

12.3.5 Queues

Using the menu item **Queues**, you can monitor your queues data structure. When you expand this menu item and click on a queue, a new tab for monitoring that queue instance is opened on the right, as shown below.



On top of the page, there are small charts to monitor the size, offers and polls of the selected queue in real-time. All charts' X-axis shows the current system time. And a chart can be opened as a separate dialog by clicking on the  button placed at top right of each chart. Below monitoring charts are available:

- **Size:** Monitors the size of the queue. Y-axis is the entry count (should be multiplied by 1000).
- **Offers:** Monitors the offers sent to the selected queue. Y-axis is the offer count.
- **Polls:** Monitors the polls sent to the selected queue. Y-axis is the poll count.

Under these charts, there are **Queue Statistics** and **Queue Operation Statistics** tables. Queue Statistics table provides item and backup item counts in the queue and age statistics of items and backup items at each node, as shown below.

From left to right, this table lists the IP address and port, items and backup items on the queue of each node, and maximum, minimum and average age of items in the queue. You can navigate through the pages using the buttons

Queue Statistics						
# ▲	Members ⇅	Items ⇅	Backups ⇅	Max Age ⇅	Min Age ⇅	Average Age ⇅
1	127.0.0.1:5701	0	0	0 ms	0 ms	0 ms
2	127.0.0.1:5703	0	0	0 ms	0 ms	0 ms
3	127.0.0.1:5702	0	0	0 ms	0 ms	0 ms
4	127.0.0.1:5708	0	0	0 ms	0 ms	0 ms
5	127.0.0.1:5707	0	200000	5064773 ms	949539 ms	4917037 ms
6	127.0.0.1:5706	0	0	0 ms	0 ms	0 ms
7	127.0.0.1:5709	200000	0	5064772 ms	949571 ms	4917038 ms
8	127.0.0.1:5704	0	0	0 ms	0 ms	0 ms
9	127.0.0.1:5713	0	0	0 ms	0 ms	0 ms
10	127.0.0.1:5716	0	0	0 ms	0 ms	0 ms

First Previous 1 2 Next Last

placed at the bottom right of the table (**First, Previous, Next, Last**). The order of the listings in each column can be ascended or descended by clicking on column headings.

Queue Operations Statistics table provides information about the operations (offers, polls, events) performed on the queues, as shown below.

Queue Operation Statistics								⚙ Last Minute ▼
# ▲	Member ⇅	Offers/s ⇅	Rejected Offers ⇅	Polls/s ⇅	Poll Misses ⇅	Others ⇅	Events ⇅	
1	127.0.0.1:5701	0	0	0	0	0	0	
2	127.0.0.1:5703	0	0	0	0	0	0	
3	127.0.0.1:5702	0	0	0	0	0	0	
4	127.0.0.1:5708	0	0	0	0	0	0	
5	127.0.0.1:5707	0	0	0	0	0	0	
6	127.0.0.1:5706	0	0	0	0	0	0	
7	127.0.0.1:5709	37.67	0	37.67	0	0	0	
8	127.0.0.1:5704	0	0	0	0	0	0	
9	127.0.0.1:5713	0	0	0	0	0	0	
10	127.0.0.1:5716	0	0	0	0	0	0	

First Previous 1 2 Next Last


From left to right, this table lists the IP address and port of each node, and counts of offers, rejected offers, polls, poll misses and events.

You can select the period in the combo box placed at top right corner of the window, for which the table data will be shown. Available values are **Since Beginning, Last Minute, Last 10 Minutes** and **Last 1 Hour**.

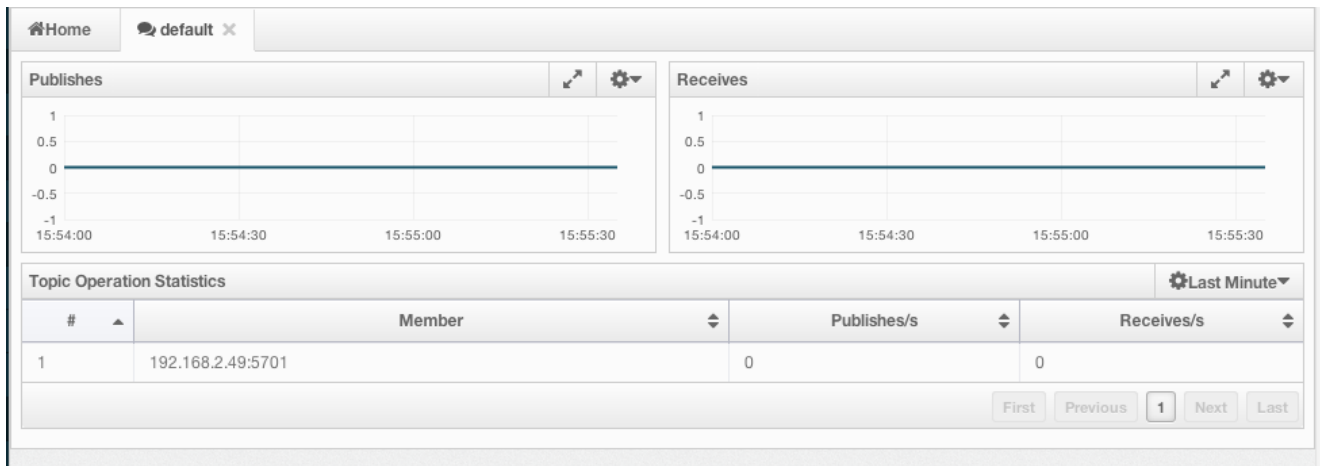
You can navigate through the pages using the buttons placed at the bottom right of the table (**First, Previous, Next, Last**). The order of the listings in each column can be ascended or descended by clicking on column headings.

12.3.6 Topics

You can monitor your topics' metrics by clicking the topic name listed on the left panel under **Topics** menu item. A new tab for monitoring that topic instance is opened on the right, as shown below.

On top of the page, there are two charts to monitor the **Publishes** and **Receives** in real-time. They show the published and received message counts of the cluster, nodes of which are subscribed to the selected topic. Both charts' X-axis shows the current system time. and a chart can be opened as a separate dialog by clicking on the  button placed at top right of each chart.

Under these charts, there is Topic Operation Statistics table. From left to right, this table lists the IP addresses and ports of each node, and counts of message published and receives per second in real-time. You can select the period in the combo box placed at top right corner of the table, for which the table data will be shown. Available values are **Since Beginning, Last Minute, Last 10 Minutes** and **Last 1 Hour**.



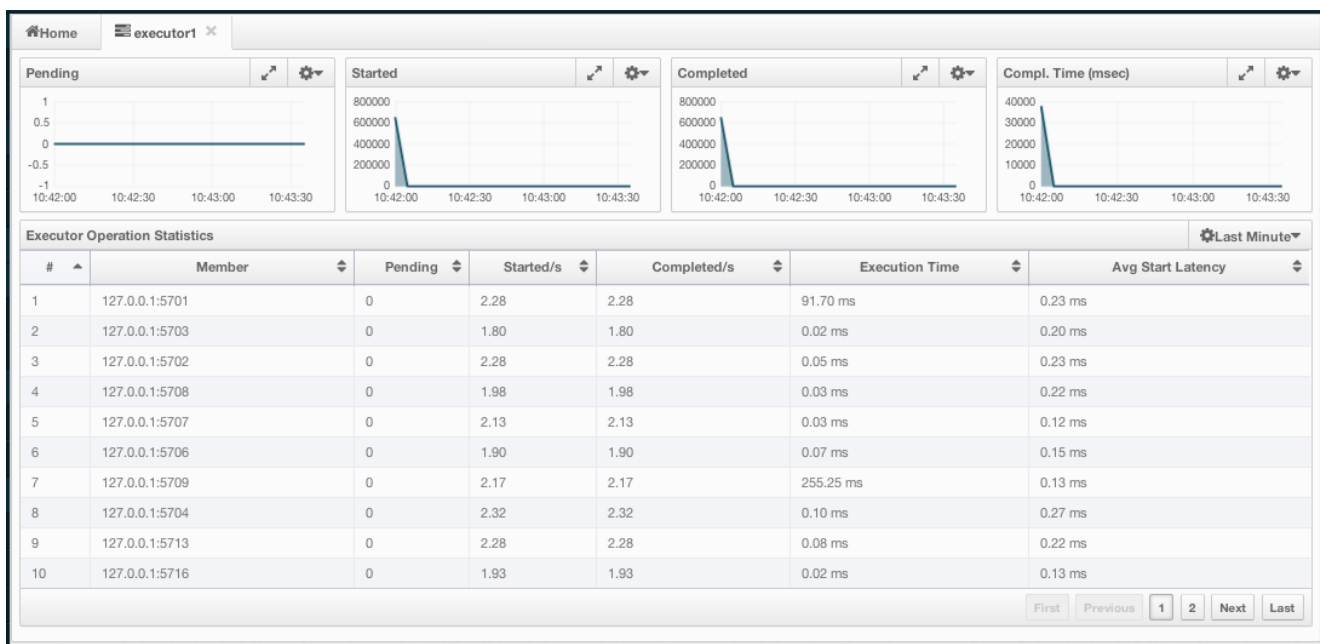
You can navigate through the pages using the buttons placed at the bottom right of the table (**First**, **Previous**, **Next**, **Last**). The order of the listings in each column can be ascended or descended by clicking on column headings.

12.3.7 MultiMaps

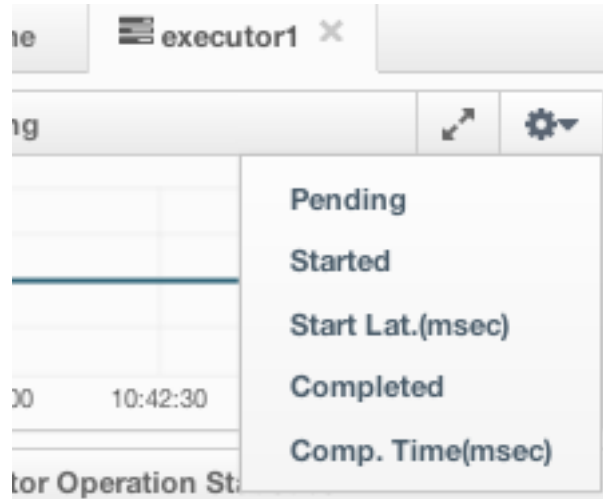
As you know, MultiMap is a specialized map where you can associate a key with multiple values. This monitoring option is similar to the **Maps** one. Same monitoring charts and data tables are used to monitor MultiMaps. Differences are; not being able to browse the MultiMaps and to re-configure it. Please see [Maps](#).


12.3.8 Executors

Executor instances are listed under the **Executors** menu item on the left. When you click on a executor, a new tab for monitoring that executor instance is opened on the right, as shown below.



On top of the page, there are small charts to monitor the pending, started, completed, etc. executors in real-time. All charts' X-axis shows the current system time. Other small monitoring charts can be selected using button placed at top right of each chart. When it is clicked, the whole list of monitoring options are listed, as shown below.



When you click on a desired monitoring, the chart is loaded with the selected option. Also, a chart can be opened as a separate dialog by clicking on the  button placed at top right of each chart. Below monitoring charts are available:

- **Pending:** Monitors the pending executors. Y-axis is the executor count.
- **Started:** Monitors the started executors. Y-axis is the executor count.
- **Start Lat. (msec):** Shows the latency when executors are started. Y-axis is the duration in milliseconds.
- **Completed:** Monitors the completed executors. Y-axis is the executor count.
- **Comp. Time (msec):** Shows the completion period of executors. Y-axis is the duration in milliseconds.

Under these charts, there is **Executor Operation Statistics** table, as shown below.

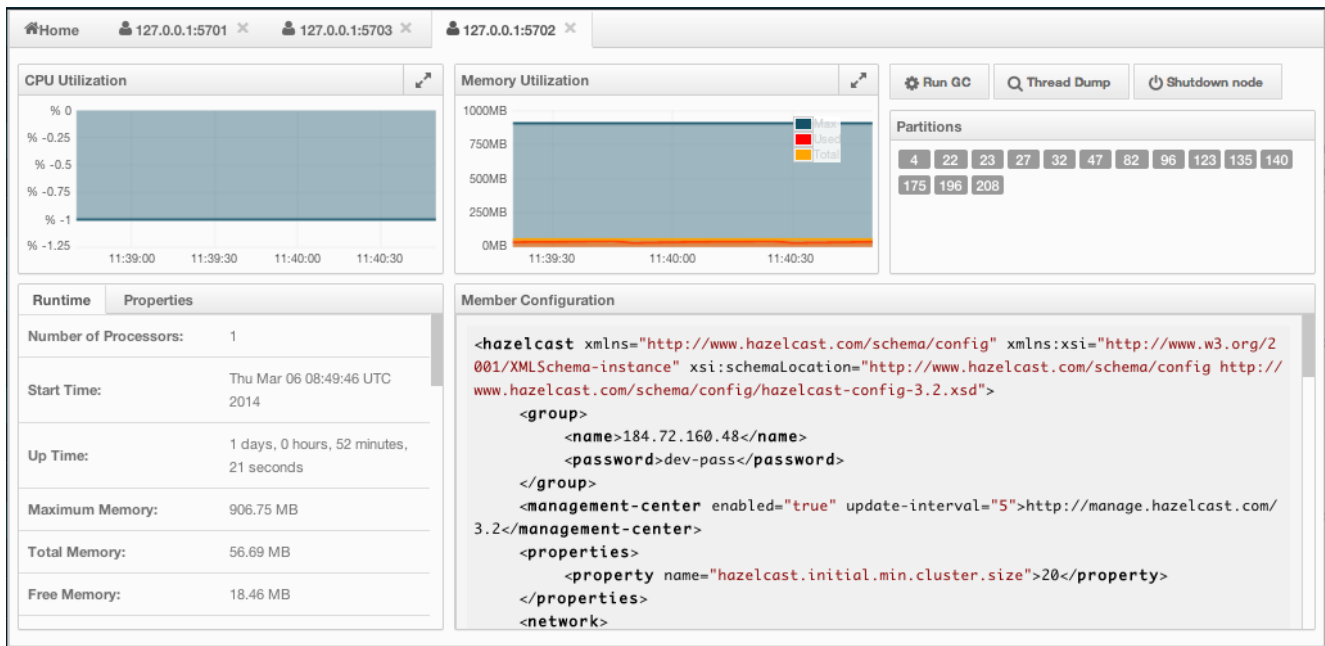
Executor Operation Statistics							Last Minute▼
# ▲	Member ⇅	Pending ⇅	Started/s ⇅	Completed/s ⇅	Execution Time ⇅	Avg Start Latency ⇅	
1	127.0.0.1:5701	0	2.28	2.28	91.70 ms	0.23 ms	
2	127.0.0.1:5703	0	1.80	1.80	0.02 ms	0.20 ms	
3	127.0.0.1:5702	0	2.28	2.28	0.05 ms	0.23 ms	
4	127.0.0.1:5708	0	1.98	1.98	0.03 ms	0.22 ms	
5	127.0.0.1:5707	0	2.13	2.13	0.03 ms	0.12 ms	
6	127.0.0.1:5706	0	1.90	1.90	0.07 ms	0.15 ms	
7	127.0.0.1:5709	0	2.17	2.17	255.25 ms	0.13 ms	
8	127.0.0.1:5704	0	2.32	2.32	0.10 ms	0.27 ms	
9	127.0.0.1:5713	0	2.28	2.28	0.08 ms	0.22 ms	
10	127.0.0.1:5716	0	1.93	1.93	0.02 ms	0.13 ms	
							First Previous 1 2 Next Last


From left to right, this table lists the IP address and port of nodes, counts of pending, started and completed executors per second, execution time and average start latency of executors on each node. You can navigate through the pages using the buttons placed at the bottom right of the table (**First, Previous, Next, Last**). The order of the listings in each column can be ascended or descended by clicking on column headings.

12.3.9 Members

This menu item is used to monitor each cluster member (node) and also perform operations like running garbage collection (GC) and taking a thread dump. Once a member is selected from the menu, a new tab for monitoring that member is opened on the right, as shown below.

CPU Utilization chart shows the CPU usage on the selected member in percentage. **Memory Utilization** chart shows the memory usage on the selected member with three different metrics (maximum, used and total memory).



Both of these charts can be opened as separate windows using the  button placed at top right of each chart, a more clearer view can be obtained by this way.

The window titled with **Partitions** shows which partitions are assigned to the selected member. **Runtime** is a dynamically updated window tab showing the processor number, start and up times, maximum, total and free memory sizes of the selected member. Next to this, there is **Properties** tab showing the system properties. **Member Configuration** window shows the connected Hazelcast cluster's XML configuration.

Besides the aforementioned monitoring charts and windows, there are also operations you can perform on the selected member through this page. You can see operation buttons located at top right of the page, explained below:

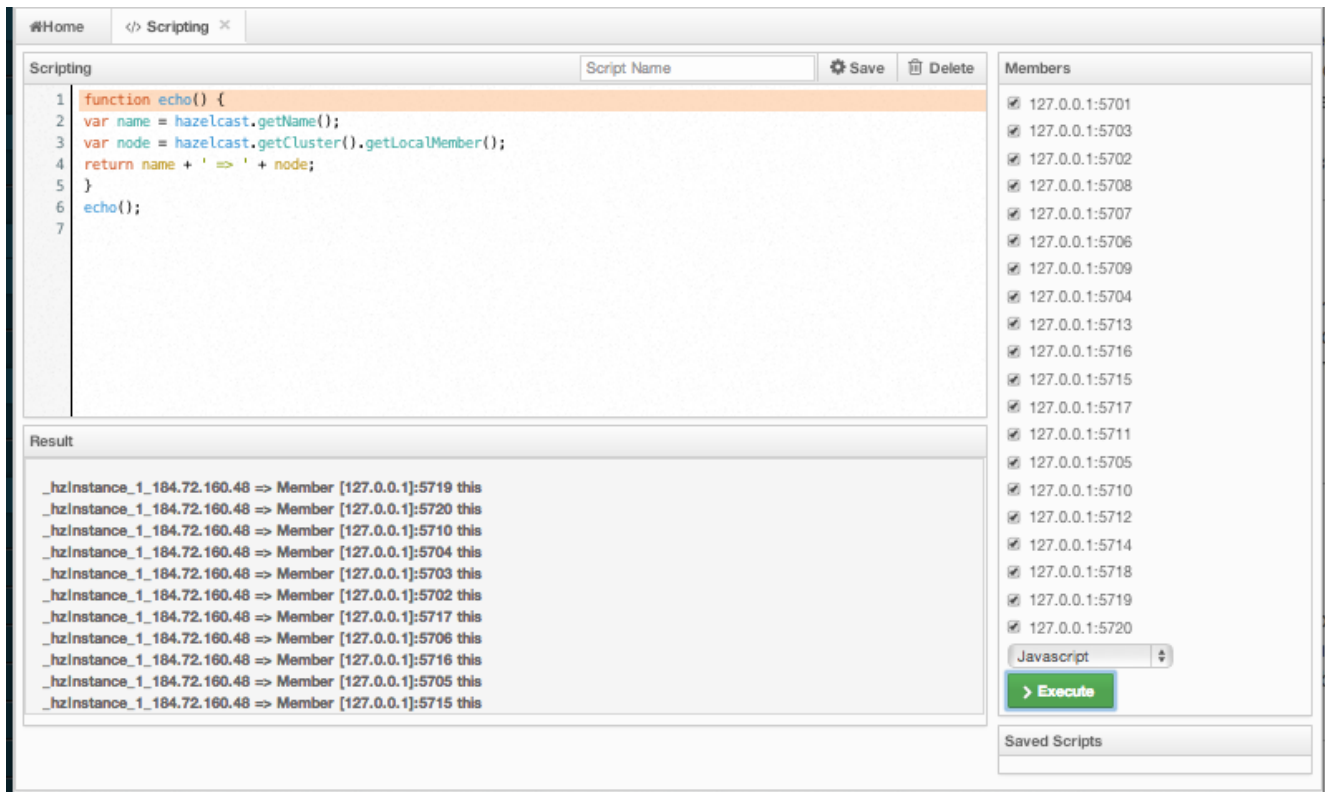
- **Run GC:** When pressed, garbage collection is executed on the selected member. A notification stating that the GC execution was successful will be shown.
- **Thread Dump:** When pressed, thread dump of the selected member is taken and shown as a separate dialog to the user.
- **Shutdown Node:** It is used to shutdown the selected member.

12.3.10 Scripting

Scripting feature of this tool is used to execute codes on the cluster. You can open this feature as a tab by selecting **Scripting** located at the toolbar on top. Once selected, it is opened as shown below.

In this window, **Scripting** part is the actual coding editor. You can select the members on which the code will be executed from the **Members** list shown at the right side of the window. Below the members list there is a combo box enabling you to select a scripting language. Currently, Javascript, Ruby, Groovy and Python languages are supported. After you write your script and press **Execute** button, you can see the execution result in the **Result** part of the window.

There are also **Save** and **Delete** buttons on top right of the scripting editor. You can save your scripts by pressing the **Save** button after you type a name for the script into the field next to this button. The scripts you saved are listed in the **Saved Scripts** part of the window, located at the bottom right of the page. You can simply click on a saved script from this list to execute or edit it. And, if you want to remove a script that you wrote and save before, just select it from this list and press **Delete** button.



12.3.11 Console

Management Center has also a console feature that enables you to execute commands on the cluster. For example, you can perform “put”s and “get”s on a map, after you set the namespace with the command `ns <name of your map>`. Same is valid for queues, topics, etc. To execute your command, just type it into the field below the console and press **Enter**. You can type `help` to see all commands that can be used.

Console window can be opened by clicking on the **Console** button located at the toolbar. A sample view with some commands executed can be seen below.

12.3.12 Alerts

Alerts feature of this tool is used to receive alerts by creating filters. In these filters, criteria can be specified for cluster, nodes or data structures. When the specified criteria are met for a filter, related alert is shown as a pop-up message on top right of the page.

Once the **Alerts** button located at the toolbar is clicked, the page shown below appears.

Creating Filters for Cluster

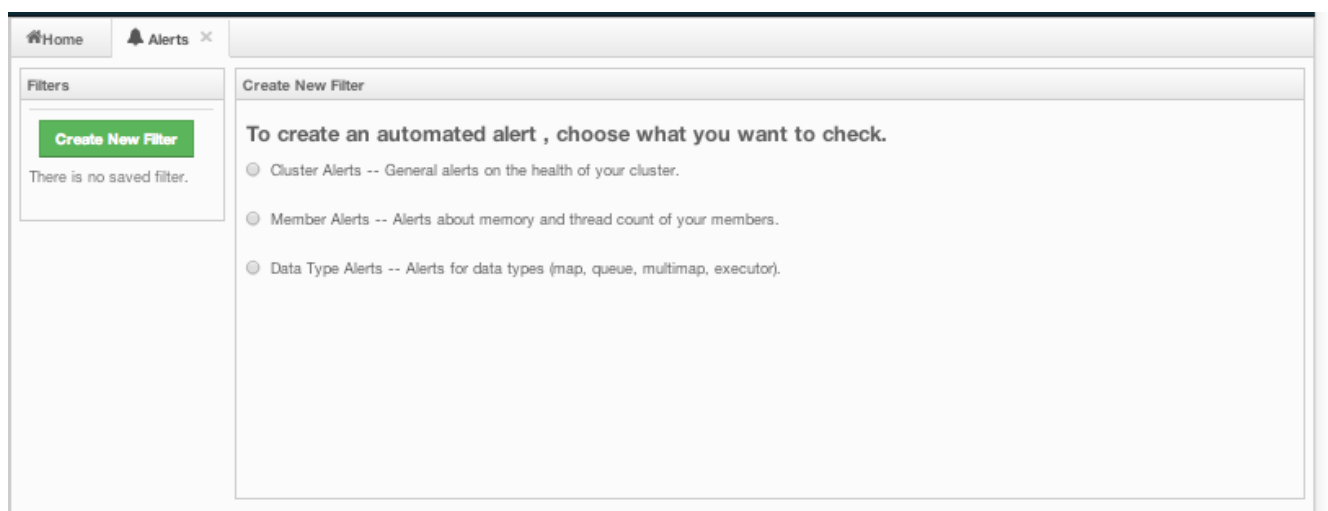
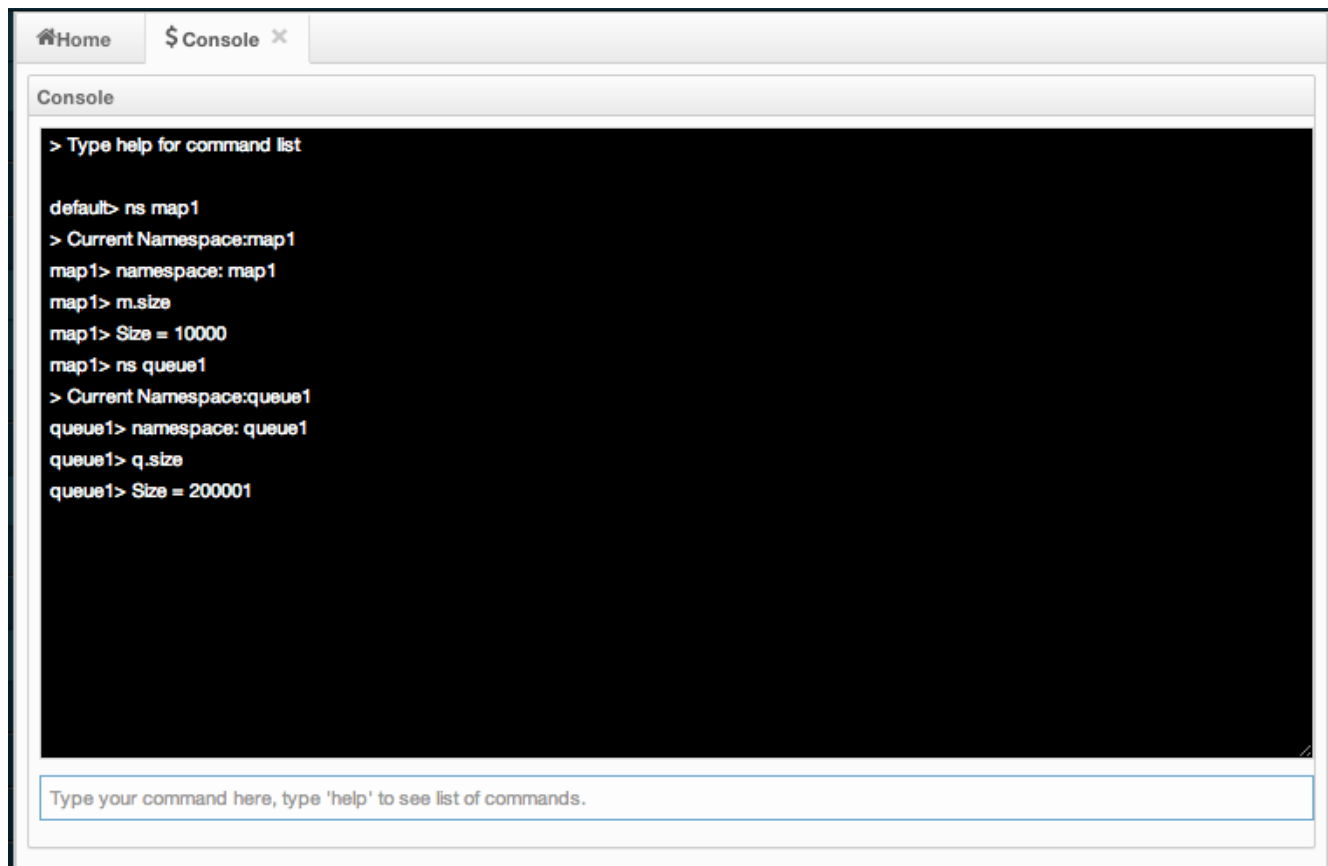
Select **Cluster Alerts** check box to create a cluster wise filter. Once selected, next screen asks the items for which alerts will be created, as shown below.

Select the desired items and click the **Next** button. On the next page shown below, specify the frequency of checks in **hour** and **min** fields, give a name for the filter, select whether notification e-mails will be sent (to no one, only admin or to all users) and select whether the alert data will be written to the disk (if checked, you can see the alert log at the directory `/users//mancenter`).

Click on the **Save** button; your filter will be saved and put into the **Filters** part of the page, as shown below.

You can edit the filter by clicking on the  icon and delete it by clicking on the .

Creating Filters for Cluster Members



Cluster Filter

Choose alert items

☐ Members

☐ Connections

☐ Locks

☐ Migration

☐ Partitions

Cancel

Next

Cluster Filter

Alert Check Frequency

0

hour

10

min

Alert Actions

Filter Name: ShowPartitionStatus

Send Email To : ☐ No One ☒ Admin Only ☐ All

☒ Persist data on disk

Cancel

Save

Filters

Create New Filter

ShowPartitionStatus



Select **Member Alerts** check box to create filters for some or all members in the cluster. Once selected, next screen asks for which members the alert will be created. Select as desired and click on the **Next** button. On the next page shown below, specify the criteria.

Alerts can be created when:

- free memory on the selected nodes is less than the specified number
- used heap memory is larger than the specified number
- number of active threads are less than the specified count
- number of daemon threads are larger than the specified count

When two or more criteria is specified they will be bound with the logical operator **AND**.

On the next page, give a name for the filter, select whether notification e-mails will be sent (to no one, only admin or to all users) and select whether the alert data will be written to the disk (if checked, you can see the alert log at the directory `/users//mancenter`).

Click on the **Save** button; your filter will be saved and put into the **Filters** part of the page. You can edit the filter by clicking on the  icon and delete it by clicking on the  icon.



Creating Filters for Data Types

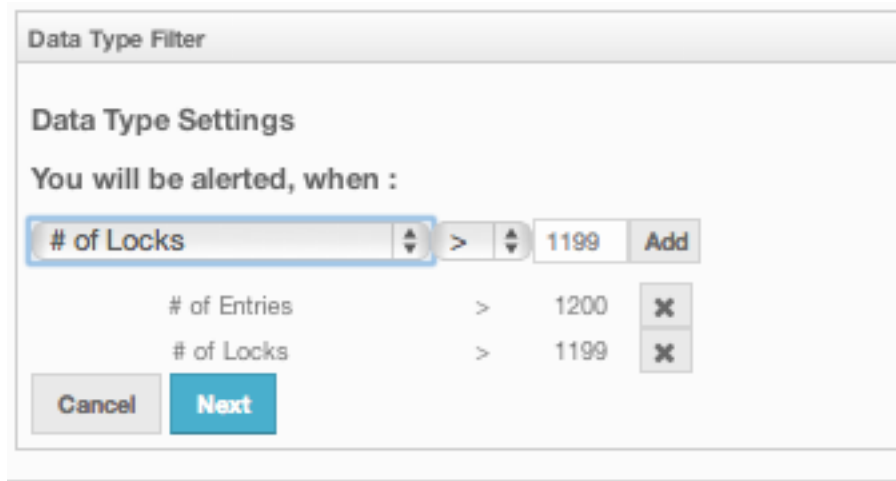
Select **Data Type Alerts** check box to create filters for data structures. Next screen asks for which data structure (maps, queues, multimaps, executors) the alert will be created. Once a structure is selected, next screen immediately loads and wants you to select the data structure instances (i.e. if you selected *Maps*, it will list all the maps defined in the cluster, you can select only one map or more). Select as desired, click on the **Next** button and select the members on which the selected data structure instances run.

Next screen, as shown below, is the one where the criteria for the selected data structure are specified.

As it can be seen, you will select an item from the left combo box, select the operator in the middle one, specify a value in the input field and click on the **Add** button. You can create more than one criteria in this page, and those will be bound by the logical operator **AND**.

After the criteria are specified and **Next** button clicked, give a name for the filter, select whether notification e-mails will be sent (to no one, only admin or to all users) and select whether the alert data will be written to the disk (if checked, you can see the alert log at the directory `/users//mancenter`).

Click on the **Save** button; your filter will be saved and put into the **Filters** part of the page. You can edit the filter by clicking on the  icon and delete it by clicking on the  icon.



Data Type Filter

Data Type Settings

You will be alerted, when :

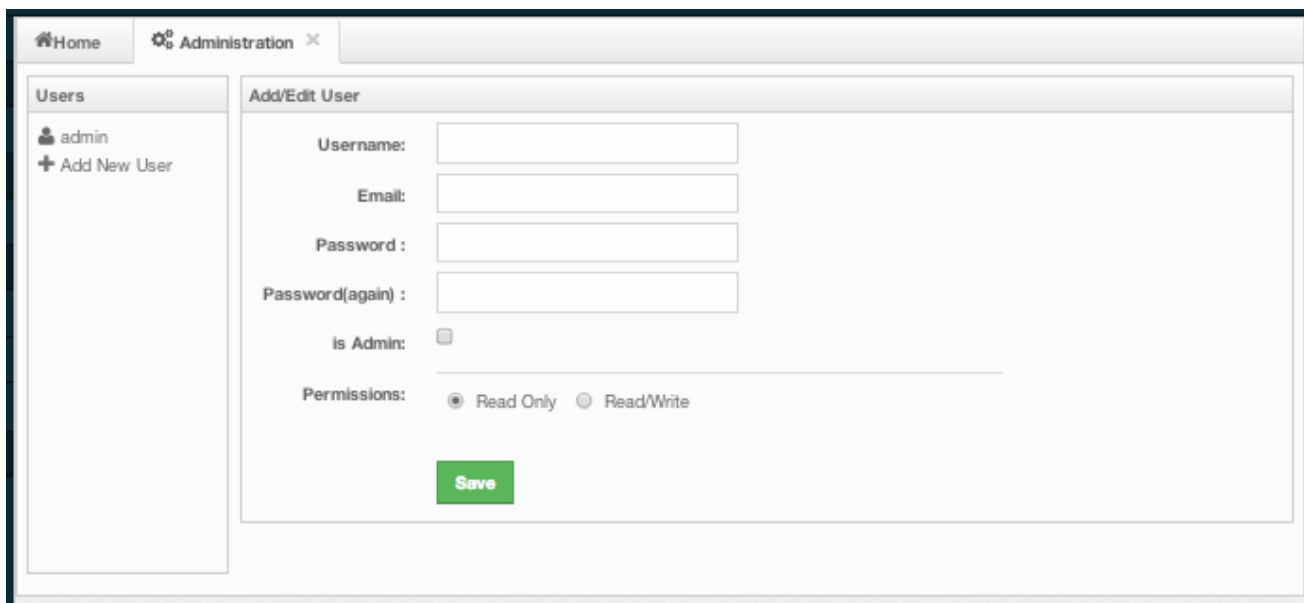
# of Locks	>	1199	Add
# of Entries	>	1200	X
# of Locks	>	1199	X

Cancel Next

12.3.13 Administration

Note: This toolbar item is available only to admin users, i.e. the users who initially have **admin*** as their both usernames and passwords.*

Admin user can add, edit, remove users and specify the permissions for the users of Management Center. To perform these operations, click on **Administration** button located at the toolbar. The page shown below appears.



Home Administration x

Users

- admin
- + Add New User

Add/Edit User

Username:

Email:

Password:

Password(again):

is Admin: ☐

Permissions: ☒ Read Only ☐ Read/Write

Save

To add a user to the system, specify the username, e-mail and password in the **Add/Edit User** part of the page. If the user to be added will have administrator privileges, select **isAdmin** checkbox. **Permissions** checkboxes have two values:

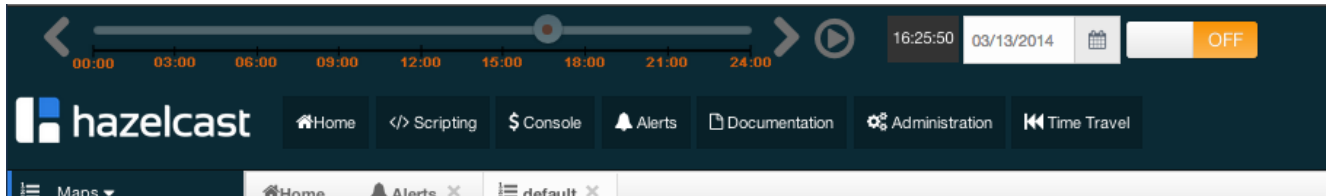
- **Read Only:** If this permission is given to the user, only *Home*, *Documentation* and *Time Travel* items will be visible at the toolbar at that user's session. Also, the users with this permission cannot update a **map configuration**, run a garbage collection and take a thread dump on a node, and shutdown a node (please see **Members** section).
- **Read/Write:** If this permission is given to the user, *Home*, *Scripting*, *Console*, *Documentation* and *Time Travel* items will be visible. The users with this permission can update a map configuration and perform operations on the nodes.

After all fields are entered/selected, click **Save** button to create the user. You will see the newly created user's username on the left side, in the **Users** part of the page.

To edit or delete a user, select a username listed in the **Users**. Selected user's information will appear on the right side of the page. To update the user information, change the fields as desired and click **Save** button. To delete the user from the system, click **Delete** button.

12.3.14 Time Travel

Time Travel is used to check the status of the cluster at a time in the past. Once this item is selected on the toolbar, a small window appears on top of the page, as shown below.



To see the cluster status in a past time, Time Travel should be enabled first. Click on the area where it says **OFF** (on the right of Time Travel window). It will turn to **ON** after it asks whether to enable the Time Travel with a dialog (just click on **Enable**).

Once it is **ON**, it means that the status of your cluster is started to be stored on your disk, as long as your web server is alive.

You can go back in time using the slider and/or calendar and check your cluster's situation at the selected time. All data structures and members can be monitored as if you are using the management center normally (charts and data tables for each data structure and members). Using the arrow buttons placed at both sides of the slider, you can go back or further with steps of 5 seconds. Naturally, it will show the status if Time Travel has been **ON** at the selected time in past. Otherwise, all charts and tables will be shown as empty.

12.3.15 Documentation

To see the documentation, click on the **Documentation** button located at the toolbar. Management Center manual will appear as a tab.

Chapter 13

Security

13.1 Socket Interceptor

Hazelcast allows you to intercept socket connections before a node joins to cluster or a client connects to a node. This provides ability to add custom hooks to join/connection procedure (like identity checking using Kerberos, etc.). You should implement `com.hazelcast.nio.MemberSocketInterceptor` for members and `com.hazelcast.nio.SocketInterceptor` for clients.

```
public class MySocketInterceptor implements MemberSocketInterceptor {
    public void init(SocketInterceptorConfig socketInterceptorConfig) {
        // initialize interceptor
    }

    void onConnect(Socket connectedSocket) throws IOException {
        // do something meaningful when connected
    }

    public void onAccept(Socket acceptedSocket) throws IOException {
        // do something meaningful when accepted a connection
    }
}

<hazelcast>
...
<network>
...
    <socket-interceptor enabled="true">
        <class-name>com.hazelcast.examples.MySocketInterceptor</class-name>
        <properties>
            <property name="kerberos-host">kerb-host-name</property>
            <property name="kerberos-config-file">kerb.conf</property>
        </properties>
    </socket-interceptor>
</network>
...
</hazelcast>

public class MyClientSocketInterceptor implements SocketInterceptor {
    void onConnect(Socket connectedSocket) throws IOException {
        // do something meaningful when connected
    }
}
```

```

}

ClientConfig clientConfig = new ClientConfig();
clientConfig.setGroupConfig(new GroupConfig("dev", "dev-pass")).addAddress("10.10.3.4");

MyClientSocketInterceptor myClientSocketInterceptor = new MyClientSocketInterceptor();
clientConfig.setSocketInterceptor(myClientSocketInterceptor);
HazelcastInstance client = HazelcastClient.newHazelcastClient(clientConfig);

```

13.2 Encryption

Hazelcast allows you to encrypt entire socket level communication among all Hazelcast members. Encryption is based on [Java Cryptography Architecture](#). In symmetric encryption, each node uses the same key, so the key is shared. Here is a sample configuration for symmetric encryption:

```

<hazelcast>
...
<network>
...
<!--
    Make sure to set enabled=true
    Make sure this configuration is exactly the same on
    all members
-->
<symmetric-encryption enabled="true">
    <!--
        encryption algorithm such as
        DES/ECB/PKCS5Padding,
        PBEWithMD5AndDES,
        Blowfish,
        DESede
    -->
    <algorithm>PBEWithMD5AndDES</algorithm>

    <!-- salt value to use when generating the secret key -->
    <salt>thesalt</salt>

    <!-- pass phrase to use when generating the secret key -->
    <password>thepass</password>

    <!-- iteration count to use when generating the secret key -->
    <iteration-count>19</iteration-count>
</symmetric-encryption>
</network>
...
</hazelcast>

```

Related Information

Please see [SSL](#).

13.3 SSL

Hazelcast allows you to use SSL socket communication among all Hazelcast members. You need to implement `com.hazelcast.nio.ssl.SSLContextFactory` and configure SSL section in network configuration.

```

public class MySSLContextFactory implements SSLContextFactory {
    public void init(Properties properties) throws Exception {
    }

    public SSLContext getSSLContext() {
        ...
        SSLContext sslCtx = SSLContext.getInstance(protocol);
        return sslCtx;
    }
}

<hazelcast>
...
<network>
...
    <ssl enabled="true">
        <factory-class-name>com.hazelcast.examples.MySSLContextFactory</factory-class-name>
        <properties>
            <property name="foo">bar</property>
        </properties>
    </ssl>
</network>
...
</hazelcast>

```

Hazelcast provides a default SSLContextFactory; `com.hazelcast.nio.ssl.BasicSSLContextFactory` which uses configured keystore to initialize SSLContext. Just define `keyStore` and `keyStorePassword`, and also you can set `keyManagerAlgorithm` (default `SunX509`), `trustManagerAlgorithm` (default `SunX509`) and `protocol` (default `TLS`).

```

<hazelcast>
...
<network>
...
    <ssl enabled="true">
        <factory-class-name>com.hazelcast.nio.ssl.BasicSSLContextFactory</factory-class-name>
        <properties>
            <property name="keyStore">keyStore</property>
            <property name="keyStorePassword">keyStorePassword</property>
            <property name="keyManagerAlgorithm">SunX509</property>
            <property name="trustManagerAlgorithm">SunX509</property>
            <property name="protocol">TLS</property>
        </properties>
    </ssl>
</network>
...
</hazelcast>

```

Hazelcast client has SSL support too. Client SSL configuration can be defined using programmatic configuration as shown below.

```

'java Properties props = new Properties(); ... ClientConfig config = new ClientConfig();
config.getSocketOptions().setSocketFactory(new SSLSocketFactory(props));

```

You can also set `keyStore` and `keyStorePassword` through `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword` system properties.

Note: You cannot use SSL when *Hazelcast Encryption* is enabled.

13.4 Enabling Security for Hazelcast Enterprise

Enterprise Only

Hazelcast has an extensible, JAAS based security feature which can be used to authenticate both cluster members and clients and to perform access control checks on client operations. Access control can be done according to endpoint principal and/or endpoint address.

Security can be enabled as stated in the below programmatic or declarative configuration.

```
<hazelcast xsi:schemaLocation="http://www.hazelcast.com/schema/config
    http://www.hazelcast.com/schema/config/hazelcast-config-3.2.xsd"
    xmlns="http://www.hazelcast.com/schema/config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    ...

    <security enabled="true">
        ...
    </security>
</hazelcast>
```

```
Config cfg = new Config();
SecurityConfig securityCfg = cfg.getSecurityConfig();
securityCfg.setEnabled(true);
```

Also, please see [Setting License Key](#).

13.5 Credentials

Enterprise Only

One of the key elements in Hazelcast security is `Credentials` object. It is used to carry all credentials of an endpoint (member or client). `Credentials` is an interface which extends `Serializable` and has three methods to be implemented. The users can either implement `Credentials` interface or extend `AbstractCredentials` class, which is an abstract implementation of `Credentials`, according to their needs.

```
package com.hazelcast.security;
public interface Credentials extends Serializable {
    String getEndpoint();
    void setEndpoint(String endpoint) ;
    String getPrincipal() ;
}
```

`Credentials.setEndpoint()` method is called by Hazelcast when authentication request arrives to node before authentication takes place.

```
package com.hazelcast.security;
...
public abstract class AbstractCredentials implements Credentials, DataSerializable {
    private transient String endpoint;
    private String principal;
    ...
}
```

UsernamePasswordCredentials, a custom implementation of Credentials can be found in Hazelcast com.hazelcast.security package. It is used by default configuration during authentication process of both members and clients.

```
package com.hazelcast.security;
...
public class UsernamePasswordCredentials extends Credentials {
    private byte[] password;
    ...
}
```

13.6 ClusterLoginModule

Enterprise Only

All security attributes are carried in Credentials object and Credentials is used by LoginModules during authentication process. Accessing user supplied attributes from LoginModules is done by CallbackHandlers. To provide access to Credentials object, Hazelcast uses its own specialized CallbackHandler. During initialization of LoginModules Hazelcast will pass this special CallbackHandler into LoginModule.initialize() method.

LoginModule implementations should create an instance of com.hazelcast.security.CredentialsCallback and call handle(Callback[] callbacks) method of CallbackHandler during login process. CredentialsCallback.getCredentials() will return supplied Credentials object.

```
public class CustomLoginModule implements LoginModule {
    CallbackHandler callbackHandler;
    Subject subject;

    public final void initialize(Subject subject, CallbackHandler callbackHandler,
        Map<String, ?> sharedState, Map<String, ?> options) {
        this.subject = subject;
        this.callbackHandler = callbackHandler;
    }

    public final boolean login() throws LoginException {
        CredentialsCallback callback = new CredentialsCallback();
        try {
            callbackHandler.handle(new Callback[]{callback});
            credentials = cb.getCredentials();
        } catch (Exception e) {
            throw new LoginException(e.getMessage());
        }
        ...
    }
    ...
}
```

To use default Hazelcast permission policy, an instance of com.hazelcast.security.ClusterPrincipal that holding Credentials object must be created and added to Subject.principals on LoginModule.commit() as shown below.

```
public class MyCustomLoginModule implements LoginModule {
    ...
    public boolean commit() throws LoginException {
        ...
    }
}
```

```

    final Principal principal = new ClusterPrincipal(credentials);
    subject.getPrincipals().add(principal);

    return true;
}
...
}

```

Hazelcast also has an abstract implementation of `LoginModule` that does callback and cleanup operations and holds resulting `Credentials` instance. `LoginModules` extending `ClusterLoginModule` can access `Credentials`, `Subject`, `LoginModule` instances and options and `sharedState` maps. Extending `ClusterLoginModule` is recommended instead of implementing all required stuff.

```

package com.hazelcast.security;
...
public abstract class ClusterLoginModule implements LoginModule {

    protected abstract boolean onLogin() throws LoginException;
    protected abstract boolean onCommit() throws LoginException;
    protected abstract boolean onAbort() throws LoginException;
    protected abstract boolean onLogout() throws LoginException;

}

```

13.7 Cluster Member Security

Enterprise Only

Hazelcast supports standard Java Security (JAAS) based authentication between cluster members. You should configure one or more `LoginModules` and an instance of `com.hazelcast.security.ICredentialsFactory`. Although Hazelcast has default implementations using cluster group and group-password and `UsernamePasswordCredentials` on authentication, it is advised to implement these according to specific needs and environment.

```

<security enabled="true">
  <member-credentials-factory class-name="com.hazelcast.examples.MyCredentialsFactory">
    <properties>
      <property name="property1">value1</property>
      <property name="property2">value2</property>
    </properties>
  </member-credentials-factory>
  <member-login-modules>
    <login-module class-name="com.hazelcast.examples.MyRequiredLoginModule" usage="required">
      <properties>
        <property name="property3">value3</property>
      </properties>
    </login-module>
    <login-module class-name="com.hazelcast.examples.MySufficientLoginModule" usage="sufficient">
      <properties>
        <property name="property4">value4</property>
      </properties>
    </login-module>
    <login-module class-name="com.hazelcast.examples.MyOptionalLoginModule" usage="optional">
      <properties>
        <property name="property5">value5</property>
      </properties>
    </login-module>
  </member-login-modules>
</security>

```

```

        </login-module>
    </member-login-modules>
    ...
</security>

```

You can define as many `asLoginModules` you wanted in configuration. Those are executed in given order. Usage attribute has 4 values; 'required', 'requisite', 'sufficient' and 'optional' as defined in `javax.security.auth.login.AppConfigurationEntry.LoginModuleControlFlag`.

```

package com.hazelcast.security;
/**
 * ICredentialsFactory is used to create Credentials objects to be used
 * during node authentication before connection accepted by master node.
 */
public interface ICredentialsFactory {

    void configure(GroupConfig groupConfig, Properties properties);

    Credentials newCredentials();

    void destroy();
}

```

Properties defined in configuration are passed to `ICredentialsFactory.configure()` method as `java.util.Properties` and to `LoginModule.initialize()` method as `java.util.Map`.

13.8 Native Client Security

Enterprise Only

Hazelcast's Client security includes both authentication and authorization.

13.8.1 Authentication

Authentication mechanism just works the same as cluster member authentication. Implementation of client authentication requires a `Credentials` and one or more `LoginModule(s)`. Client side does not have/need a factory object to create `Credentials` objects like `ICredentialsFactory`. `Credentials` must be created at client side and sent to connected node during connection process.

```

<security enabled="true">
    <client-login-modules>
        <login-module class-name="com.hazelcast.examples.MyRequiredClientLoginModule" usage="required">
            <properties>
                <property name="property3">value3</property>
            </properties>
        </login-module>
        <login-module class-name="com.hazelcast.examples.MySufficientClientLoginModule" usage="sufficient">
            <properties>
                <property name="property4">value4</property>
            </properties>
        </login-module>
        <login-module class-name="com.hazelcast.examples.MyOptionalClientLoginModule" usage="optional">
            <properties>

```

```

        <property name="property5">value5</property>
    </properties>
</login-module>
</client-login-modules>
...
</security>

```

You can define as many as LoginModules you want in configuration. Those are executed in the given order. Usage attribute has 4 values; 'required', 'requisite', 'sufficient' and 'optional' as defined in `javax.security.auth.login.AppConfigurationEntry.LoginModuleControlFlag`.

```

final Credentials credentials = new UsernamePasswordCredentials("dev", "dev-pass");
HazelcastInstance client = HazelcastClient.newHazelcastClient(credentials, "localhost");

```

13.8.2 Authorization

Hazelcast client authorization is configured by a client permission policy. Hazelcast has a default permission policy implementation that uses permission configurations defined in Hazelcast security configuration. Default policy permission checks are done against instance types (map, queue, etc.), instance names (map, queue, etc. name), instance actions (put, read, remove, add, etc.), client endpoint addresses and client principal defined by Credentials object. Instance and principal names and endpoint addresses can be defined as wildcards(*). Please see [Network Configuration](#) and [Wildcard Configuration](#) sections.

```

<security enabled="true">
  <client-permissions>
    <!-- Principal 'admin' from endpoint '127.0.0.1' has all permissions. -->
    <all-permissions principal="admin">
      <endpoints>
        <endpoint>127.0.0.1</endpoint>
      </endpoints>
    </all-permissions>

    <!-- Principals named 'dev' from all endpoints have 'create', 'destroy',
    'put', 'read' permissions for map named 'default'. -->
    <map-permission name="default" principal="dev">
      <actions>
        <action>create</action>
        <action>destroy</action>
        <action>put</action>
        <action>read</action>
      </actions>
    </map-permission>

    <!-- All principals from endpoints '127.0.0.1' or matching to '10.10.*.*'
    have 'put', 'read', 'remove' permissions for map
    whose name matches to 'com.foo.entity.*'. -->
    <map-permission name="com.foo.entity.*">
      <endpoints>
        <endpoint>10.10.*.*</endpoint>
        <endpoint>127.0.0.1</endpoint>
      </endpoints>
      <actions>
        <action>put</action>
        <action>read</action>
        <action>remove</action>
      </actions>
    </map-permission>
  </client-permissions>
</security>

```



```

    <!-- Principals named 'dev' from endpoints matching to either
         '192.168.1.1-100' or '192.168.2.*'
         have 'create', 'add', 'remove' permissions for all queues. -->
    <queue-permission name="*" principal="dev">
        <endpoints>
            <endpoint>192.168.1.1-100</endpoint>
            <endpoint>192.168.2.*</endpoint>
        </endpoints>
        <actions>
            <action>create</action>
            <action>add</action>
            <action>remove</action>
        </actions>
    </queue-permission>

    <!-- All principals from all endpoints have transaction permission.-->
    <transaction-permission />
</client-permissions>
</security>

```

The users also can define their own policy by implementing `com.hazelcast.security.IPermissionPolicy`.

```

package com.hazelcast.security;
/**
 * IPermissionPolicy is used to determine any Subject's
 * permissions to perform a security sensitive Hazelcast operation.
 *
 */
public interface IPermissionPolicy {
    void configure(SecurityConfig securityConfig, Properties properties);

    PermissionCollection getPermissions(Subject subject, Class<? extends Permission> type);

    void destroy();
}

```

Permission policy implementations can access client-permissions in configuration by using `SecurityConfig.getClientPermissions`. during `configure(SecurityConfig securityConfig, Properties properties)` method is called by Hazelcast.

`IPermissionPolicy.getPermissions(Subject subject, Class<? extends Permission> type)` method is used to determine a client request has been granted permission to do a security-sensitive operation.

Permission policy should return a `PermissionCollection` containing permissions of given type for given `Subject`. Hazelcast access controller will call `PermissionCollection.implies(Permission)` on returning `PermissionCollection` and will decide if current `Subject` has permitted to access to requested resources or not.

13.8.3 Permissions

- All Permission

```

<all-permissions principal="principal">
    <endpoints>
        ...
    </endpoints>
</all-permissions>

```

- Map Permission

```

<map-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</map-permission>

```

Actions: all, create, destroy, put, read, remove, lock, intercept, index, listen

- Queue Permission

```

<queue-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</queue-permission>

```

Actions: all, create, destroy, add, remove, read, listen

- Multimap Permission

```

<multimap-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</multimap-permission>

```

Actions: all, create, destroy, put, read, remove, listen, lock

- Topic Permission

```

<topic-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</topic-permission>

```

Actions: create, destroy, publish, listen

- List Permission

```

<list-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</list-permission>

```

Actions: all, create, destroy, add, read, remove, listen

- Set Permission

```
<set-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</set-permission>
```

Actions: all, create, destroy, add, read, remove, listen

- Lock Permission

```
<lock-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</lock-permission>
```

Actions: all, create, destroy, lock, read

- AtomicLong Permission

```
<atomic-long-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</atomic-long-permission>
```

Actions: all, create, destroy, read, modify

- CountdownLatch Permission

```
<countdown-latch-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</countdown-latch-permission>
```

Actions: all, create, destroy, modify, read

- Semaphore Permission

```
<semaphore-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</semaphore-permission>
```

Actions: all, create, destroy, acquire, release, read

- Executor Service Permission

```
<executor-service-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</executor-service-permission>
```

Actions: all, create, destroy

- Transaction Permission

```
<transaction-permission principal="principal">
  <endpoints>
    ...
  </endpoints>
</transaction-permission>
```

Chapter 14

Performance

14.1 Data Affinity

Co-location of related data and computation

Hazelcast has a standard way of finding out which member owns/manages each key object. Following operations will be routed to the same member, since all of them are operating based on the same key, “key1”.

```
Config cfg = new Config();
HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);
Map mapa = instance.getMap("mapa");
Map mapb = instance.getMap("mapb");
Map mapc = instance.getMap("mapc");
mapa.put("key1", value);
mapb.get("key1");
mapc.remove("key1");
// since map names are different, operation will be manipulating
// different entries, but the operation will take place on the
// same member since the keys ("key1") are the same

instance.getLock("key1").lock();
// lock operation will still execute on the same member of the cluster
// since the key ("key1") is same

instance.getExecutorService().executeOnKeyOwner(runnable, "key1");
// distributed execution will execute the 'runnable' on the same member
// since "key1" is passed as the key.
```

So, when the keys are the same, then entries are stored on the same node. But we sometimes want to have related entries stored on the same node. Consider customer and his/her order entries. We would have customers map with customerId as the key and orders map with orderId as the key. Since customerId and orderIds are different keys, customer and his/her orders may fall into different members/nodes in your cluster. So how can we have them stored on the same node? The trick here is to create an affinity between customer and orders. If we can somehow make them part of the same partition then these entries will be co-located. We achieve this by making orderIds PartitionAware.

```
public class OrderKey implements Serializable, PartitionAware {
    int customerId;
    int orderId;

    public OrderKey(int orderId, int customerId) {
        this.customerId = customerId;
    }
}
```

```

        this.orderId = orderId;
    }

    public int getCustomerId() {
        return customerId;
    }

    public int getOrderId() {
        return orderId;
    }

    public Object getPartitionKey() {
        return customerId;
    }

    @Override
    public String toString() {
        return "OrderKey{" +
            "customerId=" + customerId +
            ", orderId=" + orderId +
            '}';
    }
}

```

Notice that `OrderKey` implements `PartitionAware` and `getPartitionKey()` returns the `customerId`. This will make sure that `Customer` entry and its `Orders` are going to be stored on the same node.

```

Config cfg = new Config();
HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);
Map mapCustomers = instance.getMap("customers")
Map mapOrders = instance.getMap("orders")
// create the customer entry with customer id = 1
mapCustomers.put(1, customer);
// now create the orders for this customer
mapOrders.put(new OrderKey(21, 1), order);
mapOrders.put(new OrderKey(22, 1), order);
mapOrders.put(new OrderKey(23, 1), order);

```

Assume that you have a customers map where `customerId` is the key and the customer object is the value, customer object contains the customer's orders, and you want to remove one of the orders of a customer and return the number of remaining orders. Here is how you would normally do it:

```

public static int removeOrder(long customerId, long orderId) throws Exception {
    IMap<Long, Customer> mapCustomers = instance.getMap("customers");
    mapCustomers.lock(customerId);
    Customer customer = mapCustomers.get(customerId);
    customer.removeOrder(orderId);
    mapCustomers.put(customerId, customer);
    mapCustomers.unlock(customerId);
    return customer.getOrderCount();
}

```

There are couple of things you should consider:

1. There are four distributed operations there: lock, get, put, unlock. Can you reduce the number of distributed operations?

2. Customer object may not be that big, but can you not have to pass that object through the wire? Notice that, customer object is being passed through the wire twice; get and put.

So instead, why not moving the computation over to the member (JVM) where your customer data actually is. Here is how you can do this with distributed executor service:

1. Send a `PartitionAware Callable` task.
2. `Callable` does the deletion of the order right there and returns with the remaining order count.
3. Upon completion of the `Callable` task, return the result (remaining order count). Plus, you do not have to wait until the task is completed; since distributed executions are asynchronous, you can do other things in the meantime.

Here is a sample code:

```
static Config cfg = new Config();
static HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);

public static int removeOrder(long customerId, long orderId) throws Exception {
    IExecutorService es = instance.getExecutorService("ExecutorService");
    OrderDeletionTask task = new OrderDeletionTask(customerId, orderId);
    Future<Integer> future = es.submit(task);
    int remainingOrders = future.get();
    return remainingOrders;
}

public static class OrderDeletionTask implements Callable<Integer>, PartitionAware, Serializable {

    private long customerId;
    private long orderId;

    public OrderDeletionTask() {
    }
    public OrderDeletionTask(long customerId, long orderId) {
        super();
        this.customerId = customerId;
        this.orderId = orderId;
    }
    public Integer call () {
        IMap<Long, Customer> mapCustomers = instance.getMap("customers");
        mapCustomers.lock (customerId);
        Customer customer = mapCustomers.get(customerId);
        customer.removeOrder (orderId);
        mapCustomers.put(customerId, customer);
        mapCustomers.unlock(customerId);
        return customer.getOrderCount();
    }

    public Object getPartitionKey() {
        return customerId;
    }
}
```

Benefits of doing the same operation with distributed `ExecutorService` based on the key are:

- Only one distributed execution (`es.submit(task)`), instead of four.

- Less data is sent over the wire.
- Since lock/update/unlock cycle is done locally (local to the customer data), lock duration for the `Customer` entry is much less, so enabling higher concurrency.

Chapter 15

WAN

Enterprise Only

15.1 WAN Replication

There are cases where you would need to synchronize multiple clusters. Synchronization of clusters is named as WAN (Wide Area Network) Replication because it is mainly used for replicating different clusters running on WAN.

Imagine having different clusters in New York, London and Tokyo. Each cluster would be operating at very high speed in their LAN (Local Area Network) settings but you would want some or all parts of the data in these clusters replicating to each other. So, updates in Tokyo cluster goes to London and NY, in the meantime updates in New York cluster is synchronized to Tokyo and London.

You can setup active-passive WAN Replication where only one active node replicating its updates on the passive one. You can also setup active-active replication where each cluster is actively updating and replication to the other cluster(s).

In the active-active replication setup, there might be cases where each node is updating the same entry in the same named distributed map. Thus, conflicts will occur when merging. For those cases, a conflict resolution will be needed. Below is how you can setup WAN Replication for London cluster for instance.

```
<hazelcast>
  <wan-replication name="my-wan-cluster">
    <target-cluster group-name="tokyo" group-password="tokyo-pass">
      <replication-impl>com.hazelcast.wan.WanNoDelayReplication</replication-impl>
      <end-points>
        <address>10.2.1.1:5701</address>
        <address>10.2.1.2:5701</address>
      </end-points>
    </target-cluster>
    <target-cluster group-name="london" group-password="london-pass">
      <replication-impl>com.hazelcast.wan.wan.WanNoDelayReplication</replication-impl>
      <end-points>
        <address>10.3.5.1:5701</address>
        <address>10.3.5.2:5701</address>
      </end-points>
    </target-cluster>
  </wan-replication>

<network>
...
```

```

    </network>
...
</hazelcast>

```

This can be the configuration of the cluster running in NY, replicating to Tokyo and London. Tokyo and London clusters should have similar configurations if they are also active replicas.

If NY and London cluster configurations contain **wan-replication** element and Tokyo cluster does not, it means NY and London are active endpoints and Tokyo is passive endpoint.

As noted earlier, you can have Hazelcast replicating some or all of the data in your clusters. You might have 5 different distributed maps but you might want only one of these maps replicating across clusters. So you mark which maps to be replicated by adding **wan-replication-ref** element into map configuration as shown below.

```

<hazelcast>
  <wan-replication name="my-wan-cluster">
    ...
  </wan-replication>

  <network>
    ...
  </network>
  <map name="my-shared-map">
    ...
    <wan-replication-ref name="my-wan-cluster">
      <merge-policy>com.hazelcast.map.merge.PassThroughMergePolicy</merge-policy>
    </wan-replication-ref>
  </map>
...
</hazelcast>

```

Here we have **my-shared-map** is configured to replicate itself to the cluster targets defined in the **wan-replication** element.

Note that, you will also need to define a **merge policy** for merging replica entries and resolving conflicts during the merge.

Related Information

You can download the white paper Hazelcast on AWS: Best Practices for Deployment* from [Hazelcast.com](https://www.hazelcast.com/whitepapers/hazelcast-on-aws/).*

Chapter 16

Configuration

Hazelcast can be configured declaratively (XML) or programmatically (API) or even by the mix of both.

1- Declarative Configuration

If you are creating new Hazelcast instance with passing `null` parameter to `Hazelcast.newHazelcastInstance(null)` or just using empty factory method (`Hazelcast.newHazelcastInstance()`), Hazelcast will look into two places for the configuration file:

- **System property:** Hazelcast will first check if “`hazelcast.config`” system property is set to a file path. Example: `-Dhazelcast.config=C:/myhazelcast.xml`.
- **Classpath:** If config file is not set as a system property, Hazelcast will check classpath for `hazelcast.xml` file.

If Hazelcast does not find any configuration file, it will happily start with default configuration (`hazelcast-default.xml`) located in `hazelcast.jar`. (Before configuring Hazelcast, please try to work with default configuration to see if it works for you. Default should be just fine for most of the users. If not, then consider custom configuration for your environment.)

If you want to specify your own configuration file to create `Config`, Hazelcast supports several ways including filesystem, classpath, `InputStream`, `URL`, etc.:

- `Config cfg = new XmlConfigBuilder(xmlFileName).build();`
- `Config cfg = new XmlConfigBuilder(inputStream).build();`
- `Config cfg = new ClasspathXmlConfig(xmlFileName);`
- `Config cfg = new FileSystemXmlConfig(configFilename);`
- `Config cfg = new UrlXmlConfig(url);`
- `Config cfg = new InMemoryXmlConfig(xml);`

2- Programmatic Configuration

To configure Hazelcast programmatically, just instantiate a `Config` object and set/change its properties/attributes due to your needs.

```
Config cfg = new Config();
cfg.setPort(5900);
cfg.setPortAutoIncrement(false);
```

```
NetworkConfig network = cfg.getNetworkConfig();
JoinConfig join = network.getJoin();
```

```

join.getMulticastConfig().setEnabled(false);
join.getTcpIpConfig().addMember("10.45.67.32").addMember("10.45.67.100")
    .setRequiredMember("192.168.10.100").setEnabled(true);
network.getInterfaces().setEnabled(true).addInterface("10.45.67.*");

MapConfig mapCfg = new MapConfig();
mapCfg.setName("testMap");
mapCfg.setBackupCount(2);
mapCfg.getMaxSizeConfig().setSize(10000);
mapCfg.setTimeToLiveSeconds(300);

MapStoreConfig mapStoreCfg = new MapStoreConfig();
mapStoreCfg.setClassName("com.hazelcast.examples.DummyStore").setEnabled(true);
mapCfg.setMapStoreConfig(mapStoreCfg);

NearCacheConfig nearCacheConfig = new NearCacheConfig();
nearCacheConfig.setMaxSize(1000).setMaxIdleSeconds(120).setTimeToLiveSeconds(300);
mapCfg.setNearCacheConfig(nearCacheConfig);

cfg.addMapConfig(mapCfg);

```

After creating Config object, you can use it to create a new Hazelcast instance.

- `HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance(cfg);`
- To create a named HazelcastInstance you should set `instanceName` of Config object.

```

'''java
Config cfg = new Config();
config.setInstanceName('my-instance');
Hazelcast.newHazelcastInstance(config);
'''

```

- To retrieve an existing HazelcastInstance using its name, use;

```
'Hazelcast.getHazelcastInstanceByName('my-instance');'
```

- To retrieve all existing HazelcastInstances, use;

```
'Hazelcast.getAllHazelcastInstances();'
```

16.1 Network Configuration

16.1.1 Configuring TCP/IP Cluster

If multicast is not preferred as the way of discovery for your environment, then you can configure Hazelcast for full TCP/IP cluster. As below configuration shows, while `enable` attribute of `multicast` is set to false, `tcp-ip` has to be set to true. For the none-multicast option, all or subset of nodes' hostnames and/or IP addresses must be listed. Note that, all of the cluster members do not have to be listed there but at least one of them has to be active in cluster when a new member joins. The `tcp-ip` tag accepts an attribute called `connection-timeout-seconds` whose default value is 5. Increasing this value is recommended if you have many IPs listed and members cannot properly build up the cluster.

```

<hazelcast>
...
<network>

```

```

    <port auto-increment="true">5701</port>
    <join>
      <multicast enabled="false">
        <multicast-group>224.2.2.3</multicast-group>
        <multicast-port>54327</multicast-port>
      </multicast>
      <tcp-ip enabled="true">
        <member>machine1</member>
        <member>machine2</member>
        <member>machine3:5799</member>
        <member>192.168.1.0-7</member>
        <member>192.168.1.21</member>
      </tcp-ip>
    </join>
    ...
  </network>
  ...
</hazelcast>

```

16.1.2 Specifying Network Interfaces

You can also specify which network interfaces that Hazelcast should use. Servers mostly have more than one network interface so you may want to list the valid IPs. Range characters ('*' and '-') can be used for simplicity. So 10.3.10.*, for instance, refers to IPs between 10.3.10.0 and 10.3.10.255. Interface 10.3.10.4-18 refers to IPs between 10.3.10.4 and 10.3.10.18 (4 and 18 included). If network interface configuration is enabled (disabled by default) and if Hazelcast cannot find an matching interface, then it will print a message on console and won't start on that node.

```

<hazelcast>
  ...
  <network>
    ...
    <interfaces enabled="true">
      <interface>10.3.16.*</interface>
      <interface>10.3.10.4-18</interface>
      <interface>192.168.1.3</interface>
    </interfaces>
  </network>
  ...
</hazelcast>

```

16.1.3 EC2 Auto Discovery

Hazelcast supports EC2 Auto Discovery. It is useful when you do not want or cannot provide the list of possible IP addresses. To configure your cluster to be able to use EC2 Auto Discovery, disable join over multicast and TCP/IP and enable AWS. Also provide your credentials (access and secret keys). The `aws` tag accepts an attribute called *connection-timeout-seconds* whose default value is 5. Increasing this value is recommended if you have many IPs listed and members cannot properly build up the cluster.

Below is a sample configuration.

```

<join>
  <multicast enabled="false">
    <multicast-group>224.2.2.3</multicast-group>
    <multicast-port>54327</multicast-port>
  </multicast>
  <tcp-ip enabled="false">
    <interface>192.168.1.2</interface>
  </tcp-ip>
</join>

```

```

</tcp-ip>
<aws enabled="true">
  <access-key>my-access-key</access-key>
  <secret-key>my-secret-key</secret-key>
  <region>us-west-1</region>                                <!-- optional, default is us-east-1 -->
  <host-header>ec2.amazonaws.com</host-header>              <!-- optional, default is ec2.amazonaws.com -->
                                                                If set, region shouldn't be set as it will override this
  <security-group-name>hazelcast-sg</security-group-name>    <!-- optional -->
  <tag-key>type</tag-key>                                    <!-- optional -->
  <tag-value>hz-nodes</tag-value>                            <!-- optional -->
</aws>
</join>

```

You need to add *hazelcast-cloud.jar* dependency into your project. Note that it is also bundled inside *hazelcast-all.jar*. Hazelcast cloud module does not depend on any other third party modules.

Related Information

You can download the white paper “Hazelcast on AWS: Best Practices for Deployment”* from [Hazelcast.com](https://www.hazelcast.com/whitepapers/hazelcast-on-aws/).*

16.1.4 IPv6 Support

Hazelcast supports IPv6 addresses seamlessly (This support is switched off by default, please see the note at the end of this section).

All you need is to define IPv6 addresses or interfaces in **network configuration**. Only limitation at the moment is that you cannot define wildcard IPv6 addresses in TCP-IP join configuration. **Interfaces** section does not have this limitation, you can configure wildcard IPv6 interfaces same as IPv4 interfaces.

```

<hazelcast>
...
<network>
  <port auto-increment="true">5701</port>
  <join>
    <multicast enabled="false">
      <multicast-group>FF02:0:0:0:0:0:0:1</multicast-group>
      <multicast-port>54327</multicast-port>
    </multicast>
    <tcp-ip enabled="true">
      <member>[fe80::223:6cff:fe93:7c7e]:5701</member>
      <interface>192.168.1.0-7</interface>
      <interface>192.168.1.*</interface>
      <interface>fe80:0:0:0:45c5:47ee:fe15:493a</interface>
    </tcp-ip>
  </join>
  <interfaces enabled="true">
    <interface>10.3.16.*</interface>
    <interface>10.3.10.4-18</interface>
    <interface>fe80:0:0:0:45c5:47ee:fe15:*</interface>
    <interface>fe80::223:6cff:fe93:0-5555</interface>
  </interfaces>
  ...
</network>
...
</hazelcast>

```

JVM has two system properties for setting the preferred protocol stack (IPv4 or IPv6) as well as the preferred address family types (inet4 or inet6). On a dual stack machine, IPv6 stack is preferred by default, this can be

changed through `java.net.preferIPv4Stack=<true|false>` system property. And when querying name services, JVM prefers IPv4 addressed over IPv6 addresses and will return an IPv4 address if possible. This can be changed through `java.net.preferIPv6Addresses=<true|false>` system property.

Also see additional [details on IPv6 support in Java](#).

Note: *IPv6 support has been switched off by default, since some platforms have issues in use of IPv6 stack. Some other platforms such as Amazon AWS have no support at all. To enable IPv6 support, just set configuration property `hazelcast.prefer.ipv4.stack` to false. See [Advanced Configuration Properties](#).*

16.1.5 Restricting Outbound Ports

By default, Hazelcast lets the system to pick up an ephemeral port during socket bind operation. But security policies/firewalls may require to restrict outbound ports to be used by Hazelcast enabled applications. To fulfill this requirement, you can configure Hazelcast to use only defined outbound ports.

```
<hazelcast>
...
<network>
  <port auto-increment="true">5701</port>
  <outbound-ports>
    <ports>33000-35000</ports>    <!-- ports between 33000 and 35000 -->
    <ports>37000,37001,37002,37003</ports> <!-- comma separated ports -->
    <ports>38000,38500-38600</ports>
  </outbound-ports>
  ...
</network>
...
</hazelcast>

...
NetworkConfig networkConfig = config.getNetworkConfig();
networkConfig.addOutboundPortDefinition("35000-35100");           // ports between 35000 and 35100
networkConfig.addOutboundPortDefinition("36001, 36002, 36003"); // comma separated ports
networkConfig.addOutboundPort(37000);
networkConfig.addOutboundPort(37001);
...
```

Note: *You can use port ranges and/or comma separated ports.*

16.2 Partition Group Configuration

Hazelcast distributes key objects into partitions (blocks) using a consistent hashing algorithm and those partitions are assigned to nodes. That means an entry is stored in a node which is owner of partition to which entry's key is assigned. Total partition count is 271 by default and can be changed with configuration property `hazelcast.map.partition.count`. Please see [Advanced Configuration Properties](#).

Along with those partitions, there are also copies of them as backups. Backup partitions can have multiple copies due to backup count defined in configuration, such as first backup partition, second backup partition, etc. As a rule, a node can not hold more than one copy of a partition (ownership or backup). By default Hazelcast distributes partitions and their backup copies randomly and equally among cluster nodes assuming all nodes in the cluster are identical.

Now; What if some nodes share same JVM or physical machine or chassis and you want backups of these nodes to be assigned to nodes in another machine or chassis? What if processing or memory capacities of some nodes are different and you do not want equal number of partitions to be assigned to all nodes?

You can group nodes in the same JVM (or physical machine) or nodes located in the same chassis. Or, you can group nodes to create identical capacity. We call these groups **partition groups**. This way partitions are assigned to those partition groups instead of single nodes. And backups of these partitions are located in another partition group.

When you enable partition grouping, Hazelcast presents three choices to configure partition groups at the moment.

- First one is to group nodes automatically using IP addresses of nodes, so nodes sharing same network interface will be grouped together.

```
<partition-group enabled="true" group-type="HOST_AWARE" />
```

```
Config config = ...;
PartitionGroupConfig partitionGroupConfig = config.getPartitionGroupConfig();
partitionGroupConfig.setEnabled(true).setGroupType(MemberGroupType.HOST_AWARE);
```

- Second one is custom grouping using Hazelcast's interface matching configuration. This way, you can add different and multiple interfaces to a group. You can also use wildcards in interface addresses.

```
<partition-group enabled="true" group-type="CUSTOM">
  <member-group>
    <interface>10.10.0.*</interface>
    <interface>10.10.3.*</interface>
    <interface>10.10.5.*</interface>
  </member-group>
  <member-group>
    <interface>10.10.10.10-100</interface>
    <interface>10.10.1.*</interface>
    <interface>10.10.2.*</interface>
  </member-group>
</partition-group>
```

```
Config config = ...;
PartitionGroupConfig partitionGroupConfig = config.getPartitionGroupConfig();
partitionGroupConfig.setEnabled(true).setGroupType(MemberGroupType.CUSTOM);
```

```
MemberGroupConfig memberGroupConfig = new MemberGroupConfig();
memberGroupConfig.addInterface("10.10.0.*")
.addInterface("10.10.3.*").addInterface("10.10.5.*");
```

```
MemberGroupConfig memberGroupConfig2 = new MemberGroupConfig();
memberGroupConfig2.addInterface("10.10.10.10-100")
.addInterface("10.10.1.*").addInterface("10.10.2.*");
```

```
partitionGroupConfig.addMemberGroupConfig(memberGroupConfig);
partitionGroupConfig.addMemberGroupConfig(memberGroupConfig2);
```

- Third one is to give every member their own group. This gives the least amount of protection and is the default configuration for a Hazelcast cluster.

```
<partition-group enabled="true" group-type="PER_MEMBER" />
```

```
Config config = ...;
PartitionGroupConfig partitionGroupConfig = config.getPartitionGroupConfig();
partitionGroupConfig.setEnabled(true).setGroupType(MemberGroupType.PER_MEMBER);
```


16.3 Listener Configurations

Event listeners can be added to and removed from the related object using Hazelcast API.

Downside of attaching listeners using API is the possibility of missing events between creation of object and registering listener. To overcome this race condition, Hazelcast introduces registration of listeners in configuration. Listeners can be registered using either declarative, programmatic or Spring configuration.

• MembershipListener

- Declarative Configuration

```
<listeners>
  <listener>com.hazelcast.examples.MembershipListener</listener>
</listeners>
```

- Programmatic Configuration

```
config.addListenerConfig(new ListenerConfig("com.hazelcast.examples.MembershipListener"));
```

- Spring XML configuration

```
<hz:listeners>
  <hz:listener class-name="com.hazelcast.spring.DummyMembershipListener"/>
  <hz:listener implementation="dummyMembershipListener"/>
</hz:listeners>
```

• DistributedObjectListener

- Declarative Configuration

```
<listeners>
  <listener>com.hazelcast.examples.DistributedObjectListener</listener>
</listeners>
```

- Programmatic Configuration

```
config.addListenerConfig(new ListenerConfig("com.hazelcast.examples.DistributedObjectListener"));
```

- Spring XML configuration

```
<hz:listeners>
  <hz:listener class-name="com.hazelcast.spring.DummyDistributedObjectListener"/>
  <hz:listener implementation="dummyDistributedObjectListener"/>
</hz:listeners>
```

• MigrationListener

- Declarative Configuration

```
<listeners>
  <listener>com.hazelcast.examples.MigrationListener</listener>
</listeners>
```

- Programmatic Configuration

```
config.addListenerConfig(new ListenerConfig("com.hazelcast.examples.MigrationListener"));
```

- Spring XML configuration

```
<hz:listeners>
  <hz:listener class-name="com.hazelcast.spring.DummyMigrationListener"/>
  <hz:listener implementation="dummyMigrationListener"/>
</hz:listeners>
```

• LifecycleListener

- Declarative Configuration

```

<listeners>
  <listener>com.hazelcast.examples.LifecycleListener</listener>
</listeners>

```

- Programmatic Configuration

```
config.addListenerConfig(new ListenerConfig("com.hazelcast.examples.LifecycleListener"));
```

- Spring XML configuration

```

<hz:listeners>
  <hz:listener class-name="com.hazelcast.spring.DummyLifecycleListener"/>
  <hz:listener implementation="dummyLifecycleListener"/>
</hz:listeners>

```

- **EntryListener** for IMap

- Declarative Configuration

```

<map name="default">
  ...
  <entry-listeners>
    <entry-listener include-value="true" local="false">com.hazelcast.examples.EntryListener</entry-listener>
  </entry-listeners>
</map>

```

- Programmatic Configuration

```
mapConfig.addEntryListenerConfig(new EntryListenerConfig("com.hazelcast.examples.EntryListener"));
```

- Spring XML configuration

```

<hz:map name="default">
  <hz:entry-listeners>
    <hz:entry-listener class-name="com.hazelcast.spring.DummyEntryListener" include-value="true" local="false"/>
    <hz:entry-listener implementation="dummyEntryListener" local="true"/>
  </hz:entry-listeners>
</hz:map>

```

- **EntryListener** for MultiMap

- Declarative Configuration

```

<multimap name="default">
  <value-collection-type>SET</value-collection-type>
  <entry-listeners>
    <entry-listener include-value="true" local="false">com.hazelcast.examples.EntryListener</entry-listener>
  </entry-listeners>
</multimap>

```

- Programmatic Configuration

```
multiMapConfig.addEntryListenerConfig(new EntryListenerConfig("com.hazelcast.examples.EntryListener"));
```

- Spring XML configuration

```

<hz:multimap name="default" value-collection-type="LIST">
  <hz:entry-listeners>
    <hz:entry-listener class-name="com.hazelcast.spring.DummyEntryListener" include-value="true" local="false"/>
    <hz:entry-listener implementation="dummyEntryListener" local="true"/>
  </hz:entry-listeners>
</hz:multimap>

```

- **ItemListener** for IQueue

- Declarative Configuration

```

<queue name="default">
    ...
    <item-listeners>
        <item-listener include-value="true">com.hazelcast.examples.ItemListener</item-listener>
    </item-listeners>
</queue>

```

- Programmatic Configuration

```
queueConfig.addItemListenerConfig(new ItemListenerConfig("com.hazelcast.examples.ItemListener",
```

- Spring XML configuration

```

<hz:queue name="default" >
    <hz:item-listeners>
        <hz:item-listener class-name="com.hazelcast.spring.DummyItemListener" include-value="tr
    </hz:item-listeners>
</hz:queue>

```

- **MessageListener** for ITopic

- Declarative Configuration

```

<topic name="default">
    <message-listeners>
        <message-listener>com.hazelcast.examples.MessageListener</message-listener>
    </message-listeners>
</topic>

```

- Programmatic Configuration

```
topicConfig.addMessageListenerConfig(new ListenerConfig("com.hazelcast.examples.MessageListener
```

- Spring XML configuration

```

<hz:topic name="default">
    <hz:message-listeners>
        <hz:message-listener class-name="com.hazelcast.spring.DummyMessageListener"/>
    </hz:message-listeners>
</hz:topic>

```

- **ClientListener**

- Declarative Configuration

```

<listeners>
    <listener>com.hazelcast.examples.ClientListener</listener>
</listeners>

```

- Programmatic Configuration

```
topicConfig.addMessageListenerConfig(new ListenerConfig("com.hazelcast.examples.ClientListener",
```

- Spring XML configuration

```

<hz:listeners>
    <hz:listener class-name="com.hazelcast.spring.DummyClientListener"/>
    <hz:listener implementation="dummyClientListener"/>
</hz:listeners>

```

16.4 Wildcard Configuration

Hazelcast supports wildcard configuration of Maps, Queues and Topics. Using an asterisk (*) character in the name, different instances of Maps, Queues and Topics can be configured by a single configuration.

Note that, with a limitation of a single usage, asterisk (*) can be placed anywhere inside the configuration name.

For instance a map named 'com.hazelcast.test.mymap' can be configured using one of these configurations;

```

<map name="com.hazelcast.test.*">
...
</map>

<map name="com.hazel*">
...
</map>

<map name="*.test.mymap">
...
</map>

<map name="com.*test.mymap">
...
</map>

```

Or a queue 'com.hazelcast.test.myqueue'

```

<queue name="*hazelcast.test.myqueue">
...
</queue>

<queue name="com.hazelcast.*.myqueue">
...
</queue>

```

16.5 Advanced Configuration Properties

There are some advanced configuration properties to tune some aspects of Hazelcast. These can be set as property name and value pairs through declarative configuration, programmatic configuration or JVM system property.

16.5.1 Declarative Configuration

```

<hazelcast xsi:schemaLocation="http://www.hazelcast.com/schema/config
http://www.hazelcast.com/schema/config/hazelcast-config-3.0.xsd"
xmlns="http://www.hazelcast.com/schema/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
....
  <properties>
    <property name="hazelcast.property.foo">value</property>
    ....
  </properties>
</hazelcast>

```

16.5.2 Programmatic Configuration

```

Config cfg = new Config() ;
cfg.setProperty("hazelcast.property.foo", "value");

```

16.5.3 System Property

1. Using JVM parameter: `java -Dhazelcast.property.foo=value`
2. Using System class: `System.setProperty("hazelcast.property.foo", "value");`

Below table lists the advanced configuration properties with their descriptions.

Property Name	Default Value	Type	Description
<code>hazelcast.memcache.enabled</code>	<code>true</code>	<code>bool</code>	Enable Memcache client request listener service.
<code>hazelcast.rest.enabled</code>	<code>true</code>	<code>bool</code>	Enable REST client request listener service.
<code>hazelcast.logging.type</code>	<code>jdk</code>	<code>enum</code>	Name of logging framework type to send logs.
<code>hazelcast.map.load.chunk.size</code>	<code>1000</code>	<code>int</code>	Chunk size for MapLoader 's map initialization.
<code>hazelcast.merge.first.run.delay.seconds</code>	<code>300</code>	<code>int</code>	Initial run delay of split brain/merge process.
<code>hazelcast.merge.next.run.delay.seconds</code>	<code>120</code>	<code>int</code>	Run interval of split brain/merge process.
<code>hazelcast.socket.bind.any</code>	<code>true</code>	<code>bool</code>	Bind both server-socket and client-sockets to any local interface.
<code>hazelcast.socket.server.bind.any</code>	<code>true</code>	<code>bool</code>	Bind server-socket to any local interface.
<code>hazelcast.socket.client.bind.any</code>	<code>true</code>	<code>bool</code>	Bind client-sockets to any local interface.
<code>hazelcast.socket.receive.buffer.size</code>	<code>32</code>	<code>int</code>	Socket receive buffer size in KB.
<code>hazelcast.socket.send.buffer.size</code>	<code>32</code>	<code>int</code>	Socket send buffer size in KB.
<code>hazelcast.socket.keep.alive</code>	<code>true</code>	<code>bool</code>	Socket set keep alive.
<code>hazelcast.socket.no.delay</code>	<code>true</code>	<code>bool</code>	Socket set TCP no delay.
<code>hazelcast.prefer.ipv4.stack</code>	<code>true</code>	<code>bool</code>	Prefer Ipv4 network interface when picking.
<code>hazelcast.shutdownhook.enabled</code>	<code>true</code>	<code>bool</code>	Enable Hazelcast shutdownhook thread.
<code>hazelcast.wait.seconds.before.join</code>	<code>5</code>	<code>int</code>	Wait time before join operation.
<code>hazelcast.max.wait.seconds.before.join</code>	<code>20</code>	<code>int</code>	Maximum wait time before join operation.
<code>hazelcast.heartbeat.interval.seconds</code>	<code>1</code>	<code>int</code>	Heartbeat send interval in seconds.
<code>hazelcast.max.no.heartbeat.seconds</code>	<code>300</code>	<code>int</code>	Max timeout of heartbeat in seconds for a node.
<code>hazelcast.icmp.enabled</code>	<code>false</code>	<code>bool</code>	Enable ICMP ping.
<code>hazelcast.icmp.timeout</code>	<code>1000</code>	<code>int</code>	ICMP timeout in ms.
<code>hazelcast.icmp.ttl</code>	<code>0</code>	<code>int</code>	ICMP TTL (maximum numbers of hops to reach destination).
<code>hazelcast.master.confirmation.interval.seconds</code>	<code>30</code>	<code>int</code>	Interval at which nodes send master confirmation.
<code>hazelcast.max.no.master.confirmation.seconds</code>	<code>450</code>	<code>int</code>	Max timeout of master confirmation from master.
<code>hazelcast.member.list.publish.interval.seconds</code>	<code>600</code>	<code>int</code>	Interval at which master node publishes a new member list.
<code>hazelcast.prefer.ipv4.stack</code>	<code>true</code>	<code>bool</code>	Prefer IPv4 Stack, don't use IPv6. See IP .
<code>hazelcast.initial.min.cluster.size</code>	<code>0</code>	<code>int</code>	Initial expected cluster size to wait before starting.
<code>hazelcast.initial.wait.seconds</code>	<code>0</code>	<code>int</code>	Initial time in seconds to wait before node starts.
<code>hazelcast.partition.count</code>	<code>271</code>	<code>int</code>	Total partition count.
<code>hazelcast.jmx</code>	<code>false</code>	<code>bool</code>	Enable JMX agent.
<code>hazelcast.jmx.detailed</code>	<code>false</code>	<code>bool</code>	Enable detailed views on JMX .
<code>hazelcast.mc.map.excludes</code>	<code>null</code>	<code>CSV</code>	Comma separated map names to exclude from management center.
<code>hazelcast.mc.queue.excludes</code>	<code>null</code>	<code>CSV</code>	Comma separated queue names to exclude from management center.
<code>hazelcast.mc.topic.excludes</code>	<code>null</code>	<code>CSV</code>	Comma separated topic names to exclude from management center.
<code>hazelcast.version.check.enabled</code>	<code>true</code>	<code>bool</code>	Enable Hazelcast new version check on startup.
<code>hazelcast.mc.max.visible.instance.count</code>	<code>100</code>	<code>int</code>	Management Center maximum visible instance count.
<code>hazelcast.connection.monitor.interval</code>	<code>100</code>	<code>int</code>	Minimum interval to consider a connection failed.
<code>hazelcast.connection.monitor.max.faults</code>	<code>3</code>	<code>int</code>	Maximum IO error count before disconnecting.
<code>hazelcast.partition.migration.interval</code>	<code>0</code>	<code>int</code>	Interval to run partition migration tasks in seconds.
<code>hazelcast.partition.migration.timeout</code>	<code>300</code>	<code>int</code>	Timeout for partition migration tasks in seconds.

Property Name	Default Value	Type	Description
<code>hazelcast.graceful.shutdown.max.wait</code>	600	int	Maximum wait seconds during graceful shutdown.
<code>hazelcast.mc.url.change.enabled</code>	true	bool	Management Center changing server url is enabled.
<code>hazelcast.elastic.memory.enabled</code>	false	bool	Enable Hazelcast Elastic Memory off-heap.
<code>hazelcast.elastic.memory.total.size</code>	128	int	Hazelcast Elastic Memory storage total size in MB.
<code>hazelcast.elastic.memory.chunk.size</code>	1	int	Hazelcast Elastic Memory storage chunk size in MB.
<code>hazelcast.elastic.memory.shared.storage</code>	false	bool	Enable Hazelcast Elastic Memory shared storage.
<code>hazelcast.enterprise.license.key</code>	null	string	Hazelcast Enterprise license key.
<code>hazelcast.system.log.enabled</code>	true	bool	Enable system logs.

16.6 Logging Configuration

Hazelcast has a flexible logging configuration and does not depend on any logging framework except JDK logging. It has in-built adaptors for a number of logging frameworks and also supports custom loggers by providing logging interfaces.

To use built-in adaptors, you should set `hazelcast.logging.type` property to one of predefined types below.

- **jdk**: JDK logging (default)
- **log4j**: Log4j
- **slf4j**: Slf4j
- **none**: disable logging

You can set `hazelcast.logging.type` through declarative configuration, programmatic configuration or JVM system property.

- **Declarative Configuration**

```
<hazelcast xsi:schemaLocation="http://www.hazelcast.com/schema/config
http://www.hazelcast.com/schema/config/hazelcast-config-3.0.xsd"
xmlns="http://www.hazelcast.com/schema/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    ....

    <properties>
        <property name="hazelcast.logging.type">jdk</property>
        ....
    </properties>
</hazelcast>
```

- **Programmatic Configuration**

```
Config cfg = new Config() ;
cfg.setProperty("hazelcast.logging.type", "log4j");
```

- **System Property**

- Using JVM parameter: `java -Dhazelcast.logging.type=slf4j`

- Using System class: `System.setProperty("hazelcast.logging.type", "none");`

To use custom logging feature you should implement `com.hazelcast.logging.LoggerFactory` and `com.hazelcast.logging.ILogger` interfaces and set system property `hazelcast.logging.class` as your custom `LoggerFactory` class name.

```
-Dhazelcast.logging.class=foo.bar.MyLoggingFactory
```

You can also listen to logging events generated by Hazelcast runtime by registering `LogListeners` to `LoggingService`.

```
LogListener listener = new LogListener() {
    public void log(LogEvent logEvent) {
        // do something
    }
}
Config cfg = new Config();
HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);
LoggingService loggingService = instance.getLoggingService();
loggingService.addLogListener(Level.INFO, listener);
```

Through the `LoggingService`, you can get the current used `ILogger` implementation and log your own messages, too.

16.7 Setting License Key

Enterprise Only

To be able to use Hazelcast Enterprise, you need to set license the key in configuration.

- Declarative Configuration

```
<hazelcast>
...
<license-key>HAZELCAST_ENTERPRISE_LICENSE_KEY</license-key>
...
</hazelcast>
```

- Programmatic Configuration

```
Config config = new Config();
config.setLicenseKey("HAZELCAST_ENTERPRISE_LICENSE_KEY");
```

- Spring XML Configuration

```
<hz:config>
...
<hz:license-key>HAZELCAST_ENTERPRISE_LICENSE_KEY</hz:license-key>
...
</hazelcast>
```

- JVM System Property

```
-Dhazelcast.enterprise.license.key=HAZELCAST_ENTERPRISE_LICENSE_KEY
```


Chapter 17

Frequently Asked Questions

17.1 Why 271 as the default partition count

The partition count 271, being a prime number, is a good choice since it will be distributed to the nodes almost evenly. For a small to medium sized cluster, the count 271 gives almost even partition distribution and optimal sized partitions. As your cluster becomes bigger, this count should be made bigger to have evenly distributed partitions.

17.2 How do nodes discover each other

When a node is started in a cluster, it will dynamically and automatically be discovered. There are three types of discovery.

- One is the multicast. Nodes in a cluster discover each other by multicast, by default.
- Second is discovery by TCP/IP. The first node created in the cluster (leader) will form a list of IP addresses of other joining nodes and send this list to these nodes. So, nodes will know each other.
- And, if your application is placed on Amazon EC2, Hazelcast has an automatic discovery mechanism, as the third discovery type. You will just give your Amazon credentials and the joining node will be discovered automatically.

Once nodes are discovered, all the communication between them will be via TCP/IP.

17.3 What happens when a node goes down

Once a node is gone (e.g. crashes) and since data in each node has a backup in other nodes:

- First, the backups in other nodes are restored
- Then, data from these restored backups are recovered
- And finally, backups for these recovered data are formed

So, eventually, no data is lost.

17.4 How do I choose keys properly

When you store a key & value in a distributed Map, Hazelcast serializes the key and value, and stores the byte array version of them in local ConcurrentHashMaps. These ConcurrentHashMaps use `equals` and `hashCode` methods of

byte array version of your key. It does not take into account the actual `equals` and `hashCode` implementations of your objects. So it is important that you choose your keys in a proper way.

Implementing `equals` and `hashCode` is not enough, it is also important that the object is always serialized into the same byte array. All primitive types like `String`, `Long`, `Integer`, etc. are good candidates for keys to be used in Hazelcast. An unsorted `Set` is an example of a very bad candidate because Java Serialization may serialize the same unsorted set in two different byte arrays.

Note that the distributed `Set` and `List` store their entries as the keys in a distributed `Map`. So the notes above apply to the objects you store in `Set` and `List`.

17.5 How do I reflect value modifications

Hazelcast always return a clone copy of a value. Modifying the returned value does not change the actual value in the map (or multimap, list, set). You should put the modified value back to make changes visible to all nodes.

```
V value = map.get(key);
value.updateSomeProperty();
map.put(key, value);
```

Collections which return values of methods such as `IMap.keySet`, `IMap.values`, `IMap.entrySet`, `MultiMap.get`, `MultiMap.remove`, `IMap.keySet`, `IMap.values`, contain cloned values. These collections are NOT backup by related Hazelcast objects. So changes to the these are **NOT** reflected in the originals, and vice-versa.

17.6 How do I test my Hazelcast cluster

Hazelcast allows you to create more than one instance on the same JVM. Each member is called `HazelcastInstance` and each will have its own configuration, socket and threads, i.e. you can treat them as totally separate instances.

This enables us to write and run cluster unit tests on a single JVM. As you can use this feature for creating separate members different applications running on the same JVM (imagine running multiple web applications on the same JVM), you can also use this feature for testing Hazelcast cluster.

Let's say you want to test if two members have the same size of a map.

```
@Test
public void testTwoMemberMapSizes() {
    // start the first member
    HazelcastInstance h1 = Hazelcast.newHazelcastInstance(null);
    // get the map and put 1000 entries
    Map map1 = h1.getMap("testmap");
    for (int i = 0; i < 1000; i++) {
        map1.put(i, "value" + i);
    }
    // check the map size
    assertEquals(1000, map1.size());
    // start the second member
    HazelcastInstance h2 = Hazelcast.newHazelcastInstance(null);
    // get the same map from the second member
    Map map2 = h2.getMap("testmap");
    // check the size of map2
    assertEquals(1000, map2.size());
    // check the size of map1 again
    assertEquals(1000, map1.size());
}
```

In the test above, everything happens in the same thread. When developing multi-threaded test, coordination of the thread executions has to be carefully handled. Usage of `CountDownLatch` for thread coordination is highly recommended. You can certainly use other things. Here is an example where we need to listen for messages and make sure that we got these messages:

```
@Test
public void testTopic() {
    // start two member cluster
    HazelcastInstance h1 = Hazelcast.newHazelcastInstance(null);
    HazelcastInstance h2 = Hazelcast.newHazelcastInstance(null);
    String topicName = "TestMessages";
    // get a topic from the first member and add a messageListener
    ITopic<String> topic1 = h1.getTopic(topicName);
    final CountDownLatch latch1 = new CountDownLatch(1);
    topic1.addMessageListener(new MessageListener() {
        public void onMessage(Object msg) {
            assertEquals("Test1", msg);
            latch1.countDown();
        }
    });
    // get a topic from the second member and add a messageListener
    ITopic<String> topic2 = h2.getTopic(topicName);
    final CountDownLatch latch2 = new CountDownLatch(2);
    topic2.addMessageListener(new MessageListener() {
        public void onMessage(Object msg) {
            assertEquals("Test1", msg);
            latch2.countDown();
        }
    });
    // publish the first message, both should receive this
    topic1.publish("Test1");
    // shutdown the first member
    h1.shutdown();
    // publish the second message, second member's topic should receive this
    topic2.publish("Test1");
    try {
        // assert that the first member's topic got the message
        assertTrue(latch1.await(5, TimeUnit.SECONDS));
        // assert that the second members' topic got two messages
        assertTrue(latch2.await(5, TimeUnit.SECONDS));
    } catch (InterruptedException ignored) {}
}
```

You can surely start Hazelcast members with different configurations. Let's say we want to test if Hazelcast `LiteMember` can shutdown fine.

```
@Test(timeout = 60000)
public void shutdownLiteMember() {
    // first config for normal cluster member
    Config c1 = new XmlConfigBuilder().build();
    c1.setPortAutoIncrement(false);
    c1.setPort(5709);
    // second config for LiteMember
    Config c2 = new XmlConfigBuilder().build();
    c2.setPortAutoIncrement(false);
    c2.setPort(5710);
    // make sure to set LiteMember=true
```

```

c2.setLiteMember(true);
// start the normal member with c1
HazelcastInstance hNormal = Hazelcast.newHazelcastInstance(c1);
// start the LiteMember with different configuration c2
HazelcastInstance hLite = Hazelcast.newHazelcastInstance(c2);
hNormal.getMap("default").put("1", "first");
assert hLite.getMap("default").get("1").equals("first");
hNormal.shutdown();
hLite.shutdown();
}

```

Also remember to call `Hazelcast.shutdownAll()` after each test case to make sure that there is no other running member left from the previous tests.

```

@After
public void cleanup() throws Exception {
    Hazelcast.shutdownAll();
}

```

For more information please [check our existing tests](#).

17.7 How do I create separate clusters

By specifying group name and group password, you can separate your clusters in a simple way. Groupings can be by *dev*, *production*, *test*, *app*, etc.

```

<hazelcast>
  <group>
    <name>dev</name>
    <password>dev-pass</password>
  </group>
  ...
</hazelcast>

```

You can also set the `groupName` with programmatic configuration. JVM can host multiple Hazelcast instances. Each node can only participate in one group and it only joins to its own group, does not mess with others. Following code creates 3 separate Hazelcast nodes, `h1` belongs to `app1` cluster, while `h2` and `h3` belong to `app2` cluster.

```

Config configApp1 = new Config();
configApp1.getGroupConfig().setName("app1");

Config configApp2 = new Config();
configApp2.getGroupConfig().setName("app2");

HazelcastInstance h1 = Hazelcast.newHazelcastInstance(configApp1);
HazelcastInstance h2 = Hazelcast.newHazelcastInstance(configApp2);
HazelcastInstance h3 = Hazelcast.newHazelcastInstance(configApp2);

```

17.8 When `RuntimeException` is thrown

Most of the Hazelcast operations throw an `RuntimeException` (which is unchecked version of `InterruptedException`) if a user thread is interrupted while waiting a response. Hazelcast uses `RuntimeException` to pass `InterruptedException` up through interfaces that do not have `InterruptedException` in their signatures. The users should be able to catch and handle `RuntimeException` in such cases as if their threads are interrupted on a blocking operation.

17.9 When ConcurrentModificationException is thrown?

Some of Hazelcast operations can throw `ConcurrentModificationException` under transaction while trying to acquire a resource, although operation signatures do not define such an exception. Exception is thrown if resource cannot be acquired in a specific time. The users should be able to catch and handle `ConcurrentModificationException` while they are using Hazelcast transactions.

17.10 How is Split-Brain syndrome handled

Imagine that you have 10-node cluster and for some reason the network is divided into two in a way that 4 servers cannot see the other 6. As a result you ended up having two separate clusters; 4-node cluster and 6-node cluster. Members in each sub-cluster are thinking that the other nodes are dead even though they are not. This situation is called Network Partitioning (a.k.a. Split-Brain Syndrome).

Since it is a network failure, there is no way to avoid it programatically and your application will run as two separate independent clusters. But we should be able to answer the following questions: “What will happen after the network failure is fixed and connectivity is restored between these two clusters? Will these two clusters merge into one again? If they do, how are the data conflicts resolved, because you might end up having two different values for the same key in the same map?”

Here is how Hazelcast deals with it:

1. The oldest member of the cluster checks if there is another cluster with the same group-name and group-password in the network.
2. If the oldest member finds such cluster, then it figures out which cluster should merge to the other.
3. Each member of the merging cluster will do the following:
 - pause
 - take locally owned map entries
 - close all of its network connections (detach from its cluster)
 - join to the new cluster
 - send merge request for each of its locally owned map entry
 - resume

So each member of the merging cluster is actually rejoining to the new cluster and sending merge request for each of its locally owned map entry. Two important points:

- Smaller cluster will merge into the bigger one. If they have equal number of members then a hashing algorithm determines the merging cluster.
- Each cluster may have different versions of the same key in the same map. Destination cluster will decide how to handle merging entry based on the `MergePolicy` set for that map. There are built-in merge policies such as `PassThroughMergePolicy`, `PutIfAbsentMapMergePolicy`, `HigherHitsMapMergePolicy` and `LatestUpdateMapMergePolicy`. But you can develop your own merge policy by implementing `com.hazelcast.map.merge.MapMergePolicy`. You should set the full class name of your implementation to the merge-policy configuration.

```
public interface MergePolicy {
    /**
     * Returns the value of the entry after the merge
     * of entries with the same key. Returning value can be
     * null. You should consider the case where existingEntry is null.
     *
     * @param mapName      name of the map
     * @param mergingEntry entry merging into the destination cluster
     */
}
```

```

* @param existingEntry existing entry in the destination cluster
* @return final value of the entry. If returns null then entry will be removed.
*/
Object merge(String mapName, EntryView mergingEntry, EntryView existingEntry);
}

```

Here is how merge policies are specified per map:

```

<hazelcast>
...
<map name="default">
  <backup-count>1</backup-count>
  <eviction-policy>NONE</eviction-policy>
  <max-size>0</max-size>
  <eviction-percentage>25</eviction-percentage>
  <!--
    While recovering from split-brain (network partitioning),
    map entries in the small cluster will merge into the bigger cluster
    based on the policy set here. When an entry merge into the
    cluster, there might an existing entry with the same key already.
    Values of these entries might be different for that same key.
    Which value should be set for the key? Conflict is resolved by
    the policy set here. Default policy is hz.ADD_NEW_ENTRY

    There are built-in merge policies such as
    There are built-in merge policies such as
    com.hazelcast.map.merge.PassThroughMergePolicy; entry will be added if there is no existing
    com.hazelcast.map.merge.PutIfAbsentMapMergePolicy ; entry will be added if the merging entry
    com.hazelcast.map.merge.HigherHitsMapMergePolicy ; entry with the higher hits wins.
    com.hazelcast.map.merge.LatestUpdateMapMergePolicy ; entry with the latest update wins.
  -->
  <merge-policy>MY_MERGE_POLICY_CLASS</merge-policy>
</map>

...
</hazelcast>

```

17.11 Does Hazelcast support thousands of clients

Yes. However, there are some points to be considered. First of all, the environment should be LAN with a high stability and the network speed should be 10 Gbps or higher. If number of nodes are high, client type should be selected as Dummy (not Smart Client). In the case of Smart Clients, since each client will open a connection to the nodes, these nodes should be powerful enough (e.g. more cores) to handle hundreds or thousands of connections and client requests. Also, using near caches in clients should be considered to lower the network traffic. And finally, the Hazelcast releases with the NIO implementation should be used (which starts with 3.2).

Also, the clients should be configured attentively. Please refer to [Java Clients](#) section for configuration notes.

17.12 How do you give support

Support services are divided into two: community and commercial support. Community support is provided through our [Mail Group](#) and Stackoverflow web site. For information on support subscriptions, please see [Hazelcast.com](#).

17.13 Does Hazelcast persist

No. But, Hazelcast provides `MapStore` and `MapLoader` interfaces. When you implement, for example, `MapStore` interface, Hazelcast calls your store and load methods whenever needed.

17.14 Can I use Hazelcast in a single server

Yes. But, please note that, Hazelcast's main design focus is multi-node clusters to be used as a distribution platform.

17.15 How can I monitor Hazelcast

[Hazelcast Management Center](#) is used to monitor and manage the nodes running Hazelcast. In addition to monitoring overall state of a cluster, data structures can be analyzed and browsed in detail, map configurations can be updated and thread dump from nodes can be taken.

Moreover, JMX monitoring is also provided. Please see [Monitoring with JMX](#) section for details.

17.16 How can I see debug level logs

By changing the log level to “Debug”. Below sample lines are for `log4j` logging framework. Please see [Logging Configuration](#) to learn how to set logging types.

First, set the logging type as follows.

```
final String location = "log4j.configuration";
final String logging = "hazelcast.logging.type";
System.setProperty(logging, "log4j");
/**if you want to give a new location. */
System.setProperty(location, "file:/path/mylog4j.properties");
```

Then set the log level to “Debug” in properties file. Below is a sample content.

```
# direct log messages to stdout #
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p [%c{1}] - %m%n
log4j.logger.com.hazelcast=debug
#log4j.logger.com.hazelcast.cluster=debug
#log4j.logger.com.hazelcast.partition=debug
#log4j.logger.com.hazelcast.partition.InternalPartitionService=debug
#log4j.logger.com.hazelcast.nio=debug
#log4j.logger.com.hazelcast.hibernate=debug
```

The line `log4j.logger.com.hazelcast=debug` is used to see debug logs for all Hazelcast operations. Below this line, you can select to see specific logs (cluster, partition, hibernate, etc.).