Running {brandname} 10.0 Servers

Table of Contents

1. About the {brandname} Server . . . 1
   1.1. Getting Started . . . 1
2. Operating modes . . . 2
   2.1. Standalone mode . . . 2
   2.2. Domain mode . . . 2
      2.2.1. Host . . . 3
      2.2.2. Domain Controller . . . 4
      2.2.3. Server Group . . . 4
      2.2.4. Server . . . 5
   2.3. Example configurations . . . 5
3. Configuration . . . 6
   3.1. JGroups subsystem configuration . . . 6
      3.1.1. Cluster authentication and authorization . . . 9
   3.2. {brandname} subsystem configuration . . . 10
      3.2.1. Containers . . . 10
      3.2.2. Caches . . . 11
      3.2.3. Expiration . . . 11
      3.2.4. Eviction . . . 12
      3.2.5. Locking . . . 12
      3.2.6. Transactional Operations with Hot Rod . . . 12
      3.2.7. Loaders and Stores . . . 13
      3.2.8. State Transfer . . . 14
   3.3. Endpoint subsystem configuration . . . 14
      3.3.1. Hot Rod . . . 14
      3.3.2. Memcached . . . 15
      3.3.3. WebSocket . . . 15
      3.3.4. REST . . . 15
      3.3.5. Common Protocol Connector Settings . . . 15
      3.3.6. Protocol Interoperability . . . 16
      3.3.7. Custom Marshaller Bridges . . . 16
4. Performing Rolling Upgrades . . . 19
   4.1. Setting Up a Target Cluster . . . 19
   4.2. Synchronizing Data from the Source Cluster . . . 20
5. Client/Server . . . 22
   5.1. Why Client/Server? . . . 22
   5.2. Why use embedded mode? . . . 26
   5.3. Server Modules . . . 26
   5.4. Which protocol should I use? . . . 27

6. Using Hot Rod Server . . . 28
6.1. Java Hot Rod client . . . 28
6.1.1. Configuration . . . 28
6.1.2. Authentication . . . 29
6.1.3. Encryption . . . 33
6.1.4. Basic API . . . 36
6.1.5. RemoteCache(.keySet|.entrySet|.values) . . . 36
6.1.6. Remote Iterator . . . 37
6.1.7. Versioned API . . . 39
6.1.8. Streaming API . . . 40
6.1.9. Creating Event Listeners . . . 41
6.1.10. Removing Event Listeners . . . 43
6.1.11. Filtering Events . . . 43
6.1.12. Customizing Events . . . 46
6.1.13. Filter and Custom Events . . . 49
6.1.14. Event Marshalling . . . 51
6.1.15. Listener State Handling . . . 52
6.1.16. Listener Failure Handling . . . 52
6.1.17. Near Caching . . . 53
6.1.18. Unsupported methods . . . 54
6.1.19. Return values . . . 55
7. Hot Rod Transactions . . . 56
7.1. Configuring the Server . . . 56
7.2. Configuring Hot Rod Clients . . . 56
7.2.1. TransactionManagerLookup Interface . . . 57
7.2.2. Transaction Modes . . . 57
7.3. Overriding Configuration for Cache Instances . . . 58
7.4. Detecting Conflicts with Transactions . . . 58
7.5. Using the Configured Transaction Manager and Transaction Mode . . . 60
7.6. Overriding the Transaction Manager . . . 61
7.7. Overriding the Transaction Mode . . . 62
7.7.1. Client Intelligence . . . 62
7.7.2. Request Balancing . . . 63
7.7.3. Persistent connections . . . 63
7.7.4. Marshalling data . . . 63
7.7.5. Reading data in different data formats . . . 64
7.7.6. Statistics . . . 65
7.7.7. Multi-Get Operations . . . 65
7.7.8. Failover capabilities . . . 65
7.7.9. Site Cluster Failover . . . 66
7.7.10. Manual Site Cluster Switch . . . 66

7.7.11. Monitoring the Hot Rod client . . . 67
7.7.12. Concurrent Updates . . . 67
7.7.13. Javadocs . . . 70
8. REST Server . . . 71
8.1. Running the REST server . . . 71
8.1.1. Security . . . 71
8.2. Supported protocols . . . 71
8.3. CORS . . . 71
8.4. Data formats . . . 73
8.4.1. Configuration . . . 73
8.4.2. Supported formats . . . 73
8.4.3. Accept header . . . 74
8.4.4. Key-Content-Type header . . . 74
8.4.5. JSON/Protostream conversion . . . 75
8.5. REST V1 API . . . 76
8.5.1. Putting data in . . . 76
8.5.2. Getting data back out . . . 76
8.5.3. Listing keys . . . 77
8.5.4. Removing data . . . 78
8.5.5. Querying . . . 78
8.6. REST v2 API . . . 79
8.6.1. Working with Caches . . . 79
8.6.2. Monitoring {brandname} Clusters . . . 86
8.6.3. Counter . . . 87
8.7. Client-Side Code . . . 90
8.7.1. Ruby example . . . 90
8.7.2. Python 3 example . . . 92
8.7.3. Java example . . . 93
9. Memcached Server . . . 96
9.1. Client Encoding . . . 96
9.2. Command Clarifications . . . 96
9.2.1. Flush All . . . 96
9.3. Unsupported Features . . . 96
9.3.1. Individual Stats . . . 97
9.3.2. Statistic Settings . . . 97
9.3.3. Settings with Arguments Parameter . . . 97
9.3.4. Delete Hold Time Parameter . . . 97
9.3.5. Verbosity Command . . . 97
9.4. Talking To {brandname} Memcached Servers From Non-Java Clients . . . 98
9.4.1. Multi Clustered Server Tutorial . . . 98
10. Scripting . . . 100

10.1. Installing scripts . . . 100
10.2. Script metadata . . . 100
10.2.1. Metadata properties . . . 100
10.3. Script bindings . . . 101
10.4. Script parameters . . . 101
10.5. Running Scripts using the Hot Rod Java client . . . 102
10.6. Distributed execution . . . 102
11. Server Tasks . . . 103
11.1. Implementing Server Tasks . . . 103
12. Health monitoring . . . 106
12.1. Accessing Health API using JMX . . . 106
12.2. Accessing Health API using CLI . . . 106
12.3. Accessing Health API using REST . . . 107
13. Multi-tenancy . . . 110
13.1. Using REST interface . . . 110
13.2. Using Hot Rod client . . . 110
13.2.1. Multi-tenant router . . . 111
14. Single-Port . . . 113
14.1. Single-Port router . . . 113
14.1.1. Testing the Single-Port router . . . 113
14.2. Hot Rod . . . 114
14.2.1. TLS/ALPN protocol selection . . . 114

Chapter 1. About the {brandname} Server

{brandname} Server is a standalone server which exposes any number of caches to clients over a variety of protocols, including HotRod, Memcached and REST.

The server itself is built on top of the robust foundation provided by WildFly, therefore delegating services such as management, configuration, datasources, transactions, logging, security to the respective subsystems.

Because {brandname} Server is closely tied to the latest releases of {brandname} and JGroups, the subsystems which control these components are different, in that they introduce new features and change some existing ones (e.g. cross-site replication, etc).

For this reason, the configuration of these subsystems should use the {brandname} Server-specific schema, although for most use-cases the configuration is interchangeable. See the Configuration section for more information.

1.1. Getting Started

To get started using the server, download the {brandname} Server distribution, unpack it to a local directory and launch it using the bin/standalone.sh or bin/standalone.bat scripts depending on your platform. This will start a single-node server using the standalone/configuration/standalone.xml configuration file, with four endpoints, one for each of the supported protocols. These endpoints allow access to all of the caches configured in the {brandname} subsystem (apart from the Memcached endpoint which, because of the protocol's design, only allows access to a single cache).

Chapter 2. Operating modes

{brandname} Server, like WildFly, can be booted in two different modes: standalone and domain.

2.1. Standalone mode

For simple configurations, standalone mode is the easiest to start with. It allows both local and clustered configurations, although we only really recommend it for running single nodes, since the configuration, management and coordination of multiple nodes is up to the user's responsibility. For example, adding a cache to a cluster of standalone server, the user would need to configure individually to all nodes. Note that the default standalone.xml configuration does not provide a JGroups subsystem and therefore cannot work in clustered mode. To start standalone mode with an alternative configuration file, use the -c command-line switch as follows:

```
bin/standalone.sh -c clustered.xml
```

If you start the server in clustered mode on multiple hosts, they should automatically discover each other using UDP multicast and form a cluster. If you want to start multiple nodes on a single host, start each one by specifying a port offset using the jboss.socket.binding.port-offset property together with a unique jboss.node.name as follows:

```
bin/standalone.sh -Djboss.socket.binding.port-offset=100 -Djboss.node.name=nodeA
```

If, for some reason, you cannot use UDP multicast, you can use TCP discovery. Read the **JGroups Subsystem Configuration** section below for details on how to configure TCP discovery.

2.2. Domain mode

Domain mode is the recommended way to run a cluster of servers, since they can all be managed centrally from a single control point. The following diagram explains the topology of an example domain configuration, with 4 server nodes (A1, A2, B1, B2) running on two physical hosts (A, B):

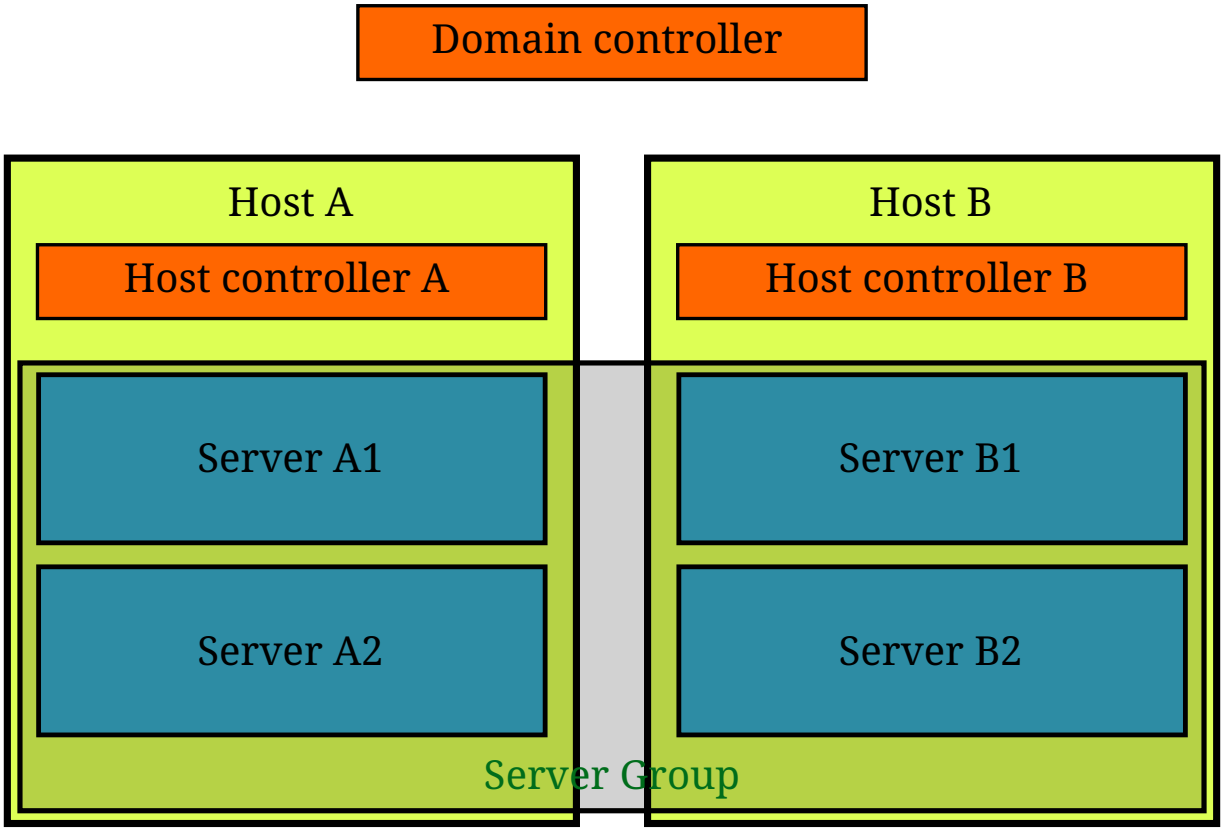


Figure 1. Domain-mode

2.2.1. Host

Each "Host" box in the above diagram represents a physical or virtual host. A physical host can contain zero, one or more server instances.

Host Controller

When the domain.sh or domain.bat script is run on a host, a process known as a Host Controller is launched. The Host Controller is solely concerned with server management; it does not itself handle {brandname} server workloads. The Host Controller is responsible for starting and stopping the individual {brandname} server processes that run on its host, and interacts with the Domain Controller to help manage them.

Each Host Controller by default reads its configuration from the domain/configuration/host.xml file located in the {brandname} Server installation on its host's filesystem. The host.xml file contains configuration information that is specific to the particular host. Primarily:

- the listing of the names of the actual {brandname} Server instances that are meant to run off of this installation.
- configuration of how the Host Controller is to contact the Domain Controller to register itself and access the domain configuration. This may either be configuration of how to find and contact a remote Domain Controller, or a configuration telling the Host Controller to itself act as the Domain Controller.
- configuration of items that are specific to the local physical installation. For example, named interface definitions declared in domain.xml (see below) can be mapped to an actual machine-specific IP address in host.xml. Abstract path names in domain.xml can be mapped to actual

filesystem paths in host.xml.

2.2.2. Domain Controller

One Host Controller instance is configured to act as the central management point for the entire domain, i.e. to be the Domain Controller. The primary responsibility of the Domain Controller is to maintain the domain's central management policy, to ensure all Host Controllers are aware of its current contents, and to assist the Host Controllers in ensuring any running {brandname} server instances are configured in accordance with this policy. This central management policy is stored by default in the domain/configuration/domain.xml file in the {brandname} Server installation on Domain Controller's host's filesystem.

A domain.xml file must be located in the domain/configuration directory of an installation that's meant to run the Domain Controller. It does not need to be present in installations that are not meant to run a Domain Controller; i.e. those whose Host Controller is configured to contact a remote Domain Controller. The presence of a domain.xml file on such a server does no harm.

The domain.xml file includes, among other things, the configuration of the various "profiles" that {brandname} Server instances in the domain can be configured to run. A profile configuration includes the detailed configuration of the various subsystems that comprise that profile (e.g. Cache Containers and Caches, Endpoints, Security Realms, DataSources, etc). The domain configuration also includes the definition of groups of sockets that those subsystems may open. The domain configuration also includes the definition of "server groups".

2.2.3. Server Group

A server group is set of server instances that will be managed and configured as one. In a managed domain each application server instance is a member of a server group. Even if the group only has a single server, the server is still a member of a group. It is the responsibility of the Domain Controller and the Host Controllers to ensure that all servers in a server group have a consistent configuration. They should all be configured with the same profile and they should have the same deployment content deployed. To keep things simple, ensure that all the nodes that you want to belong to an {brandname} cluster are configured as servers of one server group.

The domain can have multiple server groups, i.e. multiple {brandname} clusters. Different server groups can be configured with different profiles and deployments; for example in a domain with different {brandname} Server clusters providing different services. Different server groups can also run the same profile and have the same deployments.

An example server group definition is as follows:

```
<server-group name="main-server-group" profile="clustered">
    <socket-binding-group ref="standard-sockets"/>
</server-group>
```

A server-group configuration includes the following required attributes:

- name — the name of the server group

- profile — the name of the profile the servers in the group should run

In addition, the following optional elements are available:

- socket-binding-group — specifies the name of the default socket binding group to use on servers in the group. Can be overridden on a per-server basis in host.xml. If not provided in the server-group element, it must be provided for each server in host.xml.
- deployments — the deployment content that should be deployed on the servers in the group.
- system-properties — system properties that should be set on all servers in the group
- jvm — default jvm settings for all servers in the group. The Host Controller will merge these settings with any provided in host.xml to derive the settings to use to launch the server's JVM. See JVM settings for further details.

2.2.4. Server

Each "Server" in the above diagram represents an actual {brandname} Server node. The server runs in a separate JVM process from the Host Controller. The Host Controller is responsible for launching that process. In a managed domain the end user cannot directly launch a server process from the command line.

The Host Controller synthesizes the server's configuration by combining elements from the domain wide configuration (from domain.xml) and the host-specific configuration (from host.xml).

2.3. Example configurations

The server distribution also provides a set of example configuration files in the docs/examples/configs (mostly using standalone mode) which illustrate a variety of possible configurations and use-cases. To use them, just copy them to the standalone/configuration directory and start the server using the following syntax:

```
bin/standalone.sh -c configuration_file_name.xml
```

For more information regarding the parameters supported by the startup scripts, refer to the WildFly documentation on Command line parameters.

Chapter 3. Configuration

Since the server is based on the WildFly codebase, refer to the WildFly documentation, apart from the JGroups, {brandname} and Endpoint subsytems.

3.1. JGroups subsystem configuration

The JGroups subsystem configures the network transport and is only required when clustering multiple {brandname} Server nodes together.

The subsystem declaration is enclosed in the following XML element:

```
<subsystem xmlns="urn:infinispan:server:jgroups:9.2">
  <channels default="cluster">
    <channel name="cluster"/>
  </channels>
  <stacks default="${jboss.default.jgroups.stack:udp}">
    ...
  </stacks>
</subsystem>
```

Within the subsystem, you need to declare the stacks that you wish to use and name them. The default-stack attribute in the subsystem declaration must point to one of the declared stacks. You can switch stacks from the command-line using the jboss.default.jgroups.stack property:

```
bin/standalone.sh -c clustered.xml -Djboss.default.jgroups.stack=tcp
```

A stack declaration is composed of a transport, `UDP` or `TCP`, followed by a list of protocols. You can tune protocols by adding properties as child elements with this format:

```
<property name="prop_name">prop_value</property>
```

Default stacks for {brandname} are as follows:

```
<stack name="udp">
  <transport type="UDP" socket-binding="jgroups-udp"/>
  <protocol type="PING"/>
  <protocol type="MERGE3"/>
  <protocol type="FD_SOCK" socket-binding="jgroups-udp-fd"/>
  <protocol type="FD_ALL"/>
  <protocol type="VERIFY_SUSPECT"/>
  <protocol type="pbcast.NAKACK2"/>
  <protocol type="UNICAST3"/>
  <protocol type="pbcast.STABLE"/>
  <protocol type="pbcast.GMS"/>
  <protocol type="UFC_NB"/>
  <protocol type="MFC_NB"/>
  <protocol type="FRAG3"/>
</stack>
<stack name="tcp">
  <transport type="TCP" socket-binding="jgroups-tcp"/>
  <protocol type="MPING" socket-binding="jgroups-mping"/>
  <protocol type="MERGE3"/>
  <protocol type="FD_SOCK" socket-binding="jgroups-tcp-fd"/>
  <protocol type="FD_ALL"/>
  <protocol type="VERIFY_SUSPECT"/>
  <protocol type="pbcast.NAKACK2">
    <property name="use_mcast_xmit">false</property>
  </protocol>
  <protocol type="UNICAST3"/>
  <protocol type="pbcast.STABLE"/>
  <protocol type="pbcast.GMS"/>
  <protocol type="MFC_NB"/>
  <protocol type="FRAG3"/>
</stack>
```

For some properties, {brandname} uses values other than the JGroups defaults to tune performance. You should examine the following files to review the JGroups configuration for {brandname}:

- Remote Client/Server Mode:
  - `jgroups-defaults.xml`
  - `infinispan-jgroups.xml`
- Library Mode:
  - `default-jgroups-tcp.xml`
  - `default-jgroups-udp.xml`

See JGroups Protocol documentation for more information about available properties and default values.

The default TCP stack uses the MPING protocol for discovery, which uses UDP multicast. If you need to use a different protocol, look at the JGroups Discovery Protocols . The following example stack

configures the TCPPING discovery protocol with two initial hosts:

```
<stack name="tcp">
  <transport type="TCP" socket-binding="jgroups-tcp"/>
  <protocol type="TCPPING">
    <property name="initial_hosts">HostA[7800],HostB[7800]</property>
  </protocol>
  <protocol type="MERGE3"/>
  <protocol type="FD_SOCK" socket-binding="jgroups-tcp-fd"/>
  <protocol type="FD_ALL"/>
  <protocol type="VERIFY_SUSPECT"/>
  <protocol type="pbcast.NAKACK2">
    <property name="use_mcast_xmit">false</property>
  </protocol>
  <protocol type="UNICAST3"/>
  <protocol type="pbcast.STABLE"/>
  <protocol type="pbcast.GMS"/>
  <protocol type="UFC_NB"/>
  <protocol type="MFC_NB"/>
  <protocol type="FRAG3"/>
</stack>
```

The default configurations come with a variety of pre-configured stacks for different enviroments. For example, the tcpgossip stack uses Gossip discovery:

```
<protocol type="TCPGOSSIP">
  <property name="initial_hosts">${jgroups.gossip.initial_hosts:}</property>
</protocol>
```

Use the s3 stack when running in Amazon AWS:

```
<protocol type="S3_PING">
  <property name="location">${jgroups.s3.bucket:}</property>
  <property name="access_key">${jgroups.s3.access_key:}</property>
  <property name="secret_access_key">${jgroups.s3.secret_access_key:}</property>
  <property name="pre_signed_delete_url">
${jgroups.s3.pre_signed_delete_url:}</property>
  <property name="pre_signed_put_url">${jgroups.s3.pre_signed_put_url:}</property>
  <property name="prefix">${jgroups.s3.prefix:}</property>
</protocol>
```

Similarly, when using Google's Cloud Platform, use the google stack:

```
<protocol type="GOOGLE_PING">
  <property name="location">${jgroups.google.bucket:}</property>
  <property name="access_key">${jgroups.google.access_key:}</property>
  <property name="secret_access_key">${jgroups.google.secret_access_key:}</property>
</protocol>
```

Use the dns-ping stack to run {brandname} on Kubernetes environments such as OKD or OpenShift:

```
<protocol type="dns.DNS_PING">
    <property name="dns_query">${jgroups.dns_ping.dns_query}</property>
</protocol>
```

The value of the `dns_query` property is the DNS query that returns the cluster members. See DNS for Services and Pods for information about Kubernetes DNS naming.

3.1.1. Cluster authentication and authorization

The JGroups subsystem can be configured so that nodes need to authenticate each other when joining / merging. The authentication uses SASL and integrates with the security realms.

```
<management>
  <security-realms>
  ...
    <security-realm name="ClusterRealm">
      <authentication>
        <properties path="cluster-users.properties" relative-to=
"jboss.server.config.dir"/>
      </authentication>
      <authorization>
        <properties path="cluster-roles.properties" relative-to=
"jboss.server.config.dir"/>
      </authorization>
    </security-realm>
  ...
  </security-realms>
</management>

<stack name="udp">
  ...
  <sasl mech="DIGEST-MD5" security-realm="ClusterRealm" cluster-role="cluster">
    <property name="client_name">node1</property>
    <property name="client_password">password</property>
  </sasl>
  ...
</stack>
```

In the above example the nodes will use the DIGEST-MD5 mech to authenticate against the

ClusterRealm. In order to join, nodes need to have the cluster role. If the cluster-role attribute is not specified it defaults to the name of the {brandname} cache-container, as described below. Each node identifies itself using the client_name property. If none is explicitly specified, the hostname on which the server is running will be used. This name can also be overridden by specifying the jboss.node.name system property. The client_password property contains the password of the node. It is recommended that this password be stored in the Vault. Refer to AS7: Utilising masked passwords via the vault for instructions on how to do so. When using the GSSAPI mech, client_name will be used as the name of a Kerberos-enabled login module defined within the security domain subsystem:

```
<security-domain name="krb-node0" cache-type="default">
  <authentication>
    <login-module code="Kerberos" flag="required">
      <module-option name="storeKey" value="true"/>
      <module-option name="useKeyTab" value="true"/>
      <module-option name="refreshKrb5Config" value="true"/>
      <module-option name="principal" value="jgroups/node0/clustered@INFINISPAN.ORG"/>
      <module-option name="keyTab" value=
"${jboss.server.config.dir}/keytabs/jgroups_node0_clustered.keytab"/>
      <module-option name="doNotPrompt" value="true"/>
    </login-module>
  </authentication>
</security-domain>
```

3.2. {brandname} subsystem configuration

The {brandname} subsystem configures the cache containers and caches.

The subsystem declaration is enclosed in the following XML element:

```
<subsystem xmlns="urn:infinispan:server:core:9.4" default-cache-container="clustered">
  ...
</subsystem>
```

3.2.1. Containers

The {brandname} subsystem can declare multiple containers. A container is declared as follows:

```
<cache-container name="clustered" default-cache="default">
  ...
</cache-container>
```

Note that in server mode is the lack of an implicit default cache, but the ability to specify a named cache as the default.

If you need to declare clustered caches (distributed, replicated, invalidation), you also need to

specify the <transport/> element which references an existing JGroups transport. This is not needed if you only intend to have local caches only.

```
<transport executor="infinispan-transport" lock-timeout="60000" stack="udp" cluster=
"my-cluster-name"/>
```

3.2.2. Caches

Now you can declare your caches. Please be aware that only the caches declared in the configuration will be available to the endpoints and that attempting to access an undefined cache is an illegal operation. Contrast this with the default {brandname} library behaviour where obtaining an undefined cache will implicitly create one using the default settings. The following are example declarations for all four available types of caches:

```
<local-cache name="default" start="EAGER">
  ...
</local-cache>

<replicated-cache name="replcache" mode="SYNC" remote-timeout="30000" start="EAGER">
  ...
</replicated-cache>

<invalidation-cache name="invcache" mode="SYNC" remote-timeout="30000" start="EAGER">
  ...
</invalidation-cache>

<distributed-cache name="distcache" mode="SYNC" segments="20" owners="2" remote-
timeout="30000" start="EAGER">
  ...
</distributed-cache>
```

3.2.3. Expiration

To define a default expiration for entries in a cache, add the <expiration/> element as follows:

```
<expiration lifespan="2000" max-idle="1000"/>
```

The possible attributes for the expiration element are:

- *lifespan* maximum lifespan of a cache entry, after which the entry is expired cluster-wide, in milliseconds. -1 means the entries never expire.
- *max-idle* maximum idle time a cache entry will be maintained in the cache, in milliseconds. If the idle time is exceeded, the entry will be expired cluster-wide. -1 means the entries never expire.
- *interval* interval (in milliseconds) between subsequent runs to purge expired entries from

memory and any cache stores. If you wish to disable the periodic eviction process altogether, set interval to -1.

3.2.4. Eviction

To define eviction for a cache, add the `<memory/>` element as follows:

```
<memory> <binary size="1000" eviction="COUNT"/> </memory>
```

The possible attributes for the eviction element are:

- *strategy* sets the cache eviction strategy. Available options are 'UNORDERED', 'FIFO', 'LRU', 'LIRS' and 'NONE' (to disable eviction).
- *max-entries* maximum number of entries in a cache instance. If selected value is not a power of two the actual value will default to the least power of two larger than selected value. -1 means no limit.

3.2.5. Locking

To define the locking configuration for a cache, add the `<locking/>` element as follows:

```
<locking isolation="REPEATABLE_READ" acquire-timeout="30000" concurrency-level="1000"
striping="false"/>
```

The possible attributes for the locking element are:

- *isolation* sets the cache locking isolation level. Can be NONE, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE. Defaults to REPEATABLE_READ
- *striping* if true, a pool of shared locks is maintained for all entries that need to be locked. Otherwise, a lock is created per entry in the cache. Lock striping helps control memory footprint but may reduce concurrency in the system.
- *acquire-timeout* maximum time to attempt a particular lock acquisition.
- *concurrency-level* concurrency level for lock containers. Adjust this value according to the number of concurrent threads interacting with {brandname}.
- *concurrent-updates* for non-transactional caches only: if set to true(default value) the cache keeps data consistent in the case of concurrent updates. For clustered caches this comes at the cost of an additional RPC, so if you don't expect your application to write data concurrently, disabling this flag increases performance.

3.2.6. Transactional Operations with Hot Rod

Hot Rod clients can take advantage of transactional capabilities when performing cache operations. No other protocols that {brandname} supports offer transactional capabilities.

3.2.7. Loaders and Stores

Loaders and stores can be defined in server mode in almost the same way as in embedded mode. See Persistence in the User Guide.

However, in server mode it is no longer necessary to define the `<persistence>···</persistence>` tag. Instead, a store's attributes are now defined on the store type element. For example, to configure the H2 database with a distributed cache in domain mode we define the "default" cache as follows in our domain.xml configuration:

```
<subsystem xmlns="urn:infinispan:server:core:9.4">
  <cache-container name="clustered" default-cache="default" statistics="true">
    <transport lock-timeout="60000"/>
    <global-state/>
    <distributed-cache name="default">
      <string-keyed-jdbc-store datasource="java:jboss/datasources/ExampleDS" fetch-
state="true" shared="true">
        <string-keyed-table prefix="ISPN">
          <id-column name="id" type="VARCHAR"/>
          <data-column name="datum" type="BINARY"/>
          <timestamp-column name="version" type="BIGINT"/>
        </string-keyed-table>
        <write-behind modification-queue-size="20"/>
      </string-keyed-jdbc-store>
    </distributed-cache>
  </cache-container>
</subsystem>
```

Another important thing to note in this example, is that we use the "ExampleDS" datasource which is defined in the datasources subsystem in our domain.xml configuration as follows:

```
<subsystem xmlns="urn:jboss:domain:datasources:4.0">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS"
enabled="true" use-java-context="true">
      <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
      <driver>h2</driver>
      <security>
        <user-name>sa</user-name>
        <password>sa</password>
      </security>
    </datasource>
  </datasources>
</subsystem>
```

For additional examples of store configurations, please view the configuration templates in the default "domain.xml" file provided with in the server distribution at `./domain/configuration/domain.xml`.

3.2.8. State Transfer

To define the state transfer configuration for a distributed or replicated cache, add the `<state-transfer/>` element as follows:

```
<state-transfer enabled="true" timeout="240000" chunk-size="512" await-initial-
transfer="true" />
```

The possible attributes for the state-transfer element are:

- *enabled* if true, this will cause the cache to ask neighboring caches for state when it starts up, so the cache starts 'warm', although it will impact startup time. Defaults to true.
- *timeout* the maximum amount of time (ms) to wait for state from neighboring caches, before throwing an exception and aborting startup. Defaults to 240000 (4 minutes).
- *chunk-size* the number of cache entries to batch in each transfer. Defaults to 512.
- *await-initial-transfer* if true, this will cause the cache to wait for initial state transfer to complete before responding to requests. Defaults to true.

3.3. Endpoint subsystem configuration

The endpoint subsystem exposes a whole container (or in the case of Memcached, a single cache) over a specific connector protocol. You can define as many connector as you need, provided they bind on different interfaces/ports.

The subsystem declaration is enclosed in the following XML element:

```
<subsystem xmlns="urn:infinispan:server:endpoint:9.4">
 ...
</subsystem>
```

3.3.1. Hot Rod

The following connector declaration enables a HotRod server using the *hotrod* socket binding (declared within a `<socket-binding-group />` element) and exposing the caches declared in the *local* container, using defaults for all other settings.

```
<hotrod-connector socket-binding="hotrod" cache-container="local" />
```

The connector will create a supporting topology cache with default settings. If you wish to tune these settings add the `<topology-state-transfer />` child element to the connector as follows:

```
<hotrod-connector socket-binding="hotrod" cache-container="local">
   <topology-state-transfer lazy-retrieval="false" lock-timeout="1000" replication-
timeout="5000" />
</hotrod-connector>
```

The Hot Rod connector can be further tuned with additional settings such as concurrency and buffering. See the protocol connector settings paragraph for additional details

Furthermore the HotRod connector can be secured using SSL. First you need to declare an SSL server identity within a security realm in the management section of the configuration file. The SSL server identity should specify the path to a keystore and its secret. Refer to the AS documentation on this. Next add the `<security />` element to the HotRod connector as follows:

```
<hotrod-connector socket-binding="hotrod" cache-container="local">
    <security ssl="true" security-realm="ApplicationRealm" require-ssl-client-auth=
"false" />
</hotrod-connector>
```

3.3.2. Memcached

The following connector declaration enables a Memcached server using the *memcached* socket binding (declared within a `<socket-binding-group />` element) and exposing the *memcachedCache* cache declared in the *local* container, using defaults for all other settings. Because of limitations in the Memcached protocol, only one cache can be exposed by a connector. If you wish to expose more than one cache, declare additional memcached-connectors on different socket-bindings.

```
<memcached-connector socket-binding="memcached" cache-container="local"/>
```

3.3.3. WebSocket

```
<websocket-connector socket-binding="websocket" cache-container="local"/>
```

3.3.4. REST

```
<rest-connector socket-binding="rest" cache-container="local" security-domain="other"
auth-method="BASIC"/>
```

3.3.5. Common Protocol Connector Settings

The HotRod, Memcached and WebSocket protocol connectors support a number of tuning attributes in their declaration:

- *worker-threads* Sets the number of worker threads. Defaults to 160.

- *idle-timeout* Specifies the maximum time in seconds that connections from client will be kept open without activity. Defaults to -1 (connections will never timeout)
- *tcp-nodelay* Affects TCP NODELAY on the TCP stack. Defaults to enabled.
- *send-buffer-size* Sets the size of the send buffer.
- *receive-buffer-size* Sets the size of the receive buffer.

3.3.6. Protocol Interoperability

Clients exchange data with {brandname} through endpoints such as REST or Hot Rod.

Each endpoint uses a different protocol so that clients can read and write data in a suitable format. Because {brandname} can interoperate with multiple clients at the same time, it must convert data between client formats and the storage formats.

For more information, see Protocol Interoperability in the User Guide.

3.3.7. Custom Marshaller Bridges

{brandname} provides two marshalling bridges for marshalling client/server requests using the Kryo and Protostuff libraries. To utilise either of these marshallers, you simply place the dependency of the marshaller you require in your client pom. Custom schemas for object marshalling must then be registered with the selected library using the library's api on the client or by implementing a RegistryService for the given marshaller bridge. Examples of how to achieve this for both libraries are presented below:

Protostuff

Add the protostuff marshaller dependency to your pom:

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

Replace `${version.infinispan}` with the appropriate version of {brandname}.

To register custom Protostuff schemas in your own code, you must register the custom schema with Protostuff before any marshalling begins. This can be achieved by simply calling:

```
RuntimeSchema.register(ExampleObject.class, new ExampleObjectSchema());
```

Or, you can implement a service provider for the `SchemaRegistryService.java` interface, placing all Schema registrations in the `register()` method. Implementations of this interface are loaded via Java's ServiceLoader api, therefore the full path of the implementing class(es) should be provided in a `META-INF/services/org/infinispan/marshaller/protostuff/SchemaRegistryService` file within

your deployment jar.

Kryo

Add the kryo marshaller dependency to your pom:

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

Replace `${version.infinispan}` with the appropriate version of {brandname}.

To register custom Kryo serializer in your own code, you must register the custom serializer with Kryo before any marshalling begins. This can be achieved by implementing a service provider for the `SerializerRegistryService.java` interface, placing all serializer registrations in the `register(Kryo)` method; where serializers should be registered with the supplied `Kryo` object using the Kryo api. e.g. `kryo.register(ExampleObject.class, new ExampleObjectSerializer())`. Implementations of this interface are loaded via Java's ServiceLoader api, therefore the full path of the implementing class(es) should be provided in a `META-INF/services/org/infinispan/marshaller/kryo/SerializerRegistryService` file within your deployment jar.

Storing deserialized objects

When using the Protostuff/Kryo bridges in caches configured with *application/x-java-object* as MediaType (storing POJOs instead of binary content) it is necessary for the class files of all custom objects to be placed on the classpath of the server. To achieve this, you should place a jar containing all of their custom classes on the server's classpath.

When utilising a custom marshaller, it is also necessary for the marshaller and it's runtime dependencies to be on the server's classpath. To aid with this step we have created a "bundle" jar for each of the bridge implementations which includes all of the runtime class files required by the bridge and underlying library. Therefore, it is only necessary to include this single jar on the server's classpath.

Bundle jar downloads:

- Kryo Bundle
- Protostuff Bundle

Jar files containing custom classes must be placed in the same module/directory as the custom marshaller bundle so that the marshaller can load them. i.e. if you register the marshaller bundle in `modules/system/layers/base/org/infinispan/main/modules.xml`, then you must also register your custom classes here.

Registering Custom Schemas/Serializers

Custom serializers/schemas for the Kryo/Protostuff marshallers must be registered via their respective service interfaces in order to store deserialized objects. To achieve this, it is necessary for a **JAR** that contains the service provider to be registered in the same directory or module as the marshaller bundle and custom classes.

It is not necessary for the service provider implementation to be provided in the same **JAR** as the user's custom classes. However, the **JAR** that contains the provider must be in the same directory/module as the marshaller and custom class **JAR** files.

Chapter 4. Performing Rolling Upgrades

Upgrade {brandname} without downtime or data loss. You can perform rolling upgrades in Remote Client/Server Mode to start using a more recent version of {brandname}.

This section explains how to upgrade {brandname} servers, see the appropriate documentation for your Hot Rod client for upgrade procedures.

From a high-level, you do the following to perform rolling upgrades:

1. Set up a target cluster. The target cluster is the {brandname} version to which you want to migrate data. The source cluster is the {brandname} deployment that is currently in use. After the target cluster is running, you configure all clients to point to it instead of the source cluster.
2. Synchronize data from the source cluster to the target cluster.

4.1. Setting Up a Target Cluster

1. Start the target cluster with unique network properties or a different JGroups cluster name to keep it separate from the source cluster.
2. Configure a `RemoteCacheStore` on the target cluster for each cache you want to migrate from the source cluster.

   `RemoteCacheStore` **settings**

   - `remote-server` must point to the source cluster via the `outbound-socket-binding` property.
   - `remoteCacheName` must match the cache name on the source cluster.
   - `hotrod-wrapping` must be `true` (enabled).
   - `shared` must be `true` (enabled).
   - `purge` must be `false` (disabled).
   - `passivation` must be `false` (disabled).
   - `protocol-version` matches the Hot Rod protocol version of the source cluster.

Example `RemoteCacheStore` *Configuration*

```
<distributed-cache>
   <remote-store cache="MyCache" socket-timeout="60000" tcp-no-delay="true"
protocol-version="2.5" shared="true" hotrod-wrapping="true" purge="false"
passivation="false">
      <remote-server outbound-socket-binding="remote-store-hotrod-server"/>
   </remote-store>
</distributed-cache>
...
<socket-binding-group name="standard-sockets" default-interface="public"
port-offset="${jboss.socket.binding.port-offset:0}">
  ...
  <outbound-socket-binding name="remote-store-hotrod-server">
     <remote-destination host="198.51.100.0" port="11222"/>
  </outbound-socket-binding>
  ...
</socket-binding-group>
```

3. Configure the target cluster to handle all client requests instead of the source cluster:
   a. Configure all clients to point to the target cluster instead of the source cluster.
   b. Restart each client node.

   The target cluster lazily loads data from the source cluster on demand via `RemoteCacheStore`.

4.2. Synchronizing Data from the Source Cluster

1. Call the `synchronizeData()` method in the `TargetMigrator` interface. Do one of the following on the target cluster for each cache that you want to migrate:

   JMX

   Invoke the `synchronizeData` operation and specify the `hotrod` parameter on the `RollingUpgradeManager` MBean.

   CLI

   ```
   $ bin/ispn-cli.sh --connect controller=127.0.0.1:9990 -c "/subsystem=datagrid-
   infinispan/cache-container=clustered/distributed-cache=MyCache:synchronize-
   data(migrator-name=hotrod)"
   ```

   Data migrates to all nodes in the target cluster in parallel, with each node receiving a subset of the data.

   Use the following parameters to tune the operation:

   - `read-batch` configures the number of entries to read from the source cluster at a time. The default value is `10000`.

- `write-threads` configures the number of threads used to write data. The default value is the number of processors available.

  For example:

  ```
  synchronize-data(migrator-name=hotrod, read-batch=100000, write-threads=3)
  ```

2. Disable the `RemoteCacheStore` on the target cluster. Do one of the following:

   JMX

   Invoke the `disconnectSource` operation and specify the `hotrod` parameter on the `RollingUpgradeManager` MBean.

   CLI

   ```
   $ bin/ispn-cli.sh --connect controller=127.0.0.1:9990 -c "/subsystem=datagrid-infinispan/cache-container=clustered/distributed-cache=MyCache:disconnect-source(migrator-name=hotrod)"
   ```

3. Decommission the source cluster.

Chapter 5. Client/Server

{brandname} offers two alternative access methods: embedded mode and client-server mode.

- In Embedded mode the {brandname} libraries co-exist with the user application in the same JVM as shown in the following diagram

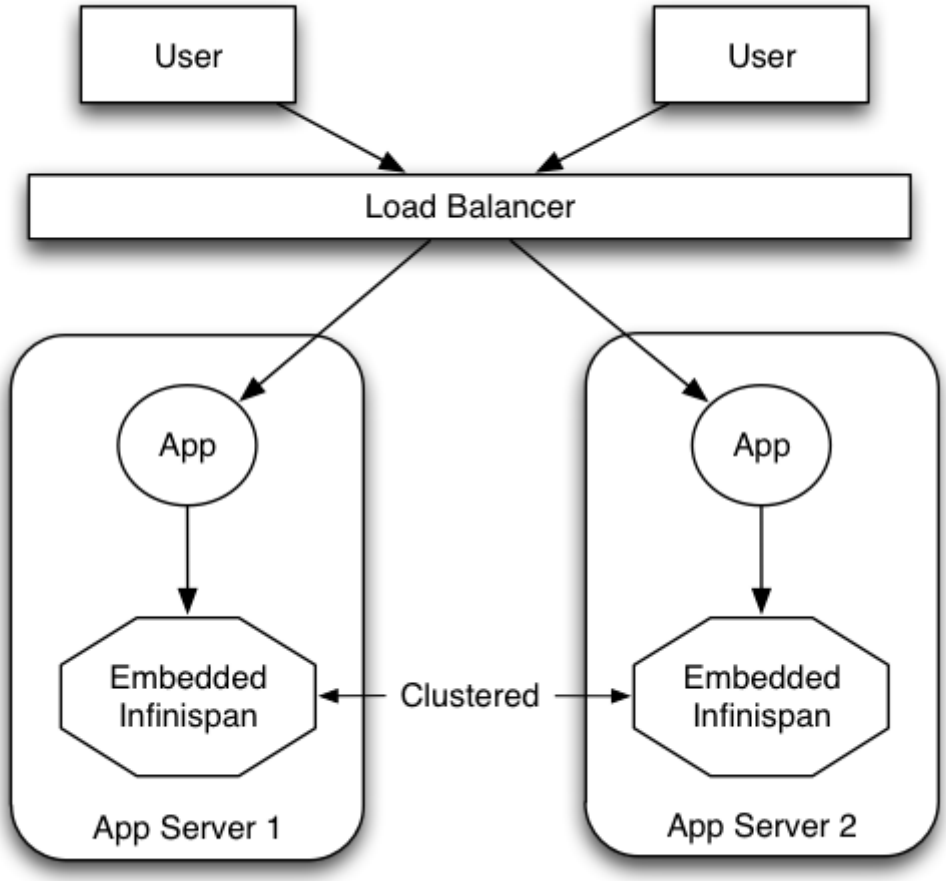


Figure 2. Peer-to-peer access

- Client-server mode is when applications access the data stored in a remote {brandname} server using some kind of network protocol

5.1. Why Client/Server?

There are situations when accessing {brandname} in a client-server mode might make more sense than embedding it within your application, for example, when trying to access {brandname} from a non-JVM environment. Since {brandname} is written in Java, if someone had a C\\ application that wanted to access it, it couldn't just do it in a p2p way. On the other hand, client-server would be perfectly suited here assuming that a language neutral protocol was used and the corresponding client and server implementations were available.

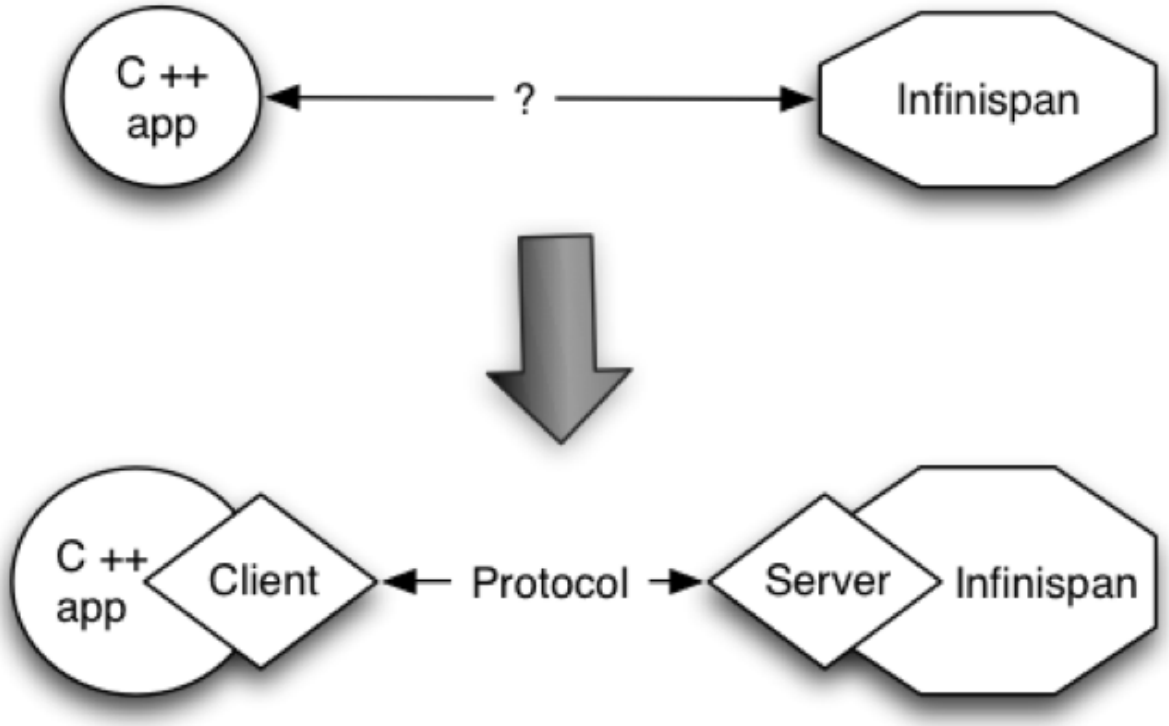


Figure 3. Non-JVM access

In other situations, {brandname} users want to have an elastic application tier where you start/stop business processing servers very regularly. Now, if users deployed {brandname} configured with distribution or state transfer, startup time could be greatly influenced by the shuffling around of data that happens in these situations. So in the following diagram, assuming {brandname} was deployed in p2p mode, the app in the second server could not access {brandname} until state transfer had completed.

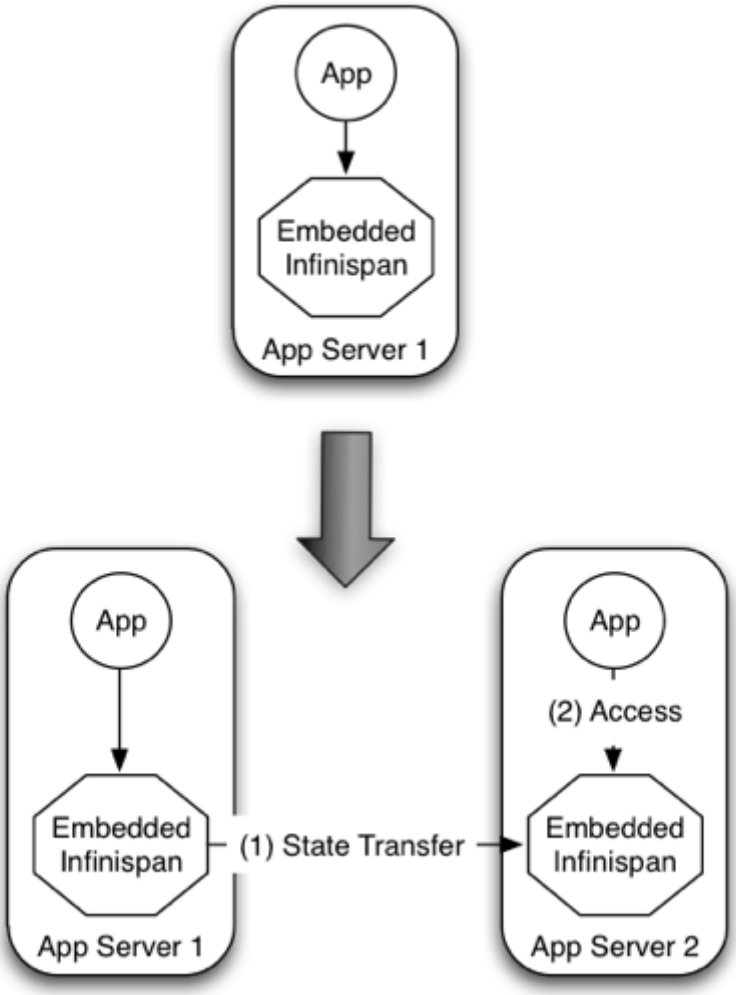


Figure 4. Elasticity issue with P2P

This effectively means that bringing up new application-tier servers is impacted by things like state transfer because applications cannot access {brandname} until these processes have finished and if the state being shifted around is large, this could take some time. This is undesirable in an elastic environment where you want quick application-tier server turnaround and predictable startup times. Problems like this can be solved by accessing {brandname} in a client-server mode because starting a new application-tier server is just a matter of starting a lightweight client that can connect to the backing data grid server. No need for rehashing or state transfer to occur and as a result server startup times can be more predictable which is very important for modern cloud-based deployments where elasticity in your application tier is important.

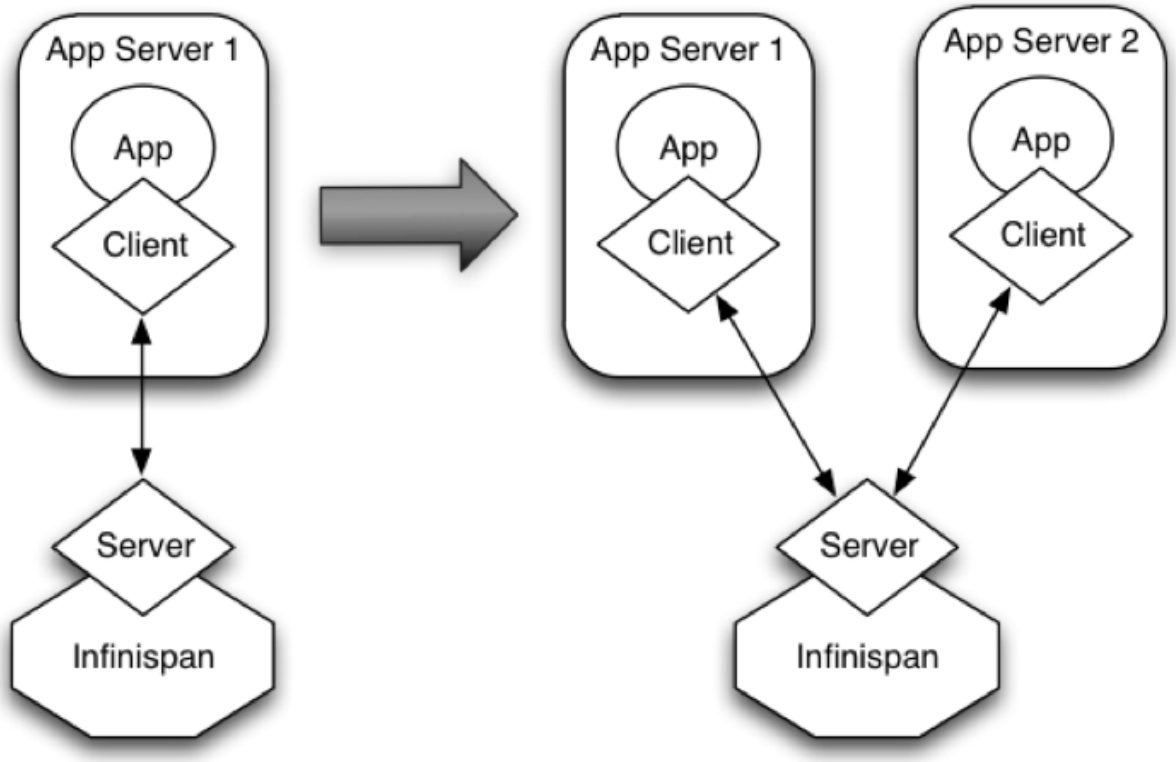


Figure 5. Achieving elasticity

Other times, it's common to find multiple applications needing access to data storage. In this cases, you could in theory deploy an {brandname} instance per each of those applications but this could be wasteful and difficult to maintain. Think about databases here, you don't deploy a database alongside each of your applications, do you? So, alternatively you could deploy {brandname} in client-server mode keeping a pool of {brandname} data grid nodes acting as a shared storage tier for your applications.

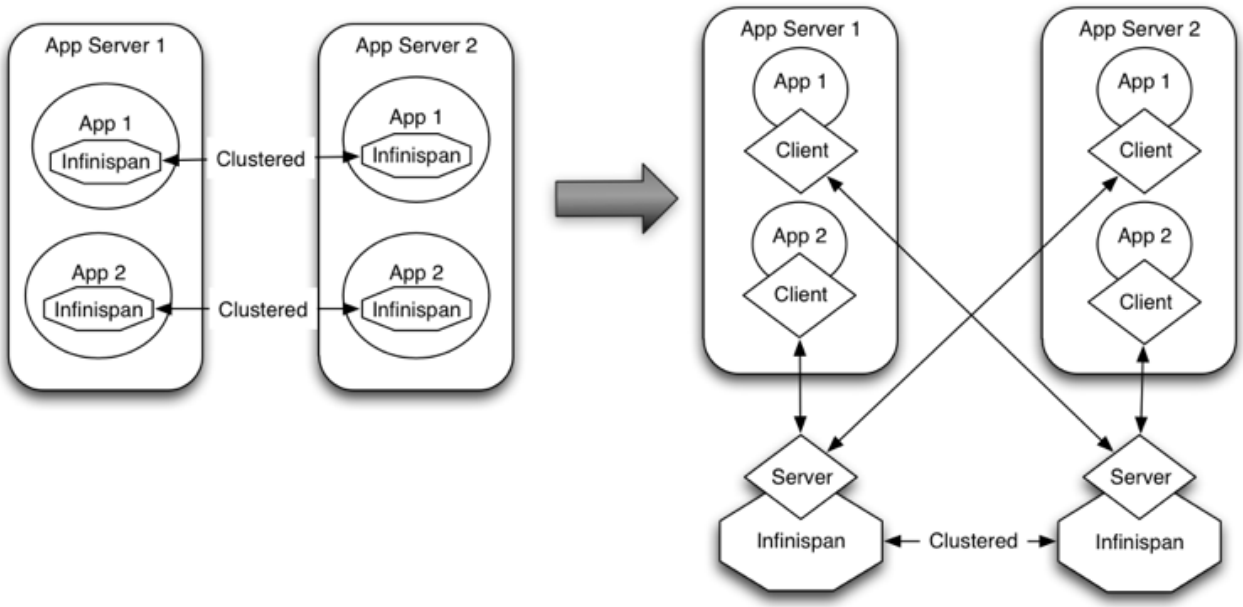


Figure 6. Shared data storage

Deploying {brandname} in this way also allows you to manage each tier independently, for example, you can upgrade you application or app server without bringing down your {brandname} data grid nodes.

5.2. Why use embedded mode?

Before talking about individual {brandname} server modules, it's worth mentioning that in spite of all the benefits, client-server {brandname} still has disadvantages over p2p. Firstly, p2p deployments are simpler than client-server ones because in p2p, all peers are equals to each other and hence this simplifies deployment. So, if this is the first time you're using {brandname}, p2p is likely to be easier for you to get going compared to client-server.

Client-server {brandname} requests are likely to take longer compared to p2p requests, due to the serialization and network cost in remote calls. So, this is an important factor to take in account when designing your application. For example, with replicated {brandname} caches, it might be more performant to have lightweight HTTP clients connecting to a server side application that accesses {brandname} in p2p mode, rather than having more heavyweight client side apps talking to {brandname} in client-server mode, particularly if data size handled is rather large. With distributed caches, the difference might not be so big because even in p2p deployments, you're not guaranteed to have all data available locally.

Environments where application tier elasticity is not so important, or where server side applications access state-transfer-disabled, replicated {brandname} cache instances are amongst scenarios where {brandname} p2p deployments can be more suited than client-server ones.

5.3. Server Modules

So, now that it's clear when it makes sense to deploy {brandname} in client-server mode, what are available solutions? All {brandname} server modules are based on the same pattern where the server backend creates an embedded {brandname} instance and if you start multiple backends, they can form a cluster and share/distribute state if configured to do so. The server types below primarily differ in the type of listener endpoint used to handle incoming connections.

Here's a brief summary of the available server endpoints.

- **Hot Rod Server Module** - This module is an implementation of the Hot Rod binary protocol backed by {brandname} which allows clients to do dynamic load balancing and failover and smart routing.
  - A variety of clients exist for this protocol.
  - If you're clients are running Java, this should be your defacto server module choice because it allows for dynamic load balancing and failover. This means that Hot Rod clients can dynamically detect changes in the topology of Hot Rod servers as long as these are clustered, so when new nodes join or leave, clients update their Hot Rod server topology view. On top of that, when Hot Rod servers are configured with distribution, clients can detect where a particular key resides and so they can route requests smartly.
  - Load balancing and failover is dynamically provided by Hot Rod client implementations using information provided by the server.

- **REST Server Module** - The REST server, which is distributed as a WAR file, can be deployed in any servlet container to allow {brandname} to be accessed via a RESTful HTTP interface.
  - To connect to it, you can use any HTTP client out there and there're tons of different client implementations available out there for pretty much any language or system.
  - This module is particularly recommended for those environments where HTTP port is the only access method allowed between clients and servers.
  - Clients wanting to load balance or failover between different {brandname} REST servers can do so using any standard HTTP load balancer such as mod_cluster . It's worth noting though these load balancers maintain a static view of the servers in the backend and if a new one was to be added, it would require manual update of the load balancer.
- **Memcached Server Module** - This module is an implementation of the Memcached text protocol backed by {brandname}.
  - To connect to it, you can use any of the existing Memcached clients which are pretty diverse.
  - As opposed to Memcached servers, {brandname} based Memcached servers can actually be clustered and hence they can replicate or distribute data using consistent hash algorithms around the cluster. So, this module is particularly of interest to those users that want to provide failover capabilities to the data stored in Memcached servers.
  - In terms of load balancing and failover, there're a few clients that can load balance or failover given a static list of server addresses (perl's Cache::Memcached for example) but any server addition or removal would require manual intervention.

5.4. Which protocol should I use?

Choosing the right protocol depends on a number of factors.

	Hot Rod	HTTP / REST	Memcached
Topology-aware	**Y**	**N**	**N**
Hash-aware	**Y**	**N**	**N**
Encryption	**Y**	**Y**	**N**
Authentication	**Y**	**Y**	**N**
Conditional ops	**Y**	**Y**	**Y**
Bulk ops	**Y**	**N**	**N**
Transactions	**N**	**N**	**N**
Listeners	**Y**	**N**	**N**
Query	**Y**	**Y**	**N**
Execution	**Y**	**N**	**N**
Cross-site failover	**Y**	**N**	**N**

Chapter 6. Using Hot Rod Server

The {brandname} Server distribution contains a server module that implements {brandname}'s custom binary protocol called Hot Rod. The protocol was designed to enable faster client/server interactions compared to other existing text based protocols and to allow clients to make more intelligent decisions with regards to load balancing, failover and even data location operations. Please refer to {brandname} Server's documentation for instructions on how to configure and run a HotRod server.

To connect to {brandname} over this highly efficient Hot Rod protocol you can either use one of the clients described in this chapter, or use higher level tools such as Hibernate OGM.

6.1. Java Hot Rod client

Hot Rod is a binary, language neutral protocol. This article explains how a Java client can interact with a server via the Hot Rod protocol. A reference implementation of the protocol written in Java can be found in all {brandname} distributions, and this article focuses on the capabilities of this java client.

Looking for more clients? Visit this website for clients written in a variety of different languages.

6.1.1. Configuration

The Java Hot Rod client can be configured both programmatically and externally, through a configuration file.

The code snippet below illustrates the creation of a client instance using the available Java fluent API:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
      = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.tcpNoDelay(true)
  .connectionPool()
      .numTestsPerEvictionRun(3)
      .testOnBorrow(false)
      .testOnReturn(false)
      .testWhileIdle(true)
  .addServer()
      .host("localhost")
      .port(11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

For a complete reference to the available configuration option please refer to the ConfigurationBuilder's javadoc.

It is also possible to configure the Java Hot Rod client using a properties file, e.g.:

```
infinispan.client.hotrod.transport_factory =
org.infinispan.client.hotrod.impl.transport.tcp.TcpTransportFactory
infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.marshaller =
org.infinispan.commons.marshall.jboss.GenericJBossMarshaller
infinispan.client.hotrod.async_executor_factory =
org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory
infinispan.client.hotrod.default_executor_factory.pool_size = 1
infinispan.client.hotrod.default_executor_factory.queue_size = 10000
infinispan.client.hotrod.tcp_no_delay = true
infinispan.client.hotrod.request_balancing_strategy =
org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy
infinispan.client.hotrod.key_size_estimate = 64
infinispan.client.hotrod.value_size_estimate = 512
infinispan.client.hotrod.force_return_values = false
infinispan.client.hotrod.client_intelligence = HASH_DISTRIBUTION_AWARE
infinispan.client.hotrod.batch_Size = 10000

## below is connection pooling config
maxActive=-1
maxTotal = -1
maxIdle = -1
whenExhaustedAction = 1
timeBetweenEvictionRunsMillis=120000
minEvictableIdleTimeMillis=300000
testWhileIdle = true
minIdle = 1
```

The properties file is then passed to one of constructors of RemoteCacheManager. You can use property substitution to replace values at runtime with Java system properties:

```
infinispan.client.hotrod.server_list = ${server_list}
```

In the above example the value of the *infinispan.client.hotrod.server_list* property will be expanded to the value of the *server_list* Java system property.

which means that the value should be taken from a system property named jboss.bind.address.management and if it is not defined use 127.0.0.1.

For a complete reference of the available configuration options for the properties file please refer to remote client configuration javadoc.

6.1.2. Authentication

If the server has set up authentication, you need to configure your client accordingly. Depending on the mechs enabled on the server, the client must provide the required information.

This section is about client configuration. If you want to set up the server to require authentication, read the Hot Rod server authentication section.

DIGEST-MD5

DIGEST-MD5 is the recommended approach for simple username/password authentication scenarios. If you are using the default realm on the server (*"ApplicationRealm"*), all you need to do is provide your credentials as follows:

Hot Rod client configuration with DIGEST-MD5 authentication

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            .username("myuser")
            .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

PLAIN

The PLAIN mechanism is not really recommended unless the connection is also encrypted, as anyone can sniff the clear-text password being sent along the wire.

Hot Rod client configuration with DIGEST-MD5 authentication

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .authentication()
            .saslMechanism("PLAIN")
            .username("myuser")
            .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

EXTERNAL

The EXTERNAL mechanism is special in that it doesn't explicitly provide credentials but uses the client certificate as identity. In order for this to work, in addition to the *TrustStore* (to validate the server certificate) you need to provide a *KeyStore* (to supply the client certificate).

Hot Rod client configuration with EXTERNAL authentication (client cert)

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            // TrustStore is a KeyStore which contains part of the server certificate
chain (e.g. the CA Root public cert)
            .trustStoreFileName("/path/to/truststore")
            .trustStorePassword("truststorepassword".toCharArray())
            // KeyStore containing this client's own certificate
            .keyStoreFileName("/path/to/keystore")
            .keyStorePassword("keystorepassword".toCharArray())
        .authentication()
            .saslMechanism("EXTERNAL");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

For more details, read the Encryption section below.

GSSAPI (Kerberos)

GSSAPI/Kerberos requires a much more complex setup, but it is used heavily in enterprises with centralized authentication servers. To successfully authenticate with Kerberos, you need to create a *LoginContext*. This will obtain a Ticket Granting Ticket (TGT) which will be used as a token to authenticate with the service.

You will need to define a login module in a login configuration file:

gss.conf

```
GssExample {
    com.sun.security.auth.module.Krb5LoginModule required client=TRUE;
};
```

If you are using the IBM JDK, the above becomes:

gss-ibm.conf

```
GssExample {
    com.ibm.security.auth.module.Krb5LoginModule required client=TRUE;
};
```

You will also need to set the following system properties:

java.security.auth.login.config=gss.conf

java.security.krb5.conf=/etc/krb5.conf

The krb5.conf file is dependent on your environment and needs to point to your KDC. Ensure that you can authenticate via Kerberos using *kinit*.

Next up, configure your client as follows:

Hot Rod client GSSAPI configuration

```
LoginContext lc = new LoginContext("GssExample", new BasicCallbackHandler("krb_user",
"krb_password".toCharArray()));
lc.login();
Subject clientSubject = lc.getSubject();

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .authentication()
            .enable()
            .serverName("infinispan-server")
            .saslMechanism("GSSAPI")
            .clientSubject(clientSubject)
            .callbackHandler(new BasicCallbackHandler());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

For brevity we used the same callback handler both for obtaining the client subject and for handling authentication in the SASL GSSAPI mech, however different callbacks will actually be invoked: NameCallback and PasswordCallback are needed to construct the client subject, while the AuthorizeCallback will be called during the SASL authentication.

Custom CallbackHandlers

In all of the above examples, the Hot Rod client is setting up a default *CallbackHandler* for you that supplies the provided credentials to the SASL mechanism. For advanced scenarios it may be necessary to provide your own custom *CallbackHandler*:

Hot Rod client configuration with authentication via callback

```
public class MyCallbackHandler implements CallbackHandler {
   final private String username;
   final private char[] password;
   final private String realm;

   public MyCallbackHandler(String username, String realm, char[] password) {
      this.username = username;
      this.password = password;
      this.realm = realm;
```

```
   }

   @Override
   public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
      for (Callback callback : callbacks) {
         if (callback instanceof NameCallback) {
            NameCallback nameCallback = (NameCallback) callback;
            nameCallback.setName(username);
         } else if (callback instanceof PasswordCallback) {
            PasswordCallback passwordCallback = (PasswordCallback) callback;
            passwordCallback.setPassword(password);
         } else if (callback instanceof AuthorizeCallback) {
            AuthorizeCallback authorizeCallback = (AuthorizeCallback) callback;
            authorizeCallback.setAuthorized(authorizeCallback.getAuthenticationID()
.equals(
                  authorizeCallback.getAuthorizationID()));
         } else if (callback instanceof RealmCallback) {
            RealmCallback realmCallback = (RealmCallback) callback;
            realmCallback.setText(realm);
         } else {
            throw new UnsupportedCallbackException(callback);
         }
      }
   }
}

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .authentication()
            .enable()
            .serverName("myhotrodserver")
            .saslMechanism("DIGEST-MD5")
            .callbackHandler(new MyCallbackHandler("myuser", "ApplicationRealm",
"qwer1234!".toCharArray()));
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

The actual type of callbacks that your CallbackHandler will need to be able to handle are mech-specific, so the above is just a simple example.

6.1.3. Encryption

This section is about client configuration. If you want to set up the server to require encryption, read the Hot Rod server encryption section.

Encryption uses TLS/SSL, so it requires setting up an appropriate server certificate chain. Generally, a certificate chain looks like the following:

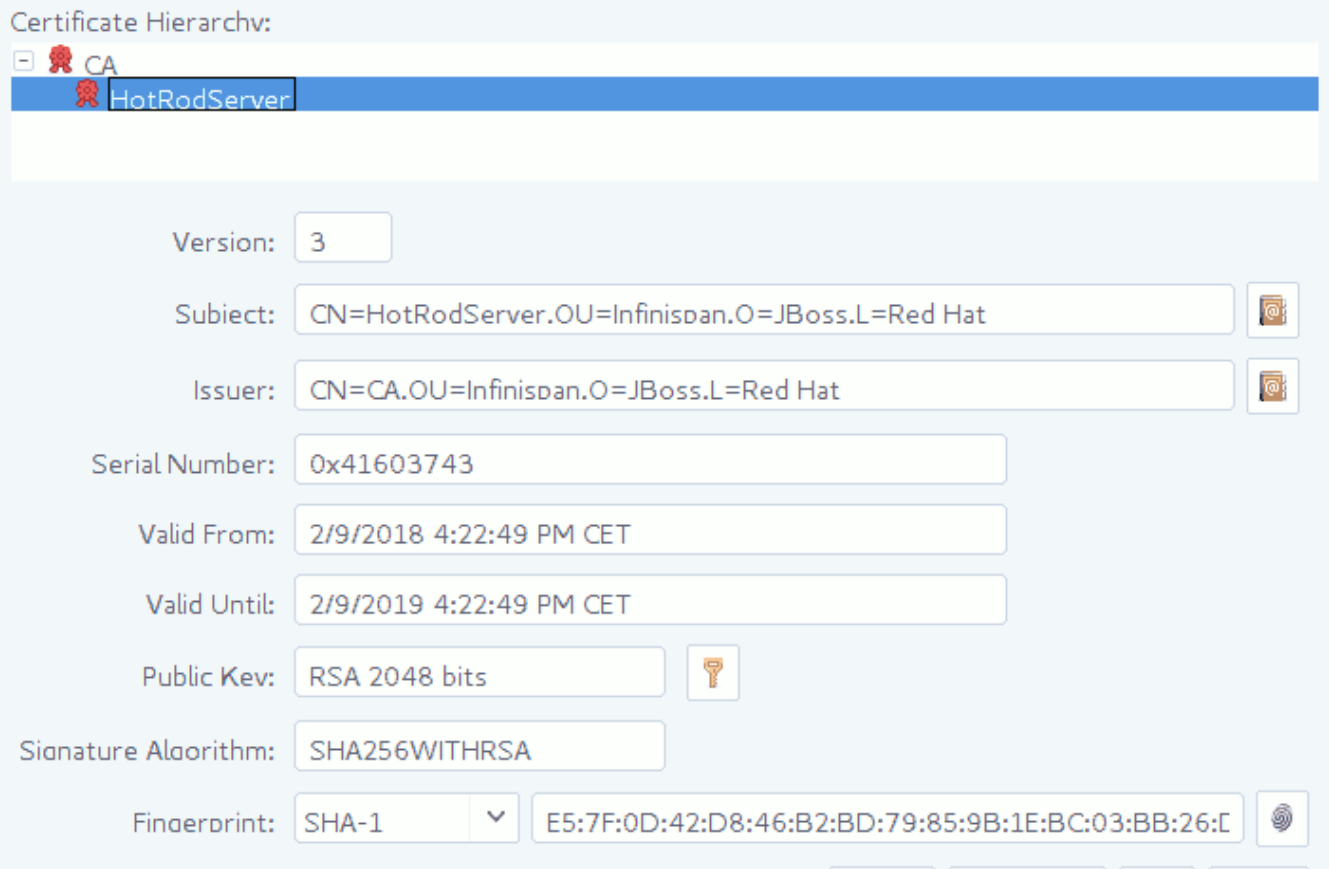


Figure 7. Certificate chain

In the above example there is one certificate authority *"CA"* which has issued a certificate for *"HotRodServer"*. In order for a client to trust the server, it needs at least a portion of the above chain (usually, just the public certificate for *"CA"*). This certificate needs to placed in a keystore and used as a *TrustStore* on the client and used as shown below:

Hot Rod client configuration with TLS (server cert)

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            // TrustStore is a KeyStore which contains part of the server certificate
chain (e.g. the CA Root public cert)
            .trustStoreFileName("/path/to/truststore")
            .trustStorePassword("truststorepassword".toCharArray());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

SNI

The server may have been configured with TLS/SNI support (Server Name Indication). This means

that the server is presenting multiple identities (probably bound to separate cache containers). The client can specify which identity to connect to by specifying its name:

Hot Rod client configuration with SNI (server cert)

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            .sniHostName("myservername")
            // TrustStore is a KeyStore which contains part of the server certificate
chain (e.g. the CA Root public cert)
            .trustStoreFileName("/path/to/truststore")
            .trustStorePassword("truststorepassword".toCharArray());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

Client certificates

With the above configurations the client trusts the server. For increased security, a server administrator may have set up the server to require the client to offer a valid certificate for mutual trust. This kind of configuration requires the client to present its own certificate, usually issued by the same certificate authority as the server. This certificate must be stored in a keystore and used as follows:

Hot Rod client configuration with TLS (server and client cert)

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            // TrustStore is a KeyStore which contains part of the server certificate
chain (e.g. the CA Root public cert)
            .trustStoreFileName("/path/to/truststore")
            .trustStorePassword("truststorepassword".toCharArray())
            // KeyStore containing this client's own certificate
            .keyStoreFileName("/path/to/keystore")
            .keyStorePassword("keystorepassword".toCharArray())
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

Please read the KeyTool documentation for more details on KeyStores. Additionally, the KeyStore Explorer is a great GUI tool for easily managing KeyStores.

6.1.4. Basic API

Below is a sample code snippet on how the client API can be used to store or retrieve information from a Hot Rod server using the Java Hot Rod client. It assumes that a Hot Rod server has been started bound to the default location (localhost:11222)

```
//API entry point, by default it connects to localhost:11222
CacheContainer cacheContainer = new RemoteCacheManager();

//obtain a handle to the remote default cache
Cache<String, String> cache = cacheContainer.getCache();

//now add something to the cache and make sure it is there
cache.put("car", "ferrari");
assert cache.get("car").equals("ferrari");

//remove the data
cache.remove("car");
assert !cache.containsKey("car") : "Value must have been removed!";
```

The client API maps the local API: RemoteCacheManager corresponds to DefaultCacheManager (both implement CacheContainer). This common API facilitates an easy migration from local calls to remote calls through Hot Rod: all one needs to do is switch between DefaultCacheManager and RemoteCacheManager - which is further simplified by the common CacheContainer interface that both inherit.

6.1.5. RemoteCache(.keySet|.entrySet|.values)

The collection methods `keySet`, `entrySet` and `values` are backed by the remote cache. That is that every method is called back into the `RemoteCache`. This is useful as it allows for the various keys, entries or values to be retrieved lazily, and not requiring them all be stored in the client memory at once if the user does not want. These collections adhere to the `Map` specification being that `add` and `addAll` are not supported but all other methods are supported.

One thing to note is the `Iterator.remove` and `Set.remove` or `Collection.remove` methods require more than 1 round trip to the server to operate. You can check out the RemoteCache Javadoc to see more details about these and the other methods.

Iterator Usage

The iterator method of these collections uses `retrieveEntries` internally, which is described below. If you notice `retrieveEntries` takes an argument for the batch size. There is no way to provide this to the iterator. As such the batch size can be configured via system property `infinispan.client.hotrod.batch_size` or through the ConfigurationBuilder when configuring the `RemoteCacheManager`.

Also the `retrieveEntries` iterator returned is `Closeable` as such the iterators from `keySet`, `entrySet` and `values` return an `AutoCloseable` variant. Therefore you should always close these `Iterator`s when you are done with them.

```
try (CloseableIterator<Entry<K, V>> iterator = remoteCache.entrySet().iterator) {
 ...
}
```

What if I want a deep copy and not a backing collection?

Previous version of `RemoteCache` allowed for the retrieval of a deep copy of the `keySet`. This is still possible with the new backing map, you just have to copy the contents yourself. Also you can do this with `entrySet` and `values`, which we didn't support before.

```
Set<K> keysCopy = remoteCache.keySet().stream().collect(Collectors.toSet());
```

Please use extreme cautiong with this as a large number of keys can and will cause OutOfMemoryError in the client.

```
Set keys = remoteCache.keySet();
```

6.1.6. Remote Iterator

Alternatively, if memory is a concern (different batch size) or you wish to do server side filtering or conversion), use the remote iterator api to retrieve entries from the server. With this method you can limit the entries that are retrieved or even returned a converted value if you dont' need all properties of your entry.

```
// Retrieve all entries in batches of 1000
int batchSize = 1000;
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries
(null, batchSize)) {
     while(iterator.hasNext()) {
        // Do something
     }
}

// Filter by segment
Set<Integer> segments = ...
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries
(null, segments, batchSize)) {
     while(iterator.hasNext()) {
        // Do something
     }
}

// Filter by custom filter
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(
"myFilterConverterFactory", segments, batchSize)) {
     while(iterator.hasNext()) {
        // Do something
     }
}
```

In order to use custom filters, it's necessary to deploy them first in the server. Follow the steps:

- Create a factory for the filter extending KeyValueFilterConverterFactory, annotated with @NamedFactory containing the name of the factory, example:

```
import java.io.Serializable;

import org.infinispan.filter.AbstractKeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverterFactory;
import org.infinispan.filter.NamedFactory;
import org.infinispan.metadata.Metadata;

@NamedFactory(name = "myFilterConverterFactory")
public class MyKeyValueFilterConverterFactory implements
KeyValueFilterConverterFactory {

   @Override
   public KeyValueFilterConverter<String, SampleEntity1, SampleEntity2>
getFilterConverter() {
      return new MyKeyValueFilterConverter();
   }
   // Filter implementation. Should be serializable or externalizable for DIST caches
   static class MyKeyValueFilterConverter extends AbstractKeyValueFilterConverter
<String, SampleEntity1, SampleEntity2> implements Serializable {
      @Override
      public SampleEntity2 filterAndConvert(String key, SampleEntity1 entity, Metadata
metadata) {
         // returning null will case the entry to be filtered out
         // return SampleEntity2 will convert from the cache type SampleEntity1
      }

      @Override
      public MediaType format() {
         // returns the MediaType that data should be presented to this converter.
         // When ommitted, the server will use "application/x-java-object".
         // Returning null will cause the filter/converter to be done in the storage
format.
      }
   }
}
```

- Create a jar with a `META-INF/services/org.infinispan.filter.KeyValueFilterConverterFactory` file and within it, write the fully qualified class name of the filter factory class implementation.
- Optional: If the filter uses custom key/value classes, these must be included in the JAR so that the filter can correctly unmarshall key and/or value instances.
- Deploy the JAR file in the {brandname} Server.

6.1.7. Versioned API

A RemoteCacheManager provides instances of RemoteCache interface that represents a handle to the named or default cache on the remote cluster. API wise, it extends the Cache interface to which it also adds some new methods, including the so called versioned API. Please find below some

examples of this API link:#server_hotrod_failover[but to understand the motivation behind it, make sure you read this section.

The code snippet bellow depicts the usage of these versioned methods:

```
// To use the versioned API, remote classes are specifically needed
RemoteCacheManager remoteCacheManager = new RemoteCacheManager();
RemoteCache<String, String> cache = remoteCacheManager.getCache();

remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");

// removal only takes place only if the version has not been changed
// in between. (a new version is associated with 'car' key on each change)
assert remoteCache.remove("car", valueBinary.getVersion());
assert !cache.containsKey("car");
```

In a similar way, for replace:

```
remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");
assert remoteCache.replace("car", "lamborghini", valueBinary.getVersion());
```

For more details on versioned operations refer to RemoteCache 's javadoc.

6.1.8. Streaming API

When sending / receiving large objects, it might make sense to stream them between the client and the server. The Streaming API implements methods similar to the Hot Rod Basic API and Hot Rod Versioned API described above but, instead of taking the value as a parameter, they return instances of InputStream and OutputStream. The following example shows how one would write a potentially large object:

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
OutputStream os = streamingCache.put("a_large_object");
os.write(...);
os.close();
```

Reading such an object through streaming:

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
for(int b = is.read(); b >= 0; b = is.read()) {
   ...
}
is.close();
```

The streaming API does **not** apply marshalling/unmarshalling to the values. For this reason you cannot access the same entries using both the streaming and non-streaming API at the same time, unless you provide your own marshaller to detect this situation.

The InputStream returned by the `RemoteStreamingCache.get(K key)` method implements the `VersionedMetadata` interface, so you can retrieve version and expiration information:

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
int version = ((VersionedMetadata) is).getVersion();
for(int b = is.read(); b >= 0; b = is.read()) {
   ...
}
is.close();
```

Conditional write methods (`putIfAbsent`, `replace`) only perform the actual condition check once the value has been completely sent to the server (i.e. when the `close()` method has been invoked on the `OutputStream`.

6.1.9. Creating Event Listeners

Java Hot Rod clients can register listeners to receive cache-entry level events. Cache entry created, modified and removed events are supported.

Creating a client listener is very similar to embedded listeners, except that different annotations and event classes are used. Here's an example of a client listener that prints out each event received:

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class EventPrintListener {

   @ClientCacheEntryCreated
   public void handleCreatedEvent(ClientCacheEntryCreatedEvent e) {
      System.out.println(e);
   }

   @ClientCacheEntryModified
   public void handleModifiedEvent(ClientCacheEntryModifiedEvent e) {
      System.out.println(e);
   }

   @ClientCacheEntryRemoved
   public void handleRemovedEvent(ClientCacheEntryRemovedEvent e) {
      System.out.println(e);
   }

}
```

`ClientCacheEntryCreatedEvent` and `ClientCacheEntryModifiedEvent` instances provide information on the affected key, and the version of the entry. This version can be used to invoke conditional operations on the server, such as `replaceWithVersion` or `removeWithVersion`.

`ClientCacheEntryRemovedEvent` events are only sent when the remove operation succeeds. In other words, if a remove operation is invoked but no entry is found or no entry should be removed, no event is generated. Users interested in removed events, even when no entry was removed, can develop event customization logic to generate such events. More information can be found in the customizing client events section.

All `ClientCacheEntryCreatedEvent`, `ClientCacheEntryModifiedEvent` and `ClientCacheEntryRemovedEvent` event instances also provide a `boolean isCommandRetried()` method that will return true if the write command that caused this had to be retried again due to a topology change. This could be a sign that this event has been duplicated or another event was dropped and replaced (eg: ClientCacheEntryModifiedEvent replaced ClientCacheEntryCreatedEvent).

Once the client listener implementation has been created, it needs to be registered with the server. To do so, execute:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());
```

6.1.10. Removing Event Listeners

When an client event listener is not needed any more, it can be removed:

```
EventPrintListener listener = ...
cache.removeClientListener(listener);
```

6.1.11. Filtering Events

In order to avoid inundating clients with events, users can provide filtering functionality to limit the number of events fired by the server for a particular client listener. To enable filtering, a cache event filter factory needs to be created that produces filter instances:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-filter")
class StaticCacheEventFilterFactory implements CacheEventFilterFactory {
   @Override
   public CacheEventFilterFactory<Integer, String> getFilter(Object[] params) {
      return new StaticCacheEventFilter();
   }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventFilter implements CacheEventFilter<Integer, String>,
Serializable {
   @Override
   public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
         String newValue, Metadata newMetadata, EventType eventType) {
      if (key.equals(1)) // static key
         return true;

      return false;
   }
}
```

The cache event filter factory instance defined above creates filter instances which statically filter out all entries except the one whose key is 1.

To be able to register a listener with this cache event filter factory, the factory has to be given a unique name, and the Hot Rod server needs to be plugged with the name and the cache event filter factory instance. Plugging the {brandname} Server with a custom filter involves the following steps:

1. Create a JAR file with the filter implementation within it.
2. Optional: If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the

client listener has `useRawData` enabled, this is not necessary since the callback key/value instances will be provided in binary format.

3. Create a `META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory` file within the JAR file and within it, write the fully qualified class name of the filter class implementation.
4. Deploy the JAR file in the {brandname} Server.

On top of that, the client listener needs to be linked with this cache event filter factory by adding the factory's name to the @ClientListener annotation:

```
@ClientListener(filterFactoryName = "static-filter")
public class EventPrintListener { ... }
```

And, register the listener with the server:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());
```

Dynamic filter instances that filter based on parameters provided when the listener is registered are also possible. Filters use the parameters received by the filter factories to enable this option. For example:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventFilter;

class DynamicCacheEventFilterFactory implements CacheEventFilterFactory {
   @Override
   public CacheEventFilter<Integer, String> getFilter(Object[] params) {
      return new DynamicCacheEventFilter(params);
   }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class DynamicCacheEventFilter implements CacheEventFilter<Integer, String>,
Serializable {
   final Object[] params;

   DynamicCacheEventFilter(Object[] params) {
      this.params = params;
   }

   @Override
   public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
         String newValue, Metadata newMetadata, EventType eventType) {
      if (key.equals(params[0])) // dynamic key
         return true;

      return false;
   }
}
```

The dynamic parameters required to do the filtering are provided when the listener is registered:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), new Object[]{1}, null);
```

Filter instances have to marshallable when they are deployed in a cluster so that the filtering can happen right where the event is generated, even if the even is generated in a different node to where the listener is registered. To make them marshallable, either make them extend `Serializable`, `Externalizable`, or provide a custom `Externalizer` for them.

Skipping Notifications

Include the `SKIP_LISTENER_NOTIFICATION` flag when calling remote API methods to perform operations without getting event notifications from the server. For example, to prevent listener notifications when creating or modifying values, set the flag as follows:

```
remoteCache.withFlags(Flag.SKIP_LISTENER_NOTIFICATION).put(1, "one");
```

6.1.12. Customizing Events

The events generated by default contain just enough information to make the event relevant but they avoid cramming too much information in order to reduce the cost of sending them. Optionally, the information shipped in the events can be customised in order to contain more information, such as values, or to contain even less information. This customization is done with `CacheEventConverter` instances generated by a `CacheEventConverterFactory`:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-converter")
class StaticConverterFactory implements CacheEventConverterFactory {
   final CacheEventConverter<Integer, String, CustomEvent> staticConverter = new
StaticCacheEventConverter();
   public CacheEventConverter<Integer, String, CustomEvent> getConverter(final
Object[] params) {
      return staticConverter;
   }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventConverter implements CacheEventConverter<Integer, String,
CustomEvent>, Serializable {
   public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata,
String newValue, Metadata newMetadata, EventType eventType) {
      return new CustomEvent(key, newValue);
   }
}

// Needs to be Serializable, Externalizable or marshallable with Infinispan
Externalizers
// regardless of cluster or local caches
static class CustomEvent implements Serializable {
   final Integer key;
   final String value;
   CustomEvent(Integer key, String value) {
      this.key = key;
      this.value = value;
   }
}
```

In the example above, the converter generates a new custom event which includes the value as well as the key in the event. This will result in bigger event payloads compared with default events, but

if combined with filtering, it can reduce its network bandwidth cost.

The target type of the converter must be either `Serializable` or `Externalizable`. In this particular case of converters, providing an Externalizer will not work by default since the default Hot Rod client marshaller does not support them.

Handling custom events requires a slightly different client listener implementation to the one demonstrated previously. To be more precise, it needs to handle `ClientCacheEntryCustomEvent` instances:

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class CustomEventPrintListener {

   @ClientCacheEntryCreated
   @ClientCacheEntryModified
   @ClientCacheEntryRemoved
   public void handleCustomEvent(ClientCacheEntryCustomEvent<CustomEvent> e) {
      System.out.println(e);
   }

}
```

The `ClientCacheEntryCustomEvent` received in the callback exposes the custom event via `getEventData` method, and the `getType` method provides information on whether the event generated was as a result of cache entry creation, modification or removal.

Similar to filtering, to be able to register a listener with this converter factory, the factory has to be given a unique name, and the Hot Rod server needs to be plugged with the name and the cache event converter factory instance. Plugging the {brandname} Server with an event converter involves the following steps:

1. Create a JAR file with the converter implementation within it.
2. Optional: If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the client listener has `useRawData` enabled, this is not necessary since the callback key/value instances will be provided in binary format.
3. Create a `META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory` file within the JAR file and within it, write the fully qualified class name of the converter class implementation.
4. Deploy the JAR file in the {brandname} Server.

On top of that, the client listener needs to be linked with this converter factory by adding the factory's name to the @ClientListener annotation:

```
@ClientListener(converterFactoryName = "static-converter")
public class CustomEventPrintListener { ... }
```

And, register the listener with the server:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener());
```

Dynamic converter instances that convert based on parameters provided when the listener is registered are also possible. Converters use the parameters received by the converter factories to enable this option. For example:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

@NamedFactory(name = "dynamic-converter")
class DynamicCacheEventConverterFactory implements CacheEventConverterFactory {
   public CacheEventConverter<Integer, String, CustomEvent> getConverter(final
Object[] params) {
      return new DynamicCacheEventConverter(params);
   }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed
when running in a cluster
class DynamicCacheEventConverter implements CacheEventConverter<Integer, String,
CustomEvent>, Serializable {
   final Object[] params;

   DynamicCacheEventConverter(Object[] params) {
      this.params = params;
   }

   public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata,
         String newValue, Metadata newMetadata, EventType eventType) {
      // If the key matches a key given via parameter, only send the key information
      if (params[0].equals(key))
         return new CustomEvent(key, null);

      return new CustomEvent(key, newValue);
   }
}
```

The dynamic parameters required to do the conversion are provided when the listener is registered:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), null, new Object[]{1});
```

Converter instances have to marshallable when they are deployed in a cluster, so that the conversion can happen right where the event is generated, even if the even is generated in a different node to where the listener is registered. To make them marshallable, either make them extend `Serializable`, `Externalizable`, or provide a custom `Externalizer` for them.

6.1.13. Filter and Custom Events

If you want to do both event filtering and customization, it's easier to implement `org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter` which allows both filter and customization to happen in a single step. For convenience, it's recommended to extend `org.infinispan.notifications.cachelistener.filter.AbstractCacheEventFilterConverter` instead of implementing `org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter` directly. For example:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

@NamedFactory(name = "dynamic-filter-converter")
class DynamicCacheEventFilterConverterFactory implements
CacheEventFilterConverterFactory {
   public CacheEventFilterConverter<Integer, String, CustomEvent> getFilterConverter
(final Object[] params) {
      return new DynamicCacheEventFilterConverter(params);
   }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed
when running in a cluster
//
class DynamicCacheEventFilterConverter extends AbstractCacheEventFilterConverter
<Integer, String, CustomEvent>, Serializable {
   final Object[] params;

   DynamicCacheEventFilterConverter(Object[] params) {
      this.params = params;
   }

   public CustomEvent filterAndConvert(Integer key, String oldValue, Metadata
oldMetadata,
         String newValue, Metadata newMetadata, EventType eventType) {
      // If the key matches a key given via parameter, only send the key information
      if (params[0].equals(key))
         return new CustomEvent(key, null);

      return new CustomEvent(key, newValue);
   }
}
```

Similar to filters and converters, to be able to register a listener with this combined filter/converter factory, the factory has to be given a unique name via the `@NamedFactory` annotation, and the Hot Rod server needs to be plugged with the name and the cache event converter factory instance. Plugging the {brandname} Server with an event converter involves the following steps:

1. Create a JAR file with the converter implementation within it.
2. Optional: If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the client listener has `useRawData` enabled, this is not necessary since the callback key/value instances will be provided in binary format.
3. Create a `META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverterFactory` file within the JAR file and within it, write the fully qualified class name of the converter class implementation.

4. Deploy the JAR file in the {brandname} Server.

From a client perspective, to be able to use the combined filter and converter class, the client listener must define the same filter factory and converter factory names, e.g.:

```
@ClientListener(filterFactoryName = "dynamic-filter-converter", converterFactoryName =
"dynamic-filter-converter")
public class CustomEventPrintListener { ... }
```

The dynamic parameters required in the example above are provided when the listener is registered via either filter or converter parameters. If filter parameters are non-empty, those are used, otherwise, the converter parameters:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener(), new Object[]{1}, null);
```

6.1.14. Event Marshalling

Hot Rod servers can store data in different formats, but in spite of that, Java Hot Rod client users can still develop `CacheEventConverter` or `CacheEventFilter` instances that work on typed objects. By default, filters and converter will use data as POJO (application/x-java-object) but it is possible to override the desired format by overriding the method `format()` from the filter/converter. If the format returns `null`, the filter/converter will receive data as it's stored.

As indicated in the Marshalling Data section, Hot Rod Java clients can be configured to use a different `org.infinispan.commons.marshall.Marshaller` instance. If doing this and deploying `CacheEventConverter` or `CacheEventFilter` instances, to be able to present filters/converter with Java Objects rather than marshalled content, the server needs to be able to convert between objects and the binary format produced by the marshaller.

To deploy a Marshaller instance server-side, follow a similar method to the one used to deploy `CacheEventConverter` or `CacheEventFilter` instances:

1. Create a JAR file with the converter implementation within it.
2. Create a `META-INF/services/org.infinispan.commons.marshall.Marshaller` file within the JAR file and within it, write the fully qualified class name of the marshaller class implementation.
3. Deploy the JAR file in the {brandname} Server.

Note that the Marshaller could be deployed in either a separate jar, or in the same jar as the `CacheEventConverter` and/or `CacheEventFilter` instances.

Deploying Protostream Marshallers

If a cache stores protobuf content, as it happens when using protostream marshaller in the Hot Rod client, it's not necessary to deploy a custom marshaller since the format is already support by the server: there are transcoders from protobuf format to most common formats like JSON and POJO.

When using filters/converters with those caches, and it's desirable to use filter/converters with Java Objects rather binary prototobuf data, it's necessary to deploy the extra protostream marshallers so that the server can unmarshall the data before filtering/converting. To do so, follow these steps:

1. Create a jar and include an implementation of the interface `org.infinispan.query.remote.client.ProtostreamSerializationContextInitializer`, adding extra marshallers and optionally extra protobuf files to the cache manager's Serialization context.
2. Create a `META-INF/services/org.infinispan.query.remote.client.ProtostreamSerializationContextInitializer` file within the JAR file containing the fully qualified class name of the `ProtostreamSerializationContextInitializer` class implementation.
3. Create a META-INF/MANIFEST.MF with `Dependencies: org.infinispan.protostream, org.infinispan.remote-query.client`
4. Deploy the JAR file in the {brandname} Server in the `standalone/deployments` folder
5. Configure this deployment in the desired cache manager:

```
<cache-container name="local" default-cache="default">
   <modules>
     <module name="deployment.my-file.jar"/>
   </modules>
   ...
</cache-container>
```

The deployment must be available during the server startup!

6.1.15. Listener State Handling

Client listener annotation has an optional `includeCurrentState` attribute that specifies whether state will be sent to the client when the listener is added or when there's a failover of the listener.

By default, `includeCurrentState` is false, but if set to true and a client listener is added in a cache already containing data, the server iterates over the cache contents and sends an event for each entry to the client as a `ClientCacheEntryCreated` (or custom event if configured). This allows clients to build some local data structures based on the existing content. Once the content has been iterated over, events are received as normal, as cache updates are received. If the cache is clustered, the entire cluster wide contents are iterated over.

`includeCurrentState` also controls whether state is received when the node where the client event listener is registered fails and it's moved to a different node. The next section discusses this topic in depth.

6.1.16. Listener Failure Handling

When a Hot Rod client registers a client listener, it does so in a single node in a cluster. If that node fails, the Java Hot Rod client detects that transparently and fails over all listeners registered in the node that failed to another node.

During this fail over the client might miss some events. To avoid missing these events, the client listener annotation contains an optional parameter called `includeCurrentState` which if set to true, when the failover happens, the cache contents can iterated over and `ClientCacheEntryCreated` events (or custom events if configured) are generated. By default, `includeCurrentState` is set to false.

Java Hot Rod clients can be made aware of such fail over event by adding a callback to handle it:

```
@ClientCacheFailover
public void handleFailover(ClientCacheFailoverEvent e) {
  ...
}
```

This is very useful in use cases where the client has cached some data, and as a result of the fail over, taking in account that some events could be missed, it could decide to clear any locally cached data when the fail over event is received, with the knowledge that after the fail over event, it will receive events for the contents of the entire cache.

6.1.17. Near Caching

The Java Hot Rod client can be optionally configured with a near cache, which means that the Hot Rod client can keep a local cache that stores recently used data. Enabling near caching can significantly improve the performance of read operations `get` and `getVersioned` since data can potentially be located locally within the Hot Rod client instead of having to go remote.

To enable near caching, the user must set the near cache mode to `INVALIDATED`. By doing that near cache is populated upon retrievals from the server via calls to `get` or `getVersioned` operations. When near cached entries are updated or removed server-side, the cached near cache entries are invalidated. If a key is requested after it's been invalidated, it'll have to be re-fetched from the server.

You should not use `maxIdle` expiration with near caches, as near-cache reads will not propagate the last access change to the server and to the other clients.

When near cache is enabled, its size must be configured by defining the maximum number of entries to keep in the near cache. When the maximum is reached, near cached entries are evicted using a least-recently-used (LRU) algorithm. If providing 0 or a negative value, it is assumed that the near cache is unbounded.

Users should be careful when configuring near cache to be unbounded since it shifts the responsibility to keep the near cache's size within the boundaries of the client JVM to the user.

The Hot Rod client's near cache mode is configured using the `NearCacheMode` enumeration and calling:

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.configuration.NearCacheMode;
...

// Unbounded invalidated near cache
ConfigurationBuilder unbounded = new ConfigurationBuilder();
unbounded.nearCache().mode(NearCacheMode.INVALIDATED).maxEntries(-1);

// Bounded invalidated near cache
ConfigurationBuilder bounded = new ConfigurationBuilder();
bounded.nearCache().mode(NearCacheMode.INVALIDATED).maxEntries(100);
```

Since the configuration is shared by all caches obtained from a single `RemoteCacheManager`, you may not want to enable near-caching for all of them. You can use the `cacheNamePattern` configuration attribute to define a regular expression which matches the names of the caches for which you want near-caching. Caches whose name don't match the regular expression, will not have near-caching enabled.

```
// Bounded invalidated near cache with pattern matching
ConfigurationBuilder bounded = new ConfigurationBuilder();
bounded.nearCache()
  .mode(NearCacheMode.INVALIDATED)
  .maxEntries(100)
  .cacheNamePattern("near.*"); // enable near-cache only for caches whose name starts
with 'near'
```

Near caches work the same way for local caches as they do for clustered caches, but in a clustered cache scenario, if the server node sending the near cache notifications to the Hot Rod client goes down, the Hot Rod client transparently fails over to another node in the cluster, clearing the near cache along the way.

6.1.18. Unsupported methods

Some of the Cache methods are not being supported by the RemoteCache . Calling one of these methods results in an UnsupportedOperationException being thrown. Most of these methods do not make sense on the remote cache (e.g. listener management operations), or correspond to methods that are not supported by local cache as well (e.g. containsValue). Another set of unsupported operations are some of the atomic operations inherited from ConcurrentMap :

```
boolean remove(Object key, Object value);
boolean replace(Object key, Object value);
boolean replace(Object key, Object oldValue, Object value);
```

RemoteCache offers alternative versioned methods for these atomic operations, that are also network friendly, by not sending the whole value object over the network, but a version identifier. See the section on versioned API.

Each one of these unsupported operation is documented in the RemoteCache javadoc.

6.1.19. Return values

There is a set of methods that alter a cached entry and return the previous existing value, e.g.:

```
V remove(Object key);
V put(K key, V value);
```

By default on RemoteCache, these operations return null even if such a previous value exists. This approach reduces the amount of data sent over the network. However, if these return values are needed they can be enforced on a per invocation basis using flags:

```
cache.put("aKey", "initialValue");
assert null == cache.put("aKey", "aValue");
assert "aValue".equals(cache.withFlags(Flag.FORCE_RETURN_VALUE).put("aKey",
   "newValue"));
```

This default behavior can can be changed through force-return-value=true configuration parameter (see configuration section bellow).

Chapter 7. Hot Rod Transactions

You can configure and use Hot Rod clients in JTA Transactions.

To participate in a transaction, the Hot Rod client requires the TransactionManager with which it interacts and whether it participates in the transaction through the Synchronization or XAResource interface.

Transactions are optimistic in that clients acquire write locks on entries during the prepare phase. To avoid data inconsistency, be sure to read about Detecting Conflicts with Transactions.

7.1. Configuring the Server

Caches in the server must also be transactional for clients to participate in JTA Transactions.

The following server configuration is required, otherwise transactions rollback only:

- Isolation level must be `REPEATABLE_READ`.
- Locking mode must be `PESSIMISTIC`. In a future release, `OPTIMISTIC` locking mode will be supported.
- Transaction mode should be `NON_XA` or `NON_DURABLE_XA`. Hot Rod transactions cannot use `FULL_XA` because it degrades performance.

Hot Rod transactions have their own recovery mechanism.

7.2. Configuring Hot Rod Clients

When you create the RemoteCacheManager, you can set the default TransactionManager and TransactionMode that the RemoteCache uses.

The RemoteCacheManager lets you create only one configuration for transactional caches, as in the following example:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
//other client configuration parameters
cb.transaction().transactionManagerLookup(GenericTransactionManagerLookup.getInstance(
));
cb.transaction().transactionMode(TransactionMode.NON_XA);
cb.transaction().timeout(1, TimeUnit.MINUTES)
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

The preceding configuration applies to all instances of a remote cache. If you need to apply different configurations to remote cache instances, you can override the RemoteCache configuration. See Overriding RemoteCacheManager Configuration.

See ConfigurationBuilder Javadoc for documentation on configuration parameters.

You can also configure the Java Hot Rod client with a properties file, as in the following example:

```
infinispan.client.hotrod.transaction.transaction_manager_lookup =
org.infinispan.client.hotrod.transaction.lookup.GenericTransactionManagerLookup
infinispan.client.hotrod.transaction.transaction_mode = NON_XA
infinispan.client.hotrod.transaction.timeout = 60000
```

7.2.1. TransactionManagerLookup Interface

`TransactionManagerLookup` provides an entry point to fetch a TransactionManager.

Available implementations of `TransactionManagerLookup`:

GenericTransactionManagerLookup

A lookup class that locates TransactionManagers running in Java EE application servers. Defaults to the RemoteTransactionManager if it cannot find a TransactionManager.

In most cases, GenericTransactionManagerLookup is suitable. However, you can implement the `TransactionManagerLookup` interface if you need to integrate a custom TransactionManager.

RemoteTransactionManagerLookup

A basic, and volatile, TransactionManager if no other implementation is available. Note that this implementation has significant limitations when handling concurrent transactions and recovery.

7.2.2. Transaction Modes

TransactionMode controls how a RemoteCache interacts with the TransactionManager.

Configure transaction modes on both the {brandname} server and your client application. If clients attempt to perform transactional operations on non-transactional caches, runtime exceptions can occur.

Transaction modes are the same in both the {brandname} configuration and client settings. Use the following modes with your client, see the {brandname} configuration schema for the server:

`NONE`

The RemoteCache does not interact with the TransactionManager. This is the default mode and is non-transactional.

`NON_XA`

The RemoteCache interacts with the TransactionManager via Synchronization.

`NON_DURABLE_XA`

The RemoteCache interacts with the TransactionManager via XAResource. Recovery capabilities

are disabled.

`FULL_XA`
: The RemoteCache interacts with the TransactionManager via XAResource. Recovery capabilities are enabled. Invoke the `XaResource.recover()` method to retrieve transactions to recover.

7.3. Overriding Configuration for Cache Instances

Because RemoteCacheManager does not support different configurations for each cache instance. However, RemoteCacheManager includes the `getCache(String)` method that returns the RemoteCache instances and lets you override some configuration parameters, as follows:

`getCache(String cacheName, TransactionMode transactionMode)`
: Returns a RemoteCache and overrides the configured TransactionMode.

`getCache(String cacheName, boolean forceReturnValue, TransactionMode transactionMode)`
: Same as previous, but can also force return values for write operations.

`getCache(String cacheName, TransactionManager transactionManager)`
: Returns a RemoteCache and overrides the configured TransactionManager.

`getCache(String cacheName, boolean forceReturnValue, TransactionManager transactionManager)`
: Same as previous, but can also force return values for write operations.

`getCache(String cacheName, TransactionMode transactionMode, TransactionManager transactionManager)`
: Returns a RemoteCache and overrides the configured TransactionManager and TransactionMode. Uses the configured values, if `transactionManager` or `transactionMode` is null.

`getCache(String cacheName, boolean forceReturnValue, TransactionMode transactionMode, TransactionManager transactionManager)`
: Same as previous, but can also force return values for write operations.

The `getCache(String)` method returns RemoteCache instances regardless of whether they are transaction or not. RemoteCache includes a `getTransactionManager()` method that returns the TransactionManager that the cache uses. If the RemoteCache is not transactional, the method returns `null`.

7.4. Detecting Conflicts with Transactions

Transactions use the initial values of keys to detect conflicts. For example, "k" has a value of "v" when a transaction begins. During the prepare phase, the transaction fetches "k" from the server to read the value. If the value has changed, the transaction rolls back to avoid a conflict.

Transactions use versions to detect changes instead of checking value equality.

The `forceReturnValue` parameter controls write operations to the RemoteCache and helps avoid conflicts. It has the following values:

- If `true`, the TransactionManager fetches the most recent value from the server before performing write operations. However, the `forceReturnValue` parameter applies only to write operations that access the key for the first time.
- If `false`, the TransactionManager does not fetch the most recent value from the server before performing write operations. Because this setting

This parameter does not affect *conditional* write operations such as `replace` or `putIfAbsent` because they require the most recent value.

The following transactions provide an example where the `forceReturnValue` parameter can prevent conflicting write operations:

Transaction 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v1");
tm.commit();
```

Transaction 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v2");
tm.commit();
```

In this example, TX1 and TX2 are executed in parallel. The initial value of "k" is "v".

- If `forceReturnValue = true`, the `cache.put()` operation fetches the value for "k" from the server in both TX1 and TX2. The transaction that acquires the lock for "k" first then commits. The other transaction rolls back during the commit phase because the transaction can detect that "k" has a value other than "v".
- If `forceReturnValue = false`, the `cache.put()` operation does not fetch the value for "k" from the server and returns null. Both TX1 and TX2 can successfully commit, which results in a conflict. This occurs because neither transaction can detect that the initial value of "k" changed.

The following transactions include `cache.get()` operations to read the value for "k" before doing the `cache.put()` operations:

Transaction 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v1");
tm.commit();
```

Transaction 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v2");
tm.commit();
```

In the preceding examples, TX1 and TX2 both read the key so the `forceReturnValue` parameter does not take effect. One transaction commits, the other rolls back. However, the `cache.get()` operation requires an additional server request. If you do not need the return value for the `cache.put()` operation that server request is inefficient.

7.5. Using the Configured Transaction Manager and Transaction Mode

The following example shows how to use the `TransactionManager` and `TransactionMode` that you configure in the `RemoteCacheManager`:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//The my-cache instance uses the RemoteCacheManager configuration.
RemoteCache<String, String> cache = rcm.getCache("my-cache");

//Return the transaction manager that the cache uses.
TransactionManager tm = cache.getTransactionManager();

//Perform a simple transaction.
tm.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
tm.commit();
```

7.6. Overriding the Transaction Manager

The following example shows how to override `TransactionManager` with the `getCache` method:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//Define a custom TransactionManager.
TransactionManager myCustomTM = ...

//Override the TransactionManager for the my-cache instance. Use the default
configuration if null is returned.
RemoteCache<String, String> cache = rcm.getCache("my-cache", null, myCustomTM);

//Perform a simple transaction.
myCustomTM.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
myCustomTM.commit();
```

7.7. Overriding the Transaction Mode

The following example shows how to override `TransactionMode` with the `getCache` method:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//Override the transaction mode for the my-cache instance.
RemoteCache<String, String> cache = rcm.getCache("my-cache", TransactionMode
.NON_DURABLE_XA, null);

//Return the transaction manager that the cache uses.
TransactionManager tm = cache.getTransactionManager();

//Perform a simple transaction.
tm.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
tm.commit();
```

7.7.1. Client Intelligence

HotRod defines three level of intelligence for the clients:

1. basic client, interested in neither cluster nor hash information
2. topology-aware client, interested in cluster information
3. hash-distribution-aware client, that is interested in both cluster and hash information

The java client supports all 3 levels of intelligence. It is transparently notified whenever a new server is added/removed from the HotRod cluster. At startup it only needs to know the address of one HotRod server (ip:host). On connection to the server the cluster topology is piggybacked to the client, and all further requests are being dispatched to all available servers. Any further topology change is also piggybacked.

Distribution-aware client

Another aspect of the 3rd level of intelligence is the fact that it is hash-distribution-aware. This means that, for each operation, the client chooses the most appropriate remote server to go to: the data owner. As an example, for a put(k,v) operation, the client calculates k's hash value and knows exactly on which server the data resides on. Then it picks up a tcp connection to that particular server and dispatches the operation to it. This means less burden on the server side which would otherwise need to lookup the value based on the key's hash. It also results in a quicker response

from the server, as an additional network roundtrip is skipped. This hash-distribution-aware aspect is only relevant to the distributed HotRod clusters and makes no difference for replicated server deployments.

7.7.2. Request Balancing

Request balancing is only relevant when the server side is configured with replicated {brandname} cluster (on distributed clusters the hash-distribution-aware client logic is used, as discussed in the previos paragraph). Because the client is topology-aware, it knows the list of available servers at all the time. Request balancing has to do with how the client dispatches requests to the available servers.

The default strategy is round-robin: requests are being dispatched to all existing servers in a circular manner. E.g. given a cluster of servers {s1, s2, s3} here is how request will be dispatched:

```
CacheContainer cacheContainer = new RemoteCacheManager();
Cache<String, String> cache = cacheContainer.getCache();

cache.put("key1", "aValue"); //this goes to s1
cache.put("key2", "aValue"); //this goes to s2
String value = cache.get("key1"); //this goes to s3

cache.remove("key2"); //this is dispatched to s1 again, and so on...
```

Custom types of balancing policies can defined by implementing the FailoverRequestBalancingStrategy and by specifying it through the infinispan.client.hotrod.request-balancing-strategy configuration property. Please refer to configuration section for more details on this.

7.7.3. Persistent connections

In order to avoid creating a TCP connection on each request (which is a costly operation), the client keeps a pool of persistent connections to all the available servers and it reuses these connections whenever it is possible. The validity of the connections is checked using an async thread that iterates over the connections in the pool and sends a HotRod ping command to the server. By using this connection validation process the client is being proactive: there's a hight chance for broken connections to be found while being idle in the pool and no on actual request from the application.

The number of connections per server, total number of connections, how long should a connection be kept idle in the pool before being closed - all these (and more) can be configured. Please refer to the javadoc of RemoteCacheManager for a list of all possible configuration elements.

7.7.4. Marshalling data

The Hot Rod client allows one to plug in a custom marshaller for transforming user objects into byte arrays and the other way around. This transformation is needed because of Hot Rod's binary nature - it doesn't know about objects.

The marshaller can be plugged through the "marshaller" configuration element (see Configuration section): the value should be the fully qualified name of a class implementing the Marshaller interface. This is a optional parameter, if not specified it defaults to the GenericJBossMarshaller - a highly optimized implementation based on the JBoss Marshalling library.

Since version 6.0, there's a new marshaller available to Java Hot Rod clients based on Protostream which generates portable payloads.

WARNING: If developing your own custom marshaller, take care of potential injection attacks.

To avoid such attacks, make the marshaller verify that any class names read, before instantiating it, is amongst the expected/allowed class names.

The client configuration can be enhanced with a list of regular expressions for classes that are allowed to be read.

WARNING: These checks are opt-in, so if not configured, any class can be read.

In the example below, only classes with fully qualified names containing `Person` or `Employee` would be allowed:

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;

...
ConfigurationBuilder configBuilder = ...
configBuilder.addJavaSerialWhiteList(".*Person.*", ".*Employee.*");
```

7.7.5. Reading data in different data formats

By default, every Hot Rod client operation will use the configured marshaller when reading and writing from the server for both keys and values. See Marshalling Data. Using the DataFormat API, it's possible to decorate remote caches so that all operations can happen with a custom data format.

Using different marshallers for Key and Values

Marshallers for Keys and Values can be overridden at run time. For example, to bypass all serialization in the Hot Rod client and read the byte[] as they are stored in the server:

```
// Existing Remote cache instance
RemoteCache<String, Pojo> remoteCache = ...

// IdentityMarshaller is a no-op marshaller
DataFormat rawKeyAndValues = DataFormat.builder().keyMarshaller(IdentityMarshaller
.INSTANCE).valueMarshaller(IdentityMarshaller.INSTANCE).build();

// Will create a new instance of RemoteCache with the supplied DataFormat
RemoteCache<byte[], byte[]> rawResultsCache = remoteCache.withDataFormat
(rawKeyAndValues);
```

Reading data in different formats

Apart from defining custom key and value marshallers, it's also possible to request/send data in different formats specified by a `org.infinispan.commons.dataconversion.MediaType`:

```
// Existing remote cache using ProtostreamMarshaller
RemoteCache<String, Pojo> protobufCache = ...

// Request values returned as JSON, using the UTF8StringMarshaller that converts
between UTF-8 to String:
DataFormat jsonString = DataFormat.builder().valueType(MediaType.APPLICATION_JSON)
.valueMarshaller(new UTF8StringMarshaller().build();

RemoteCache<String, String> jsonStrCache = remoteCache.withDataFormat(jsonString);

// Alternativelly, it's possible to request JSON values but marshalled/unmarshalled
with a custom value marshaller that returns `org.codehaus.jackson.JsonNode` objects:
DataFormat jsonNode = DataFormat.builder().valueType(MediaType.APPLICATION_JSON)
.valueMarshaller(new CustomJacksonMarshaller().build();

RemoteCache<String, JsonNode> jsonNodeCache = remoteCache.withDataFormat(jsonNode);
```

The data conversions happen in the server, and if it doesn't support converting from the storage format to the request format and vice versa, an error will be returned.

Using different marshallers and formats for the key, with `.keyMarshaller()` and `.keyType()` may interfere with the client intelligence routing mechanism, causing it contact the server that is not the owner of the key during Hot Rod operations. This will not result in errors but can result in extra hops inside the cluster to execute the operation. If performance is critical, make sure to use the keys in the format stored by the server.

7.7.6. Statistics

Various server usage statistics can be obtained through the RemoteCache .stats() method. This returns a ServerStatistics object - please refer to javadoc for details on the available statistics.

7.7.7. Multi-Get Operations

The Java Hot Rod client does not provide multi-get functionality out of the box but clients can build it themselves with the given APIs.

7.7.8. Failover capabilities

Hot Rod clients' capabilities to keep up with topology changes helps with request balancing and more importantly, with the ability to failover operations if one or several of the servers fail.

Some of the conditional operations mentioned above, including `putIfAbsent`, `replace` with and without version, and conditional `remove` have strict method return guarantees, as well as those operations where returning the previous value is forced.

In spite of failures, these methods return values need to be guaranteed, and in order to do so, it's necessary that these methods are not applied partially in the cluster in the event of failure. For example, imagine a `replace()` operation called in a server for key=k1 with `Flag.FORCE_RETURN_VALUE`, whose current value is `A` and the replace wants to set it to `B`. If the replace fails, it could happen that some servers contain `B` and others contain `A`, and during the failover, the original `replace()` could end up returning `B`, if the replace failovers to a node where `B` is set, or could end up returning `A`.

To avoid this kind of situations, whenever Java Hot Rod client users want to use conditional operations, or operations whose previous value is required, it's important that the cache is configured to be transactional in order to avoid incorrect conditional operations or return values.

7.7.9. Site Cluster Failover

On top of the in-cluster failover, Hot Rod clients are also able to failover to different clusters, which could be represented as an independent site.

The way site cluster failover works is that if all the main cluster nodes are not available, the client checks to see if any other clusters have been defined in which cases it tries to failover to the alternative cluster. If the failover succeeds, the client will remain connected to the alternative cluster until this becomes unavailable, in which case it'll try any other clusters defined, and ultimately, it'll try the original server settings.

To configure a cluster in the Hot Rod client, one host/port pair details must be provided for each of the clusters configured. For example:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
      = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.addCluster().addClusterNode("remote-cluster-host", 11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

Remember that regardless of the cluster definitions, the initial server(s) configuration must be provided unless the initial servers can be resolved using the default server host and port details.

7.7.10. Manual Site Cluster Switch

As well as supporting automatic site cluster failover, Java Hot Rod clients can also switch between site clusters manually by calling RemoteCacheManager's `switchToCluster(clusterName)` and `switchToDefaultCluster()`.

Using `switchToCluster(clusterName)`, users can force a client to switch to one of the clusters pre-defined in the Hot Rod client configuration. To switch to the initial servers defined in the client configuration, call `switchToDefaultCluster()`.

7.7.11. Monitoring the Hot Rod client

The Hot Rod client can be monitored and managed via JMX. By enabling statistics, an MBean will be registered for the `RemoteCacheManager` as well as for each `RemoteCache` obtained through it. Through these MBeans it is possible to obtain statistics about remote and near-cache hits/misses and connection pool usage.

7.7.12. Concurrent Updates

Data structures, such as {brandname} Cache , that are accessed and modified concurrently can suffer from data consistency issues unless there're mechanisms to guarantee data correctness. {brandname} Cache, since it implements ConcurrentMap , provides operations such as conditional replace , putIfAbsent , and conditional remove to its clients in order to guarantee data correctness. It even allows clients to operate against cache instances within JTA transactions, hence providing the necessary data consistency guarantees.

However, when it comes to Hot Rod protocol backed servers, clients do not yet have the ability to start remote transactions but they can call instead versioned operations to mimic the conditional methods provided by the embedded {brandname} cache instance API. Let's look at a real example to understand how it works.

Data Consistency Problem

Imagine you have two ATMs that connect using Hot Rod to a bank where an account's balance is stored. Two closely followed operations to retrieve the latest balance could return 500 CHF (swiss francs) as shown below:

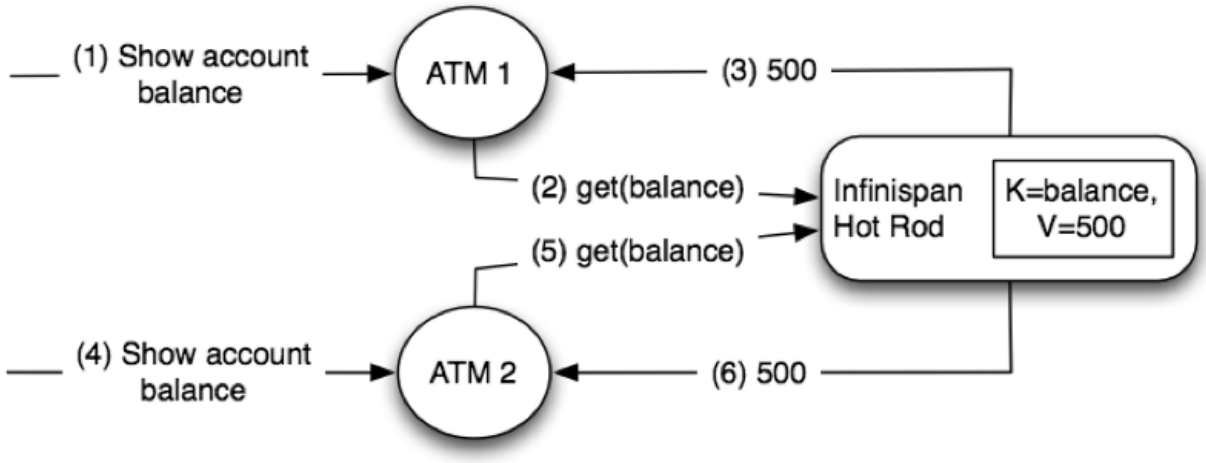
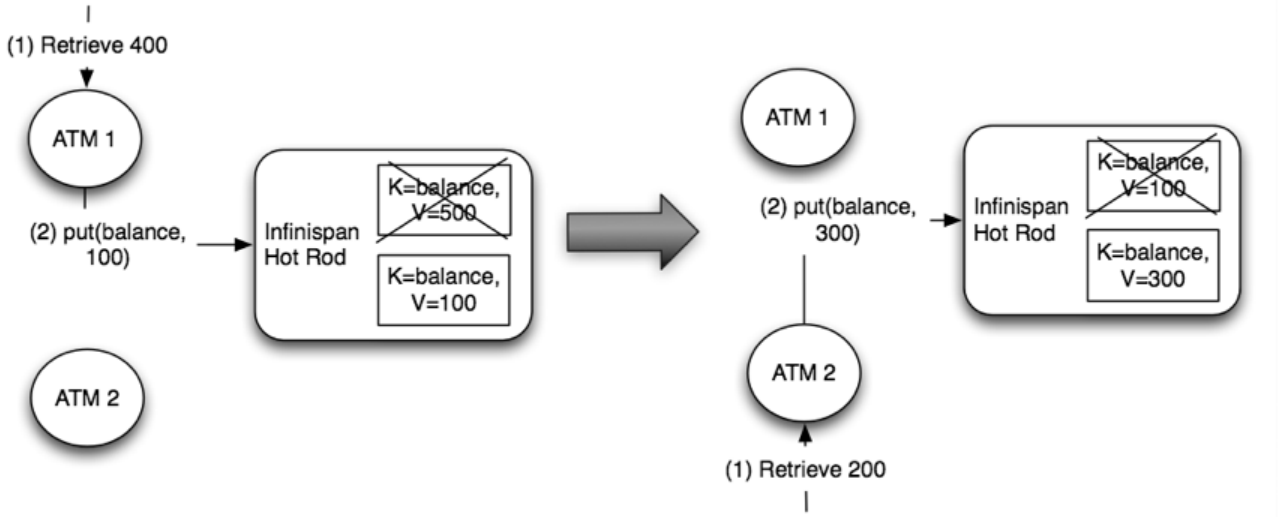


Figure 8. Concurrent readers

Next a customer connects to the first ATM and requests 400 CHF to be retrieved. Based on the last value read, the ATM could calculate what the new balance is, which is 100 CHF, and request a put with this new value. Let's imagine now that around the same time another customer connects to the ATM and requests 200 CHF to be retrieved. Let's assume that the ATM thinks it has the latest balance and based on its calculations it sets the new balance to 300 CHF:



Obviously, this would be wrong. Two concurrent updates have resulted in an incorrect account balance. The second update should not have been allowed since the balance the second ATM had was incorrect. Even if the ATM would have retrieved the balance before calculating the new balance, someone could have updated between the new balance being retrieved and the update. Before finding out how to solve this issue in a client-server scenario with Hot Rod, let's look at how this is solved when {brandname} clients run in peer-to-peer mode where clients and {brandname} live within the same JVM.

Embedded-mode Solution

If the ATM and the {brandname} instance storing the bank account lived in the same JVM, the ATM could use the conditional replace API referred at the beginning of this article. So, it could send the previous known value to verify whether it has changed since it was last read. By doing so, the first operation could double check that the balance is still 500 CHF when it was to update to 100 CHF. Now, when the second operation comes, the current balance would not be 500 CHF any more and hence the conditional replace call would fail, hence avoiding data consistency issues:

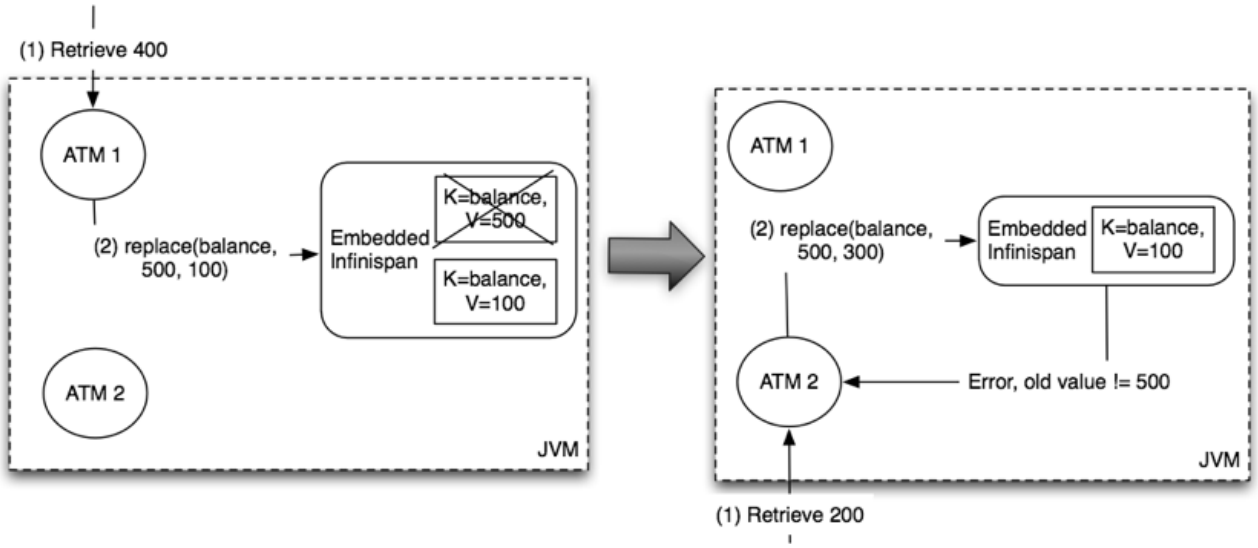


Figure 9. P2P solution

Client-Server Solution

In theory, Hot Rod could use the same p2p solution but sending the previous value would be not practical. In this example, the previous value is just an integer but the value could be a lot bigger and hence forcing clients to send it to the server would be rather wasteful. Instead, Hot Rod offers versioned operations to deal with this situation.

Basically, together with each key/value pair, Hot Rod stores a version number which uniquely identifies each modification. So, using an operation called getVersioned or getWithVersion , clients can retrieve not only the value associated with a key, but also the current version. So, if we look at the previous example once again, the ATMs could call getVersioned and get the balance's version:

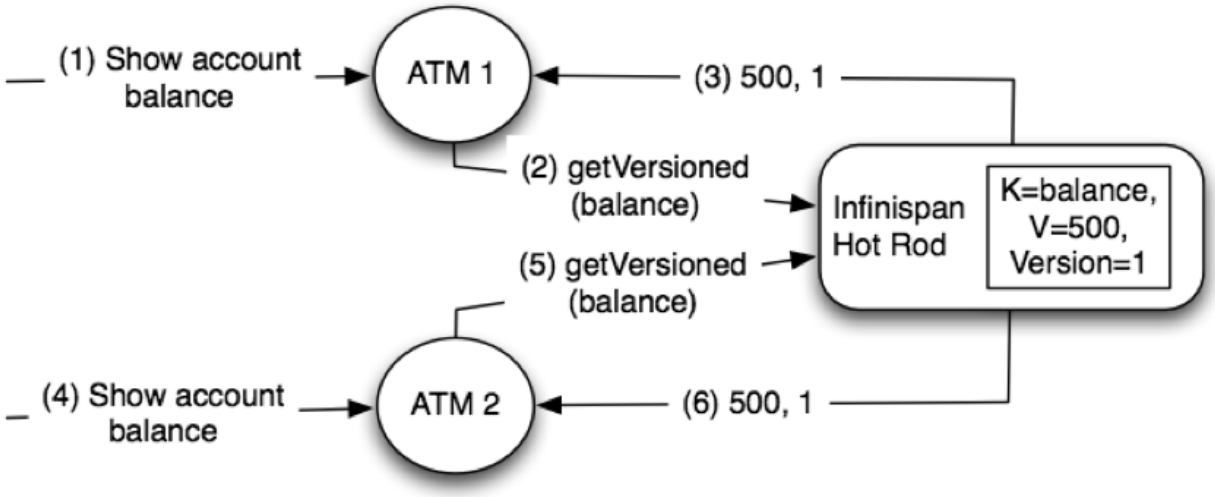


Figure 10. Get versioned

When the ATMs wanted to modify the balance, instead of just calling put, they could call replaceIfUnmodified operation passing the latest version number of which the clients are aware of. The operation will only succeed if the version passed matches the version in the server. So, the first modification by the ATM would be allowed since the client passes 1 as version and the server side version for the balance is also 1. On the other hand, the second ATM would not be able to make the modification because after the first ATMs modification the version would have been incremented to 2, and now the passed version (1) and the server side version (2) would not match:

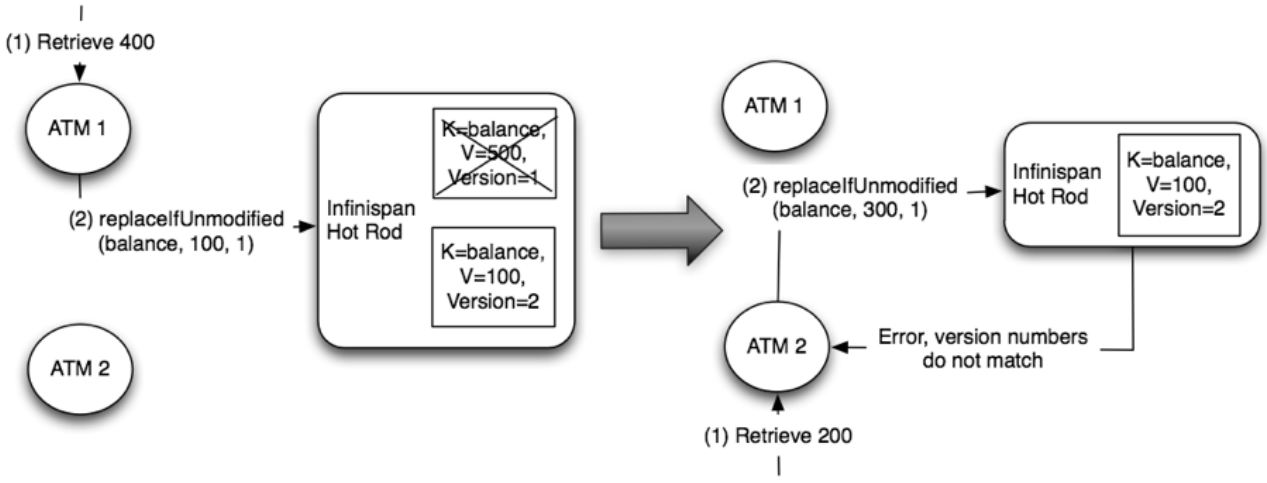


Figure 11. Replace if versions match

7.7.13. Javadocs

It is highly recommended to read the following Javadocs (this is pretty much all the public API of the client):

- RemoteCacheManager
- RemoteCache

Chapter 8. REST Server

The {brandname} Server distribution contains a module that implements RESTful HTTP access to the {brandname} data grid, built on Netty.

8.1. Running the REST server

The REST server endpoint is part of the {brandname} Server and by default listens on port 8080. To run the server locally, download the zip distribution and execute in the extracted directory:

```
bin/standalone.sh -b 0.0.0.0
```

or alternatively, run via docker:

```
docker run -it -p 8080:8080 -e "APP_USER=user" -e "APP_PASS=changeme"
jboss/infinispan-server
```

8.1.1. Security

The REST server is protected by authentication, so before usage it is necessary to create an application login. When running via docker, this is achieved by the APP_USER and APP_PASS command line arguments, but when running locally, this can be done with:

```
bin/add-user.sh -u user -p changeme -a
```

8.2. Supported protocols

The REST Server supports HTTP/1.1 as well as HTTP/2 protocols. It is possible to switch to HTTP/2 by either performing a HTTP/1.1 Upgrade procedure or by negotiating communication protocol using TLS/ALPN extension.

Note: TLS/ALPN with JDK8 requires additional steps from the client perspective. Please refer to your client documentation but it is very likely that you will need Jetty ALPN Agent or OpenSSL bindings.

8.3. CORS

The REST server supports CORS including preflight and rules based on the request origin.

Example:

```
<rest-connector name="rest1" socket-binding="rest" cache-container="default">
   <cors-rules>
      <cors-rule name="restrict host1" allow-credentials="false">
         <allowed-origins>http://host1,https://host1</allowed-origins>
         <allowed-methods>GET</allowed-methods>
      </cors-rule>
      <cors-rule name="allow ALL" allow-credentials="true" max-age-seconds="2000">
         <allowed-origins>*</allowed-origins>
         <allowed-methods>GET,OPTIONS,POST,PUT,DELETE</allowed-methods>
         <allowed-headers>Key-Content-Type</allowed-headers>
      </cors-rule>
   </cors-rules>
</rest-connector>
```

The rules are evaluated sequentially based on the "Origin" header set by the browser; in the example above if the origin is either "http://host1" or "https://host1" the rule "restrict host1" will apply, otherwise the next rule will be tested. Since the rule "allow ALL" permits all origins, any script coming from a different origin will be able to perform the methods specified and use the headers supplied.

The <cors-rule> element can be configured as follows:

Config	Description	Mandatory
name	The name of the rule	yes
allow-credentials	Enable CORS requests to use credentials	no
allowed-origins	A comma separated list used to set the CORS 'Access-Control-Allow-Origin' header to indicate the response can be shared with the origins	yes
allowed-methods	A comma separated list used to set the CORS 'Access-Control-Allow-Methods' header in the preflight response to specify the methods allowed for the configured origin(s)	yes
max-age-seconds	The amount of time CORS preflight request headers can be cached	no
expose-headers	A comma separated list used to set the CORS 'Access-Control-Expose-Headers' in the preflight response to specify which headers can be exposed to the configured origin(s)	no

8.4. Data formats

8.4.1. Configuration

Each cache exposed via REST stores data in a configurable data format defined by a MediaType. More details in the configuration here.

An example of storage configuration is as follows:

```
<cache>
   <encoding>
      <key media-type="application/x-java-object; type=java.lang.Integer"/>
      <value media-type="application/xml; charset=UTF-8"/>
   </encoding>
</cache>
```

When no MediaType is configured, {brandname} assumes "application/octet-stream" for both keys and values, with the following exceptions:

- If the cache is indexed, it assumes "application/x-protostream"

8.4.2. Supported formats

Data can be written and read in different formats than the storage format; {brandname} can convert between those formats when required.

The following "standard" formats can be converted interchangeably:

- *application/x-java-object*
- *application/octet-stream*
- *application/x-www-form-urlencoded*
- *text/plain*

The following formats can be converted to/from the formats above:

- *application/xml*
- *application/json*
- *application/x-jboss-marshalling*
- *application/x-protostream*
- *application/x-java-serialized*

Finally, the following conversion is also supported:

- Between *application/x-protostream* and *application/json*

All the REST API calls can provide headers describing the content written or the required format of

the content when reading. {brandname} supports the standard HTTP/1.1 headers "Content-Type" and "Accept" that are applied for values, plus the "Key-Content-Type" with similar effect for keys.

8.4.3. Accept header

The REST server is compliant with the RFC-2616 Accept header, and will negotiate the correct MediaType based on the conversions supported. Example, sending the following header when reading data:

```
Accept: text/plain;q=0.7, application/json;q=0.8, */*;q=0.6
```

will cause {brandname} to try first to return content in JSON format (higher priority 0.8). If it's not possible to convert the storage format to JSON, next format tried will be *text/plain* (second highest priority 0.7), and finally it falls back to */*, that will pick a format suitable for displaying automatically based on the cache configuration.

8.4.4. Key-Content-Type header

Most REST API calls have the Key included in the URL. {brandname} will assume the Key is a *java.lang.String* when handling those calls, but it's possible to use a specific header *Key-Content-Type* for keys in different formats.

Examples:

- Specifying a byte[] Key as a Base64 string:

API call:

```
`PUT /my-cache/AQIDBDM=`
```

Headers:

```
Key-Content-Type: application/octet-stream
```

- Specifying a byte[] Key as a hexadecimal string:

API call:

```
GET /my-cache/0x01CA03042F
```

Headers:

```
Key-Content-Type: application/octet-stream; encoding=hex
```

- Specifying a double Key:

API call:

```
POST /my-cache/3.141456
```

Headers:

```
Key-Content-Type: application/x-java-object;type=java.lang.Double
```

The *type* parameter for *application/x-java-object* is restricted to:

- Primitive wrapper types
- java.lang.String
- Bytes, making *application/x-java-object;type=Bytes* equivalent to *application/octet-stream;encoding=hex*

8.4.5. JSON/Protostream conversion

When caches are indexed, or specifically configured to store *application/x-protostream*, it's possible to send and receive JSON documents that are automatically converted to/from protostream. In order for the conversion to work, a protobuf schema must be registered.

The registration can be done via REST, by doing a POST/PUT in the *___protobuf_metadata* cache. Example using cURL:

```
curl -u user:password -X POST --data-binary @./schema.proto
http://127.0.0.1:8080/rest/___protobuf_metadata/schema.proto
```

When writing a JSON document, a special field ***_type*** must be present in the document to identity the protobuf *Message* corresponding to the document.

For example, consider the following schema:

```
message Person  {
  required string name = 1;
  required int32 age = 2;
}
```

A conformant JSON document would be:

```
{
   "_type": "Person",
   "name": "user1",
   "age": 32
}
```

8.5. REST V1 API

The REST V1 API supports basic cache capabilities including operations on keys and query, and is now deprecated. For a more powerful and comprehensive API, check the REST V2 API.

HTTP PUT and POST methods are used to place data in the cache, with URLs to address the cache name and key(s) - the data being the body of the request (the data can be anything you like). Other headers are used to control the cache settings and behaviour.

8.5.1. Putting data in

`PUT /{cacheName}/{cacheKey}`

A PUT request of the above URL form will place the payload (body) in the given cache, with the given key (the named cache must exist on the server). For example `http://someserver/hr/payRoll-3` (in which case `hr` is the cache name, and `payRoll-3` is the key). Any existing data will be replaced, and Time-To-Live and Last-Modified values etc will updated (if applicable).

`POST /{cacheName}/{cacheKey}`

Exactly the same as PUT, only if a value in a cache/key already exists, it will return a Http CONFLICT status (and the content will not be updated).

Headers

- Key-Content-Type: OPTIONAL The content type for the Key present in the URL.
- Content-Type : OPTIONAL The MediaType of the Value being sent.
- timeToLiveSeconds : OPTIONAL number (the number of seconds before this entry will automatically be deleted). If no parameter is sent, {brandname} assumes configuration default value. Passing any negative value will create an entry which will live forever.
- maxIdleTimeSeconds : OPTIONAL number (the number of seconds after last usage of this entry when it will automatically be deleted). If no parameter is sent, {brandname} configuration default value. Passing any negative value will create an entry which will live forever.

Passing 0 as parameter for timeToLiveSeconds and/or maxIdleTimeSeconds

- If both `timeToLiveSeconds` and `maxIdleTimeSeconds` are 0, the cache will use the default `lifespan` and `maxIdle` values configured in XML/programmatically
- If *only* `maxIdleTimeSeconds` is 0, it uses the `timeToLiveSeconds` value passed as parameter (or -1 if not present), and default `maxIdle` configured in XML/programmatically
- If *only* `timeToLiveSeconds` is 0, it uses default `lifespan` configured in XML/programmatically, and `maxIdle` is set to whatever came as parameter (or -1 if not present)

8.5.2. Getting data back out

HTTP GET and HEAD are used to retrieve data from entries.

`GET /{cacheName}/{cacheKey}`

This will return the data found in the given cacheName, under the given key - as the body of the response. A Content-Type header will be present in the response according to the Media Type negotiation. Browsers can use the cache directly of course (eg as a CDN). An ETag will be returned unique for each entry, as will the Last-Modified and Expires headers field indicating the state of the data at the given URL. ETags allow browsers (and other clients) to ask for data only in the case where it has changed (to save on bandwidth) - this is standard HTTP and is honoured by {brandname}.

Headers

- Key-Content-Type: OPTIONAL The content type for the Key present in the URL. When omitted, *application/x-java-object; type=java.lang.String* is assumed
- Accept: OPTIONAL The required format to return the content

It is possible to obtain additional information by appending the "extended" parameter on the query string, as follows:

`GET /cacheName/cacheKey?extended`

This will return the following custom headers:

- Cluster-Primary-Owner: the node name of the primary owner for this key
- Cluster-Node-Name: the JGroups node name of the server that has handled the request
- Cluster-Physical-Address: the physical JGroups address of the server that has handled the request.

`HEAD /{cacheName}/{cacheKey}`

The same as GET, only no content is returned (only the header fields). You will receive the same content that you stored. E.g., if you stored a String, this is what you get back. If you stored some XML or JSON, this is what you will receive. If you stored a binary (base 64 encoded) blob, perhaps a serialized; Java; object - you will need to; deserialize this yourself.

Similarly to the GET method, the HEAD method also supports returning extended information via headers. See above.

Headers

- Key-Content-Type: OPTIONAL The content type for the Key present in the URL. When omitted, *application/x-java-object; type=java.lang.String* is assumed

8.5.3. Listing keys

`GET /{cacheName}`

This will return a list of keys present in the given cacheName as the body of the response. The format of the response can be controlled via the Accept header as follows:

- *application/xml* - the list of keys will be returned in XML format.
- *application/json* - the list of keys will be return in JSON format.

- *text/plain* - the list of keys will be returned in plain text format, one key per line

If the cache identified by cacheName is distributed, only the keys owned by the node handling the request will be returned. To return all keys, append the "global" parameter to the query, as follows:

```
GET /cacheName?global
```

8.5.4. Removing data

Data can be removed at the cache key/element level, or via a whole cache name using the HTTP delete method.

```
DELETE /{cacheName}/{cacheKey}
```

Removes the given key name from the cache.

Headers

- Key-Content-Type: OPTIONAL The content type for the Key present in the URL. When omitted, *application/x-java-object; type=java.lang.String* is assumed

```
DELETE /{cacheName}
```

Removes ALL the entries in the given cache name (i.e., everything from that path down). If the operation is successful, it returns 200 code.

8.5.5. Querying

The REST server supports Ickle Queries in JSON format. It's important that the cache is configured with *application/x-protostream* for both Keys and Values. If the cache is indexed, no configuration is needed.

```
GET /{cacheName}?action=search&query={ickle query}
```

Will execute an Ickle query in the given cache name.

Request parameters

- *query*: REQUIRED the query string
- *max_results*: OPTIONAL the number of results to return, default is *10*
- *offset*: OPTIONAL the index of the first result to return, default is *0*
- *query_mode*: OPTIONAL the execution mode of the query once it's received by server. Valid values are *FETCH* and *BROADCAST*. Default is *FETCH*.

Query Result

Results are JSON documents containing one or more hits. Example:

```
{
  "total_results" : 150,
  "hits" : [ {
    "hit" : {
      "name" : "user1",
      "age" : 35
    }
  }, {
    "hit" : {
       "name" : "user2",
       "age" : 42
    }
  }, {
    "hit" : {
       "name" : "user3",
       "age" : 12
    }
  } ]
}
```

- *total_results*: NUMBER, the total number of results from the query.
- *hits*: ARRAY, list of matches from the query
- *hit*: OBJECT, each result from the query. Can contain all fields or just a subset of fields in case a *Select* clause is used.

`POST /{cacheName}?action=search`

Similar to que query using GET, but the body of the request is used instead to specify the query parameters.

Example:

```
{
 "query":"from Entity where name:\"user1\"",
 "max_results":20,
 "offset":10
}
```

8.6. REST v2 API

{brandname} provides a REST v2 (version 2) API that improves upon the REST v1 API. The REST v2 API gives you all the features of the v1 API in addition to support for resources beyond caching.

8.6.1. Working with Caches

Use the REST API to create and manage caches on your {brandname} cluster and interact with cached entries.

Creating Caches

To create a named cache across the {brandname} cluster, invoke a `POST` request.

```
POST /v2/caches/{cacheName}
```

To configure the cache, you supply the configuration in XML or JSON format as part of the payload.

XML Configuration

If you supply the {brandname} configuration in XML format, it must conform to the schema and include the `<infinispan>` root element and a `<cache-container>` definition, as in the following example:

```
<infinispan>
    <cache-container>
        <distributed-cache name="cacheName" mode="SYNC">
            <memory>
                <object size="20"/>
            </memory>
        </distributed-cache>
    </cache-container>
</infinispan>
```

JSON Configuration

If you supply the {brandname} configuration in a JSON payload, it requires only the cache definition. However, the JSON payload must follow the structure of an XML configuration. XML elements become JSON objects. XML attributes become JSON fields.

For example, the preceding XML configuration is represented in JSON as follows:

```
{
  "distributed-cache": {
    "mode": "SYNC",
    "memory": {
      "object": {
        "size": 20
      }
    }
  }
}
```

Table 1. Headers

Header	Required or Optional	Parameter
`Content-Type`	REQUIRED	Sets the MediaType for the {brandname} configuration payload; either `application/xml` or `application/json`.

Creating with Templates

To create a cache across the {brandname} cluster based on a pre-defined template, invoke a `POST` request with no payload and an extra request parameter:

```
POST /v2/caches/{cacheName}?template={templateName}
```

Retrieving Cache Configuration

To retrieve a {brandname} cache configuration, invoke a `GET` request.

```
GET /v2/configurations/{name}
```

Table 2. Headers

Header	Required or Optional	Parameter
`Accept`	OPTIONAL	Sets the required format to return content. Supported formats are `application/xml` and `application/json`. The default is `application/json`. See Accept for more information.

Adding Entries

To add entries to a named cache, invoke a `POST` request.

```
POST /v2/caches/{cacheName}/{cacheKey}
```

The preceding request places the payload, or request body, in the `cacheName` cache with the `cacheKey` key. The request replaces any data that already exists and updates the `Time-To-Live` and `Last-Modified` values, if they apply.

If the specified key has an existing value, the request returns an HTTP `CONFLICT` status and the value is not updated. In this case, you should use a `PUT` request. See Replacing Entries.

Table 3. Headers

Header	Required or Optional	Parameter
`Key-Content-Type`	OPTIONAL	Sets the content type for the key in the request. See Key-Content-Type for more information.
`Content-Type`	OPTIONAL	Sets the MediaType of the value for the key.
`timeToLiveSeconds`	OPTIONAL	Sets the number of seconds before the entry is automatically deleted. If you do not set this parameter, {brandname} uses the default value from the configuration. If you set a negative value, the entry is never deleted.
`maxIdleTimeSeconds`	OPTIONAL	Sets the number of seconds that entries can be idle. If a read or write operation does not occur for an entry after the maximum idle time elapses, the entry is automatically deleted. If you do not set this parameter, {brandname} uses the default value from the configuration. If you set a negative value, the entry is never deleted.

If both `timeToLiveSeconds` and `maxIdleTimeSeconds` have a value of `0`, {brandname} uses the default `lifespan` and `maxIdle` values from the configuration.

If *only* `maxIdleTimeSeconds` has a value of `0`, {brandname} uses:

- the default `maxIdle` value from the configuration.
- the value for `timeToLiveSeconds` that you pass as a request parameter or a value of `-1` if you do not pass a value.

If *only* `timeToLiveSeconds` has a value of `0`, {brandname} uses:

- the default `lifespan` value from the configuration.
- the value for `maxIdle` that you pass as a request parameter or a value of `-1` if you do not pass a value.

Replacing Entries

To replace entries in a named cache, invoke a `PUT` request.

```
PUT /v2/caches/{cacheName}/{cacheKey}
```

The preceding request is the same as a `POST` request to add entries to the cache. However, if the entry already exists, the `PUT` request replaces it instead of returning an HTTP `CONFLICT` status.

Retrieving Caches By Keys

To retrieve data for a specific key in a cache, invoke a `GET` request.

```
GET /v2/caches/{cacheName}/{cacheKey}
```

The preceding request returns data from the given cache, `cacheName`, under the given key, `cacheKey`, as the response body. Responses contain a `Content-Type` headers that correspond to the MediaType negotiation.

Browsers can also access caches directly, for example as a content delivery network (CDN). {brandname} returns a unique ETag for each entry along with the `Last-Modified` and `Expires` header fields. These fields provide information about the state of the data that is returned in your request. ETags allow browsers and other clients to request only data that has changed, which conserves bandwidth.

Table 4. Headers

Header	Required or Optional	Parameter
`Key-Content-Type`	OPTIONAL	Sets the content type for the key in the request. The default is `application/x-java-object; type=java.lang.String`. See Key-Content-Type for more information.
`Accept`	OPTIONAL	Sets the required format to return content. See Accept for more information.

Append the `extended` parameter to the query string to get additional information.

```
GET /cacheName/cacheKey?extended
```

The preceding request returns custom headers:

- `Cluster-Primary-Owner` returns the node name that is the primary owner of the key.
- `Cluster-Node-Name` returns the JGroups node name of the server that handled the request.
- `Cluster-Physical-Address` returns the physical JGroups address of the server that handled the request.

Checking if Entries Exist

To check if a specific entry exists in a cache, invoke a `HEAD` request.

```
HEAD /v2/caches/{cacheName}/{cacheKey}
```

The preceding request returns only the header fields and the same content that you stored with the entry. For example, if you stored a String, the request returns a String. If you stored binary, base64-encoded, blobs or serialized Java objects, {brandname} does not de-serialize the content in the request.

As with `GET` requests, `HEAD` requests also support the `extended` parameter.

Table 5. Headers

Header	Required or Optional	Parameter
`Key-Content-Type`	OPTIONAL	Sets the content type for the key in the request. The default is `application/x-java-object; type=java.lang.String`. See Key-Content-Type for more information.

Deleting Entries

To delete entries from a cache, invoke a `DELETE` request.

```
DELETE /v2/caches/{cacheName}/{cacheKey}
```

The preceding request removes the entry under `cacheKey` name from the cache.

Table 6. Headers

Header	Required or Optional	Parameter
`Key-Content-Type`	OPTIONAL	Sets the content type for the key in the request. The default is `application/x-java-object; type=java.lang.String`. See Key-Content-Type for more information.

Removing Caches

To remove caches, invoke a `DELETE` request.

```
DELETE /v2/caches/{cacheName}
```

The preceding request deletes all data and removes the cache named `cacheName` from the cluster.

Clearing Caches

To delete all data from a cache, invoke a `GET` request with the `?action=clear` parameter.

```
GET /v2/caches/{cacheName}?action=clear
```

Getting the size of Caches

To obtain the size of the cache across the entire cluster, invoke a `GET` request with the `?action=size` parameter.

```
GET /v2/caches/{cacheName}?action=size
```

Querying Caches

Invoke a `GET` request to perform and Ickle query on a given cache, as follows:

```
GET /v2/caches/{cacheName}?action=search&query={ickle query}
```

The preceding request returns a `JSON` document that contains one or more query hits, for example:

```
{
  "total_results" : 150,
  "hits" : [ {
    "hit" : {
      "name" : "user1",
      "age" : 35
    }
  }, {
    "hit" : {
       "name" : "user2",
       "age" : 42
    }
  }, {
    "hit" : {
       "name" : "user3",
       "age" : 12
    }
  } ]
}
```

- `total_results` displays the total number of results from the query.
- `hits` is an array of matches from the query.
- `hit` is an object that matches the query. Each hit can contain all fields or a subset of fields if you use a `Select` clause.

Table 7. Request Parameters

Parameter	Required or Optional	Value
`query`	REQUIRED	Specifies the query string.
`max_results`	OPTIONAL	Sets the number of results to return. The default is `10`.
`offset`	OPTIONAL	Specifies the index of the first result to return. The default is `0`.
`query_mode`	OPTIONAL	Specifies how the {brandname} server executes the query. Values are `FETCH` and `BROADCAST`. The default is `FETCH`.

To use the body of the request instead of specifying query parameters, invoke a `POST` request.

```
POST /v2/caches/{cacheName}?action=search
```

The following is an example of a query in the request body:

```
{
 "query":"from Entity where name:\"user1\"",
 "max_results":20,
 "offset":10
}
```

8.6.2. Monitoring {brandname} Clusters

Use the REST API to monitor {brandname} clusters.

Retrieving Cluster Information

To retrieve information about a {brandname} cluster, invoke a `GET` request.

```
GET /v2/cluster
```

The preceding request returns information such as the following:

```
{
   "clusterName":"ISPN",
   "healthStatus":"HEALTHY",
   "numberOfNodes":2,
   "nodeNames":[
      "NodeA",
      "NodeB"
   ]
}
```

- `clusterName` specifies the name of the cluster as defined in the configuration.
- `healthStatus` provides one of the following:
  - `UNHEALTHY` indicates at least one of the caches is in degraded mode.
  - `REBALANCING` indicates at least one cache is in the rebalancing state.
  - `HEALTHY` indicates all cache instances in the cluster are operating as expected.
- `numberOfNodes` displays the total number of cluster members. Returns a value of `0` for non-clustered (standalone) servers.
- `nodeNames` is an array of all cluster members. Empty for standalone servers.

Check availability

To check that a {brandname} exists and is available, invoke a `HEAD` request.

```
HEAD /v2/cluster
```

If the preceding request returns a successful response code then the {brandname} REST server is running and serving requests.

8.6.3. Counter

Use the REST API to work with counters.

Creating a Counter

To create a counter, use a `POST` request with the configuration as payload.

```
POST /v2/counters/{counterName}
```

The payload must contain a JSON configuration of the counter. Example:

```
{
    "weak-counter":{
        "initial-value":5,
        "storage":"PERSISTENT",
        "concurrency-level":1
    }
}
```

```
{
    "strong-counter":{
        "initial-value":3,
        "storage":"PERSISTENT",
        "upper-bound":{
            "value":5
        }
    }
}
```

Deleting a Counter

To delete a counter, send a `DELETE` request with the counter name.

```
DELETE /v2/counters/{counterName}
```

Retrieving Counters Configuration

The get the counter configuration, use a `GET` request with the counter name.

```
GET /v2/counters/{counterName}/config
```

The result will be a JSON representation of the counter config.

Adding Values to Counters

To add a value to a named counter, invoke a `POST` request.

```
POST /v2/counters/{counterName}
```

If the request payload is empty, the counter is incremented by one, otherwise the payload is interpreted as a signed long and added to the counter.

Request responses depend on the type of counter, as follows:

- `WEAK` counters return empty responses.

- `STRONG` counters return their values after the operation is applied.

This method processes only `plain/text` content.

Getting a Counter Value

To retrieve the value of a counter, invoke a `GET` request.

```
GET /v2/counters/{counterName}
```

Table 8. Headers

Header	Required or Optional	Parameter
`Accept`	OPTIONAL	The required format to return the content. Supported formats are *application/json* and *text/plain*. JSON is assumed if no header is provided.

Reseting Counters

To reset counters, use a `GET` request with the `?action=reset` parameter.

```
GET /v2/counters/{counterName}?action=reset
```

Incrementing Counters

To increment a Counter, use the `?action=increment` parameter.

```
GET /v2/counters/{counterName}?action=increment
```

Request responses depend on the type of counter, as follows:

- `WEAK` counters return empty responses.
- `STRONG` counters return their values after the operation is applied.

Adding a delta to Counters

To add an arbitrary amount to a Counter, use the params `?action=add` and `delta`.

```
GET /v2/counters/{counterName}?action=add&delta={delta}
```

Request responses depend on the type of counter, as follows:

- `WEAK` counters return empty responses.

- `STRONG` counters return their values after the operation is applied.

Decrementing Counters

To increment a Counter, use the `?action=decrement` parameter.

```
GET /v2/counters/{counterName}?action=decrement
```

Request responses depend on the type of counter, as follows:

- `WEAK` counters return empty responses.
- `STRONG` counters return their values after the operation is applied.

compareAndSet Strong Counters

```
GET /v2/counters/{counterName}?action=compareAndSet&expect={expect}&update={update}
```

Atomically sets the value to {update} if the current value is {expect}

Returns *true* if successful.

compareAndSwap Strong Counter

```
GET /v2/counters/{counterName}?action=compareAndSwap&expect={expect}&update={update}
```

Atomically sets the value to {update} if the current value is {expect}. Returns the previous value in the payload if the operation is successful.

8.7. Client-Side Code

Part of the point of a RESTful service is that you don't need to have tightly coupled client libraries/bindings. All you need is a HTTP client library. For Java, Apache HTTP Commons Client works just fine (and is used in the integration tests), or you can use java.net API.

8.7.1. Ruby example

```
# Shows how to interact with the REST api from ruby.
# No special libraries, just standard net/http
#
# Author: Michael Neale
#
require 'net/http'

uri = URI.parse('http://localhost:8080/rest/default/MyKey')
http = Net::HTTP.new(uri.host, uri.port)
```

```
#Create new entry

post = Net::HTTP::Post.new(uri.path, {"Content-Type" => "text/plain"})
post.basic_auth('user','pass')
post.body = "DATA HERE"

resp = http.request(post)

puts "POST response code : " + resp.code

#get it back

get = Net::HTTP::Get.new(uri.path)
get.basic_auth('user','pass')
resp = http.request(get)

puts "GET response code: " + resp.code
puts "GET Body: " + resp.body

#use PUT to overwrite

put = Net::HTTP::Put.new(uri.path, {"Content-Type" => "text/plain"})
put.basic_auth('user','pass')
put.body = "ANOTHER DATA HERE"

resp = http.request(put)

puts "PUT response code : " + resp.code

#and remove...
delete = Net::HTTP::Delete.new(uri.path)
delete.basic_auth('user','pass')

resp = http.request(delete)

puts "DELETE response code : " + resp.code

#Create binary data like this... just the same...

uri = URI.parse('http://localhost:8080/rest/default/MyLogo')
put = Net::HTTP::Put.new(uri.path, {"Content-Type" => "application/octet-stream"})
put.basic_auth('user','pass')
put.body = File.read('./logo.png')

resp = http.request(put)

puts "PUT response code : " + resp.code

#and if you want to do json...
require 'rubygems'
require 'json'
```

```
#now for fun, lets do some JSON !
uri = URI.parse('http://localhost:8080/rest/jsonCache/user')
put = Net::HTTP::Put.new(uri.path, {"Content-Type" => "application/json"})
put.basic_auth('user','pass')

data = {:name => "michael", :age => 42 }
put.body = data.to_json

resp = http.request(put)

puts "PUT response code : " + resp.code

get = Net::HTTP::Get.new(uri.path)
get.basic_auth('user','pass')
resp = http.request(get)

puts "GET Body: " + resp.body
```

8.7.2. Python 3 example

```
import urllib.request

# Setup basic auth
base_uri = 'http://localhost:8080/rest/default'
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(user='user', passwd='pass', realm='ApplicationRealm', uri
=base_uri)
opener = urllib.request.build_opener(auth_handler)
urllib.request.install_opener(opener)

# putting data in
data = "SOME DATA HERE \!"

req = urllib.request.Request(url=base_uri + '/Key', data=data.encode("UTF-8"), method
='PUT',
                             headers={"Content-Type": "text/plain"})
with urllib.request.urlopen(req) as f:
    pass

print(f.status)
print(f.reason)

# getting data out
resp = urllib.request.urlopen(base_uri + '/Key')
print(resp.read().decode('utf-8'))
```

8.7.3. Java example

```
package org.infinispan;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Base64;

/**
 * Rest example accessing a cache.
 *
 * @author Samuel Tauil (samuel@redhat.com)
 */
public class RestExample {

    /**
     * Method that puts a String value in cache.
     *
     * @param urlServerAddress URL containing the cache and the key to insert
     * @param value            Text to insert
     * @param user             Used for basic auth
     * @param password         Used for basic auth
     */
    public void putMethod(String urlServerAddress, String value, String user, String
password) throws IOException {
        System.out.println("-----------------------------------------");
        System.out.println("Executing PUT");
        System.out.println("-----------------------------------------");
        URL address = new URL(urlServerAddress);
        System.out.println("executing request " + urlServerAddress);
        HttpURLConnection connection = (HttpURLConnection) address.openConnection();
        System.out.println("Executing put method of value: " + value);
        connection.setRequestMethod("PUT");
        connection.setRequestProperty("Content-Type", "text/plain");
        addAuthorization(connection, user, password);
        connection.setDoOutput(true);

        OutputStreamWriter outputStreamWriter = new OutputStreamWriter(connection
.getOutputStream());
        outputStreamWriter.write(value);

        connection.connect();
        outputStreamWriter.flush();
        System.out.println("-----------------------------------------");
        System.out.println(connection.getResponseCode() + " " + connection
.getResponseMessage());
```

```
        System.out.println("----------------------------------------");
        connection.disconnect();
    }

    /**
     * Method that gets a value by a key in url as param value.
     *
     * @param urlServerAddress URL containing the cache and the key to read
     * @param user             Used for basic auth
     * @param password         Used for basic auth
     * @return String value
     */
    public String getMethod(String urlServerAddress, String user, String password)
throws IOException {
        String line;
        StringBuilder stringBuilder = new StringBuilder();

        System.out.println("----------------------------------------");
        System.out.println("Executing GET");
        System.out.println("----------------------------------------");

        URL address = new URL(urlServerAddress);
        System.out.println("executing request " + urlServerAddress);

        HttpURLConnection connection = (HttpURLConnection) address.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Content-Type", "text/plain");
        addAuthorization(connection, user, password);
        connection.setDoOutput(true);

        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader
(connection.getInputStream()));

        connection.connect();

        while ((line = bufferedReader.readLine()) != null) {
            stringBuilder.append(line).append('\n');
        }

        System.out.println("Executing get method of value: " + stringBuilder.toString
());

        System.out.println("----------------------------------------");
        System.out.println(connection.getResponseCode() + " " + connection
.getResponseMessage());
        System.out.println("----------------------------------------");

        connection.disconnect();

        return stringBuilder.toString();
    }
```

```
    private void addAuthorization(HttpURLConnection connection, String user, String
pass) {
        String credentials = user + ":" + pass;
        String basic = Base64.getEncoder().encodeToString(credentials.getBytes());
        connection.setRequestProperty("Authorization", "Basic " + basic);
    }

    /**
     * Main method example.
     */
    public static void main(String[] args) throws IOException {
        RestExample restExample = new RestExample();
        String user = "user";
        String pass = "pass";
        restExample.putMethod("http://localhost:8080/rest/default/1", "Infinispan REST
Test", user, pass);
        restExample.getMethod("http://localhost:8080/rest/default/1", user, pass);
    }
}
```

Chapter 9. Memcached Server

The {brandname} Server distribution contains a server module that implements the Memcached text protocol. This allows Memcached clients to talk to one or several {brandname} backed Memcached servers. These servers could either be working standalone just like Memcached does where each server acts independently and does not communicate with the rest, or they could be clustered where servers replicate or distribute their contents to other {brandname} backed Memcached servers, thus providing clients with failover capabilities. Please refer to {brandname} Server's memcached server guide for instructions on how to configure and run a Memcached server.

9.1. Client Encoding

The memcached text protocol assumes data values read and written by clients are raw bytes. The support for type negotiation will come with the memcached binary protocol implementation, as part of ISPN-8726.

Although it's not possible for a memcached client to negotiate the data type to obtain data from the server or send data in different formats, the server can optionally be configured to handle values encoded with a certain Media Type. By setting the `client-encoding` attribute in the `memcached-connector` element, the server will return content in this configured format, and clients also send data in this format.

The `client-encoding` is useful when a single cache is accessed from multiple remote endpoints (Rest, HotRod, Memcached) and it allows to tailor the responses/requests to memcached text clients. For more infomarmation on interoperability between endpoints, consult the endpoint interoperability documentation.

9.2. Command Clarifications

9.2.1. Flush All

Even in a clustered environment, flush_all command leads to the clearing of the {brandname} Memcached server where the call lands. There's no attempt to propagate this flush to other nodes in the cluster. This is done so that flush_all with delay use case can be reproduced with the {brandname} Memcached server. The aim of passing a delay to flush_all is so that different Memcached servers in a full can be flushed at different times, and hence avoid overloading the database with requests as a result of all Memcached servers being empty. For more info, check the Memcached text protocol section on flush_all .

9.3. Unsupported Features

This section explains those parts of the memcached text protocol that for one reason or the other, are not currently supported by the {brandname} based memcached implementation.

9.3.1. Individual Stats

Due to difference in nature between the original memcached implementation which is C/C\\ based and the {brandname} implementation which is Java based, there’re some general purpose stats that are not supported. For these unsupported stats, {brandname} memcached server always returns 0.

Unsupported statistics

- pid
- pointer_size
- rusage_user
- rusage_system
- bytes
- curr_connections
- total_connections
- connection_structures
- auth_cmds
- auth_errors
- limit_maxbytes
- threads
- conn_yields
- reclaimed

9.3.2. Statistic Settings

The settings statistics section of the text protocol has not been implemented due to its volatility.

9.3.3. Settings with Arguments Parameter

Since the arguments that can be send to the Memcached server are not documented, {brandname} Memcached server does not support passing any arguments to stats command. If any parameters are passed, the {brandname} Memcached server will respond with a CLIENT_ERROR .

9.3.4. Delete Hold Time Parameter

Memcached does no longer honor the optional hold time parameter to delete command and so the {brandname} based memcached server does not implement such feature either.

9.3.5. Verbosity Command

Verbosity command is not supported since {brandname} logging cannot be simplified to defining the logging level alone.

9.4. Talking To {brandname} Memcached Servers From Non-Java Clients

This section shows how to talk to {brandname} memcached server via non-java client, such as a python script.

9.4.1. Multi Clustered Server Tutorial

The example showcases the distribution capabilities of {brandname} memcached severs that are not available in the original memcached implementation.

- Start two clustered nodes: This configuration is the same one used for the GUI demo:

```
$ ./bin/standalone.sh -c clustered.xml -Djboss.node.name=nodeA
$ ./bin/standalone.sh -c clustered.xml -Djboss.node.name=nodeB
-Djboss.socket.binding.port-offset=100
```

Alternatively use

```
$ ./bin/domain.sh
```

Which automatically starts two nodes.

- Execute test_memcached_write.py script which basically executes several write operations against the {brandname} memcached server bound to port 11211. If the script is executed successfully, you should see an output similar to this:

```
Connecting to 127.0.0.1:11211
Testing set ['Simple_Key': Simple value] ... OK
Testing set ['Expiring_Key' : 999 : 3] ... OK
Testing increment 3 times ['Incr_Key' : starting at 1 ]
Initialise at 1 ... OK
Increment by one ... OK
Increment again ... OK
Increment yet again ... OK
Testing decrement 1 time ['Decr_Key' : starting at 4 ]
Initialise at 4 ... OK
Decrement by one ... OK
Testing decrement 2 times in one call ['Multi_Decr_Key' : 3 ]
Initialise at 3 ... OK
Decrement by 2 ... OK
```

- Execute test_memcached_read.py script which connects to server bound to 127.0.0.1:11311 and verifies that it can read the data that was written by the writer script to the first server. If the script is executed successfully, you should see an output similar to this:

```
Connecting to 127.0.0.1:11311
Testing get ['Simple_Key'] should return Simple value ... OK
Testing get ['Expiring_Key'] should return nothing... OK
Testing get ['Incr_Key'] should return 4 ... OK
Testing get ['Decr_Key'] should return 3 ... OK
Testing get ['Multi_Decr_Key'] should return 1 ... OK
```

Chapter 10. Scripting

Scripting is a feature of {brandname} Server which allows invoking server-side scripts from remote clients. Scripting leverages the JDK's javax.script ScriptEngines, therefore allowing the use of any JVM languages which offer one. By default, the JDK comes with Nashorn, a ScriptEngine capable of running JavaScript.

10.1. Installing scripts

Scripts are stored in a special script cache, named '___script_cache'. Adding a script is therefore as simple as put+ting it into the cache itself. If the name of the script contains a filename extension, e.g. +myscript.js, then that extension determines the engine that will be used to execute it. Alternatively the script engine can be selected using script metadata (see below). Be aware that, when security is enabled, access to the script cache via the remote protocols requires that the user belongs to the '___script_manager' role.

10.2. Script metadata

Script metadata is additional information about the script that the user can provide to the server to affect how a script is executed. It is contained in a specially-formatted comment on the first lines of the script.

Properties are specified as key=value pairs, separated by commas. You can use several different comment styles: The `//`, `;;`, `#` depending on the scripting language you use. You can split metadata over multiple lines if necessary, and you can use single (') or double (") quotes to delimit your values.

The following are examples of valid metadata comments:

```
// name=test, language=javascript
// mode=local, parameters=[a,b,c]
```

10.2.1. Metadata properties

The following metadata property keys are available

- mode: defines the mode of execution of a script. Can be one of the following values:
  - local: the script will be executed only by the node handling the request. The script itself however can invoke clustered operations
  - distributed: runs the script using the Distributed Executor Service
- language: defines the script engine that will be used to execute the script, e.g. Javascript
- extension: an alternative method of specifying the script engine that will be used to execute the script, e.g. js
- role: a specific role which is required to execute the script

- parameters: an array of valid parameter names for this script. Invocations which specify parameter names not included in this list will cause an exception.
- datatype: optional property providing information, in the form of Media Types (also known as MIME) about the type of the data stored in the caches, as well as parameter and return values. Currently it only accepts a single value which is `text/plain; charset=utf-8`, indicating that data is String UTF-8 format. This metadata parameter is designed for remote clients that only support a particular type of data, making it easy for them to retrieve, store and work with parameters.

Since the execution mode is a characteristic of the script, nothing special needs to be done on the client to invoke scripts in different modes.

10.3. Script bindings

The script engine within {brandname} exposes several internal objects as bindings in the scope of the script execution. These are:

- cache: the cache against which the script is being executed
- marshaller: the marshaller to use for marshalling/unmarshalling data to the cache
- cacheManager: the cacheManager for the cache
- scriptingManager: the instance of the script manager which is being used to run the script. This can be used to run other scripts from a script.

10.4. Script parameters

Aside from the standard bindings described above, when a script is executed it can be passed a set of named parameters which also appear as bindings. Parameters are passed as name,value pairs where name is a string and value can be any value that is understood by the marshaller in use.

The following is an example of a JavaScript script which takes two parameters, multiplicand and multiplier and multiplies them. Because the last operation is an expression evaluation, its result is returned to the invoker.

```
// mode=local,language=javascript
multiplicand * multiplier
```

To store the script in the script cache, use the following Hot Rod code:

```
RemoteCache<String, String> scriptCache = cacheManager.getCache("___script_cache");
scriptCache.put("multiplication.js",
  "// mode=local,language=javascript\n" +
  "multiplicand * multiplier\n");
```

10.5. Running Scripts using the Hot Rod Java client

The following example shows how to invoke the above script by passing two named parameters.

```
RemoteCache<String, Integer> cache = cacheManager.getCache();
// Create the parameters for script execution
Map<String, Object> params = new HashMap<>();
params.put("multiplicand", 10);
params.put("multiplier", 20);
// Run the script on the server, passing in the parameters
Object result = cache.execute("multiplication.js", params);
```

10.6. Distributed execution

The following is a script which runs on all nodes. Each node will return its address, and the results from all nodes will be collected in a List and returned to the client.

```
// mode:distributed,language=javascript
cacheManager.getAddress().toString();
```

Chapter 11. Server Tasks

Server tasks are server-side scripts defined in Java language.

11.1. Implementing Server Tasks

To develop a server task, you should define a class that extends `org.infinispan.tasks.ServerTask` interface, defined in `infinispan-tasks-api` module.

A typical server task implementation would implement these methods:

- `setTaskContext` allows server tasks implementors to access execution context information. This includes task parameters, cache reference on which the task is executed...etc. Normally, implementors would store this information locally and use it when the task is actually executed.
- `getName` should return a unique name for the task. The client will use this name to to invoke the task.
- `getExecutionMode` is used to decide whether to invoke the task in 1 node in a cluster of N nodes or invoke it in N nodes. For example, server tasks that invoke stream processing are only required to be executed in 1 node in the cluster. This is because stream processing itself makes sure processing is distributed to all nodes in cluster.
- `call` is the method that's invoked when the user invokes the server task.

Here's an example of a hello greet task that takes as parameter the name of the person to greet.

```
package example;

import org.infinispan.tasks.ServerTask;
import org.infinispan.tasks.TaskContext;

public class HelloTask implements ServerTask<String> {

   private TaskContext ctx;

   @Override
   public void setTaskContext(TaskContext ctx) {
      this.ctx = ctx;
   }

   @Override
   public String call() throws Exception {
      String name = (String) ctx.getParameters().get().get("name");
      return "Hello " + name;
   }

   @Override
   public String getName() {
      return "hello-task";
   }

}
```

Once the task has been implemented, it needs to be wrapped inside a jar. The jar is then deployed to the {brandname} Server and from them on it can be invoked. The {brandname} Server uses service loader pattern to load the task, so implementations need to adhere to these requirements. For example, server task implementations must have a zero-argument constructor.

Moreover, the jar must contain a `META-INF/services/org.infinispan.tasks.ServerTask` file containing the fully qualified name(s) of the server tasks included in the jar. For example:

```
example.HelloTask
```

With jar packaged, the next step is to push the jar to the {brandname} Server. The server is powered by WildFly Application Server, so if using Maven Wildfly's Maven plugin can be used for this:

```
<plugin>
   <groupId>org.wildfly.plugins</groupId>
   <artifactId>wildfly-maven-plugin</artifactId>
   <version>1.2.0.Final</version>
</plugin>
```

Then call the following from command line:

```
$ mvn package wildfly:deploy
```

Alternative ways of deployment jar files to Wildfly Application Server are explained here.

Executing the task can be done using the following code:

```
// Create a configuration for a locally-running server
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer().host("127.0.0.1").port(11222);

// Connect to the server
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());

// Obtain the remote cache
RemoteCache<String, String> cache = cacheManager.getCache();

// Create task parameters
Map<String, String> parameters = new HashMap<>();
parameters.put("name", "developer");

// Execute task
String greet = cache.execute("hello-task", parameters);
System.out.println(greet);
```

Chapter 12. Health monitoring

{brandname} server has special endpoints for monitoring cluster health. The API is exposed via:

- Programmatically (using `embeddedCacheManager.getHealth()`)
- JMX
- CLI
- REST (using WildFly HTTP Management API)

12.1. Accessing Health API using JMX

At first you need to connect to the {brandname} Server using JMX (use JConsole or other tool for this). Next, navigate to object name `jboss.datagrid-infinispan:type=CacheManager,name="clustered",component=CacheContainerHealth`.

12.2. Accessing Health API using CLI

You can access the Health API from the Command Line Interface (CLI), as in the following examples:

Standalone

```
$ bin/ispn-cli.sh -c "/subsystem=datagrid-infinispan/cache-
container=clustered/health=HEALTH:read-resource(include-runtime=true)"
```

Domain Mode

```
$ bin/ispn-cli.sh -c "/host=master/server=${servername}/subsystem=datagrid-
infinispan/cache-container=clustered/health=HEALTH:read-resource(include-
runtime=true)"
```

Where `${servername}` is the name of the {brandname} server instance.

The following is a sample result for the CLI invocation:

```
{
    "outcome" => "success",
    "result" => {
        "cache-health" => "HEALTHY",
        "cluster-health" => ["test"],
        "cluster-name" => "clustered",
        "free-memory" => 99958L,
        "log-tail" => [
            "<time_stamp> INFO  [org.infinispan.server.endpoint] (MSC service thread
1-5) DGENDPT10001: HotRodServer listening on 127.0.0.1:11222",
            "<time_stamp> INFO  [org.infinispan.server.endpoint] (MSC service thread
1-1) DGENDPT10001: MemcachedServer listening on 127.0.0.1:11211",
            "<time_stamp> INFO  [org.jboss.as.clustering.infinispan] (MSC service
thread 1-6) DGISPN0001: Started ___protobuf_metadata cache from clustered container",
            "<time_stamp> INFO  [org.jboss.as.clustering.infinispan] (MSC service
thread 1-6) DGISPN0001: Started ___script_cache cache from clustered container",
            "<time_stamp> INFO  [org.jboss.as.clustering.infinispan] (MSC service
thread 1-5) DGISPN0001: Started ___hotRodTopologyCache cache from clustered
container",
            "<time_stamp> INFO  [org.infinispan.rest.NettyRestServer] (MSC service
thread 1-6) ISPN012003: REST server starting, listening on 127.0.0.1:8080",
            "<time_stamp> INFO  [org.infinispan.server.endpoint] (MSC service thread
1-6) DGENDPT10002: REST mapped to /rest",
            "<time_stamp> INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0060:
Http management interface listening on http://127.0.0.1:9990/management",
            "<time_stamp> INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0051:
Admin console listening on http://127.0.0.1:9990",
            "<time_stamp> INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0025:
{brandname} Server <build_version> (WildFly Core <build_version>) started in 8681ms -
Started 196 of 237 services (121 services are lazy, passive or on-demand)"
        ],
        "number-of-cpus" => 8,
        "number-of-nodes" => 1,
        "total-memory" => 235520L
    }
}
```

12.3. Accessing Health API using REST

The REST interface lets you access the same set of resources as the CLI. However, the HTTP Management API requires authentication so you must first add credentials with the `add-user.sh` script.

After you set up credentials, access the Health API via REST as in the following examples:

Standalone

```
curl --digest -L -D - "http://localhost:9990/management/subsystem/datagrid-
infinispan/cache-container/clustered/health/HEALTH?operation=resource&include-
runtime=true&json.pretty=1" --header "Content-Type: application/json" -u
username:password
```

Domain Mode

```
curl --digest -L -D -
"http://localhost:9990/management/host/master/server/${servername}/subsystem/datagr
id-infinispan/cache-container/clustered/health/HEALTH?operation=resource&include-
runtime=true&json.pretty=1" --header "Content-Type: application/json" -u
username:password
```

Where `${servername}` is the name of the {brandname} server instance.

The following is a sample result for the REST invocation:

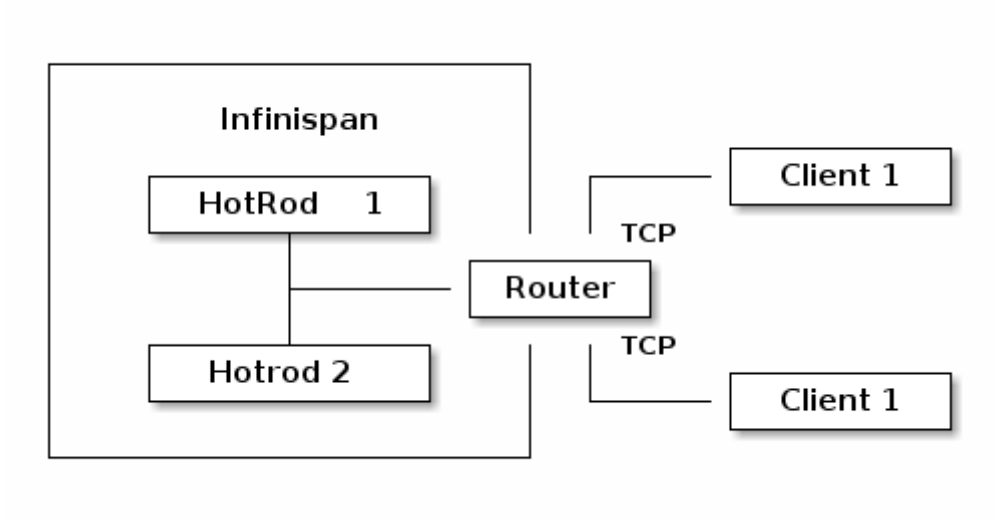
```
HTTP/1.1 200 OK
Connection: keep-alive
Authentication-Info:
nextnonce="AuZzFxz7uC4NMTQ3MDgyNTU1NTQ3OCfIJBHXVpPHPBdzGUy7Qts=",qop="auth",rspauth="b
518c3170e627bd732055c382ce5d970",cnonce="NGViOWM0NDY5OGJmNjY0MjcyOWE4NDkyZDU3YzNhYjY="
,nc=00000001
Content-Type: application/json; charset=utf-8
Content-Length: 1927
Date: <time_stamp>

{
    "cache-health" : "HEALTHY",
    "cluster-health" : ["test", "HEALTHY"],
    "cluster-name" : "clustered",
    "free-memory" : 96778,
    "log-tail" : [
        "<time_stamp> INFO  [org.infinispan.server.endpoint] (MSC service thread 1-5)
DGENDPT10001: HotRodServer listening on 127.0.0.1:11222",
        "<time_stamp> INFO  [org.infinispan.server.endpoint] (MSC service thread 1-1)
DGENDPT10001: MemcachedServer listening on 127.0.0.1:11211",
        "<time_stamp> INFO  [org.jboss.as.clustering.infinispan] (MSC service thread
1-6) DGISPN0001: Started ___protobuf_metadata cache from clustered container",
        "<time_stamp> INFO  [org.jboss.as.clustering.infinispan] (MSC service thread
1-6) DGISPN0001: Started ___script_cache cache from clustered container",
        "<time_stamp> INFO  [org.jboss.as.clustering.infinispan] (MSC service thread
1-5) DGISPN0001: Started ___hotRodTopologyCache cache from clustered container",
        "<time_stamp> INFO  [org.infinispan.rest.NettyRestServer] (MSC service thread
1-6) ISPN012003: REST server starting, listening on 127.0.0.1:8080",
        "<time_stamp> INFO  [org.infinispan.server.endpoint] (MSC service thread 1-6)
DGENDPT10002: REST mapped to /rest",
        "<time_stamp> INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0060: Http
management interface listening on http://127.0.0.1:9990/management",
        "<time_stamp> INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0051: Admin
console listening on http://127.0.0.1:9990",
        "<time_stamp> INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0025:
{brandname} Server <build_version> (WildFly Core <build_version>) started in 8681ms -
Started 196 of 237 services (121 services are lazy, passive or on-demand)"
    ],
    "number-of-cpus" : 8,
    "number-of-nodes" : 1,
    "total-memory" : 235520
}%
```

Note that the result from the REST API is exactly the same as the one obtained by CLI.

Chapter 13. Multi-tenancy

Multi-tenancy allows accessing multiple containers as shown below:



Currently there are two supported protocols for accessing the data - using Hot Rod client and using REST interface.

13.1. Using REST interface

Multi-tenancy router uses URL prefixes to separate containers using the following template: <code><a href="https://<server_ip>:<server_port>/rest/<rest_connector_name>/<cache_name>/<key>" class="bare">https://<server_ip>:<server_port>/rest/<rest_connector_name>/<cache_name>/<key>;</code>. All HTTP operations remain exactly the same as using standard <code>rest-connector</code>.

The REST connector by default support both HTTP/1.1 and HTTP/2 protocols. The switching from HTTP/1.1 to HTTP/2 procedure involves either using TLS/ALPN negotiation or HTTP/1.1 upgrade procedure. The former requires proper encryption to be enabled. The latter is always enabled.

13.2. Using Hot Rod client

Multi-tenant routing for binary protocols requires using a standard, transport layer mechanism such as SSL/TLS Server Name Indication. The server needs to be configured to support encryption and additional SNI routing needs to be added to the `router-connector`.

In order to connect to a secured Hot Rod server, the client needs to use configuration similar to this:

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(hotrodServer.getPort())
     .security()
        .ssl()
           .enabled(sslClient)
           .sniHostName("hotrod-1") // SNI Host Name
           .trustStoreFileName("truststore.jks")
           .trustStorePassword("secret".toCharArray());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

13.2.1. Multi-tenant router

The Multi-tenant router endpoint works as a facade for one or more REST/Hot Rod connectors. Its main purpose is to forward client requests into proper container.

In order to properly configure the routing, `socket-binding` attributes of other connectors must be disabled and additional attribute `name` must be used as shown below:

```
<rest-connector name="rest-1" cache-container="local"/>
<rest-connector name="rest-2" cache-container="local"/>
<hotrod-connector name="hotrod-1" cache-container="local" />
<hotrod-connector name="hotrod-2" cache-container="local" />
```

The next step is to add a new `router-connector` endpoint and configure how other containers will be accessed. Note that Hot Rod connectors require using TLS/SNI and REST connectors require using prefix in the URL:

```
<router-connector hotrod-socket-binding="hotrod" rest-socket-binding="rest" keep-
alive="true" tcp-nodelay="false" receive-buffer-size="1024" send-buffer-size="1024">
    <hotrod name="hotrod-1" >
        <sni host-name="hotrod-1" security-realm="SSLRealm1"/>
    </hotrod>
    <hotrod name="hotrod-2" >
        <sni host-name="hotrod-2" security-realm="SSLRealm2"/>
    </hotrod>
    <rest name="rest-1">
        <prefix path="rest-1" />
    </rest>
    <rest name="rest-2">
        <prefix path="rest-2" />
    </rest>
</router-connector>
```

With the following configuration, Hot Rod clients will access `hotrod-1` connector when using SNI

Host Name "hotrod-1". REST clients will need to use the following URL to access "rest-1" connector - `https://<server_ip>:<server_port>/rest/rest-1`.

Chapter 14. Single-Port

Single-Port is a special type of router connector which allows exposing multiple protocols over the same TCP port. This approach is very convenient because it reduces the number of ports required by a server, with advantages in security, configuration and management. Protocol switching is handled in three ways:

- **HTTP/1.1 Upgrade header**: initiate an HTTP/1.1 connection and send an `Upgrade: protocol` header where protocol is the name assigned to the desired endpoint.
- **TLS/ALPN**: protocol selection is performed based on the SNI specified by the client.
- **Hot Rod header detection**: if a Hot Rod endpoint is present in the router configuration, then any attempt to send a Hot Rod header will be detected and the protocol will be switched automatically.

The initial implementation supports only HTTP/1.1, HTTP/2 and Hot Rod protocols. The Memcached protocol is not supported.

14.1. Single-Port router

Internally, Single-Port is based on the same router component used to enable multi-tenancy, and therefore it shares the same configuration.

```
<!-- TLS/ALPN negotiation -->
<router-connector name="router-ssl" single-port-socket-binding="rest-ssl">
    <single-port security-realm="SSLRealm1">
        <hotrod name="hotrod" />
        <rest name="rest" />
    </single-port>
</router-connector>
<!-- HTTP 1.1/Upgrade procedure -->
<router-connector name="router" single-port-socket-binding="rest">
    <single-port>
        <hotrod name="hotrod" />
        <rest name="rest" />
    </single-port>
</router-connector>
```

With the configuration above, the Single-Port Router will operate on `rest` and `rest-ssl` socket bindings. The router named `router` should typically operate on port `8080` and will use HTTP/1.1 Upgrade (also known as *cleartext upgrade*) procedure. The other router instance (called `router-ssl`) should typically operate on port `8443` and will use TLS/ALPN.

14.1.1. Testing the Single-Port router

A tool such as `curl` can be used to access cache using both *cleartext upgrade* or TLS/ALPN. Here's an example:

```
curl -v -k --http2-prior-knowledge https://127.0.0.1:8443/rest/default/test
```

The `--http2-prior-knowledge` can be exchanged with `--http2` switch allowing to control how the switch procedure is being done (via Plain-Text Upgrade or TLS/ALPN).

14.2. Hot Rod

The single-port router has built-in automatic detection of Hot Rod messages which trigger a transparent "upgrade" to the Hot Rod protocol. This means that no changes are required on the client side to connect to a single-port endpoint. It also means that older clients will also be able to function seamlessly.

14.2.1. TLS/ALPN protocol selection

Another supported way to select the protocol is to use TLS/ALPN which uses the Application-Layer Protocol Negotiation spec. This feature requires that you have configured your endpoint to enable TLS. If you are using JDK 9 or greater, ALPN is supported out-of-the-box. However, if you are using JDK 8, you will need to use Netty's BoringSSL library, which leverages native libraries to enable ALPN.

```
<dependencyManagement>
      <dependency>
         <groupId>io.netty</groupId>
         <artifactId>netty-bom</artifactId>
         <!-- Pulled from Infinispan BOM -->
         <version>${version.netty}</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>

<dependency>
   <groupId>io.netty</groupId>
   <artifactId>netty-tcnative-boringssl-static</artifactId>
   <!-- The version is defined in Netty BOM -->
</dependency>
```

After adding the library, configure your trust store accordingly:

```
ConfigurationBuilder builder = new ConfigurationBuilder()
      .addServers("127.0.0.1:8443");

builder.security().ssl().enable()
      .trustStoreFileName("truststore.pkcs12")
      .trustStorePassword(DEFAULT_TRUSTSTORE_PASSWORD.toCharArray());

RemoteCacheManager remoteCacheManager = new RemoteCacheManager(builder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("default"");
```