Infinispan 9.0 Server Guide

The Infinispan community

Table of Contents

Chapter 1. About the Infinispan Server

Infinispan 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 Infinispan Server is closely tied to the latest releases of Infinispan 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 Infinispan Server-specific schema, although for most use-cases the configuration is interchangeable. See the Configuration section for more information.

Chapter 2. Getting Started

To get started using the server, download the Infinispan 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 Infinispan subsystem (apart from the Memcached endpoint which, because of the protocol's design, only allows access to a single cache).

Chapter 3. Operating modes

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

3.1. 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.

3.2. 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):

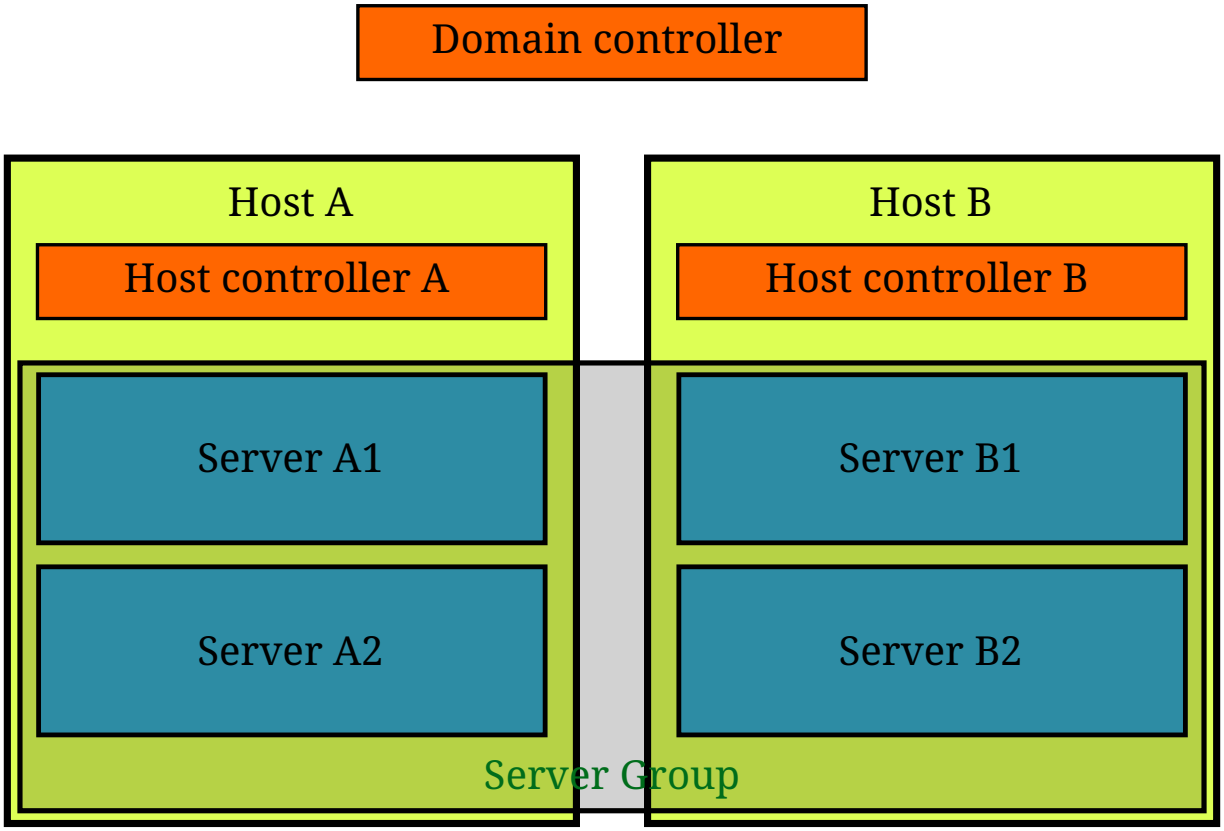

Figure 1. Domain-mode

3.2.1. 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 Infinispan server workloads. The Host Controller is responsible for starting and stopping the individual Infinispan 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 Infinispan 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 Infinispan 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.

3.2.2. 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 Infinispan 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 Infinispan 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 Infinispan 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".

3.2.3. 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 Infinispan cluster are configured as servers of one server group.

The domain can have multiple server groups, i.e. multiple Infinispan clusters. Different server groups can be configured with different profiles and deployments; for example in a domain with different Infinispan 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.

3.2.4. Server

Each "Server" in the above diagram represents an actual Infinispan 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).

3.2.5. Connecting remotely via JMX to server in Domain mode

Sometimes you want to monitor Infinispan MBeans via JMX in Domain mode. Infinispan MBeans (like cache statistics etc.) are not exposed by the host controller, you have to connect directly to the server. To do that, you have to perform following steps in domain.xml:

- uncomment <remoting-connector> in jmx subsystem:

```
<subsystem xmlns="urn:jboss:domain:jmx:1.3">
    ...
    <remoting-connector use-management-endpoint="false"/>
</subsystem>
```

- add <connector> to remoting subsystem and comment out (or remove) the default http-connector:

```
<subsystem xmlns="urn:jboss:domain:remoting:3.0">
    ...
    <!-- <http-connector name="http-remoting-connector" connector-ref="default"
security-realm="ApplicationRealm"/> -->
    <connector name="remoting-connector" socket-binding="remoting" security-
realm="ApplicationRealm"/>
</subsystem>
```

- add remoting <socket-binding> with desired port:

```
<socket-binding-groups>
    <socket-binding-group name="clustered-sockets" default-interface="public">
        ...
        <socket-binding name="remoting" port="4447"/>
    </socket-binding-group>
</socket-binding-groups>
```

Now, you should be able to connect remotely to the Infinispan server (e.g. via JConsole) using the URL service:jmx:remote://localhost:4447.

Chapter 4. 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 5. CLI

You can use the CLI to perform management operations on a standalone node or a domain controller.

```
bin/ispn-cli.sh
[disconnected /] connect
[standalone@localhost:9990 /] cd subsystem=datagrid-infinispan
[standalone@localhost:9990 subsystem=datagrid-infinispan] cd cache-container=local
[standalone@localhost:9990 cache-container=local] cd local-cache=default
[standalone@localhost:9990 local-cache=default]
```

Chapter 6. Configuration

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

6.1. JGroups subsystem configuration

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

The subsystem declaration is enclosed in the following XML element:

```
<subsystem xmlns="urn:infinispan:server:jgroups:9.0">
    <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. For each of these elements you can tune specific properties adding child <property name="prop_name">prop_value</property> elements. Since the amount of protocols and their configuration options in JGroups is huge, please refer to the appropriate JGroups Protocol documentation . The following are the default stacks:

```
<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"/>
    <protocol type="MFC"/>
    <protocol type="FRAG2"/>
</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"/>
    <protocol type="FRAG2"/>
</stack>
```

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="MFC"/>
    <protocol type="FRAG2"/>
</stack>
```

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

```
<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>
```

6.1.1. 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>
        ...
    </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 Infinispan 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>
```

6.2. Infinispan subsystem configuration

The Infinispan subsystem configures the cache containers and caches.

The subsystem declaration is enclosed in the following XML element:

```
<subsystem xmlns="urn:infinispan:server:core:9.0" default-cache-container="clustered">
  ...
</subsystem>
```

6.2.1. Containers

The Infinispan 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"/>
```

6.2.2. 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 Infinispan 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>
```

6.2.3. 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.

6.2.4. Eviction

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

```
<eviction strategy="LIRS" max-entries="1000"/>
```

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.

6.2.5. 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 Infinispan.
- *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.

6.2.6. Transactions

While it is possible to configure server caches to be transactional, none of the available protocols offer transaction capabilities.

6.2.7. Loaders and Stores

TODO

6.2.8. 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.

6.3. 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:9.0">
 ...
</subsystem>
```

6.3.1. 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>
```

6.3.2. 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"/>
```

6.3.3. WebSocket

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

6.3.4. REST

The REST connector differs from the above connectors because it piggybacks on the web subsystem. Therefore configurations such as socket binding, worker threads, timeouts, etc must be performed on the web subsystem .

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

6.3.5. 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.

6.3.6. Protocol Interoperability

By default each protocol stores data in the cache in the most efficient format for that protocol, so that no transformations are required when retrieving entries. If instead you need to access the same data from multiple protocols, you should enable compatibility mode on the caches that you want to share. This is done by adding the `<compatibility />` element to a cache definition, as follows:

```
<cache-container name="local" default-cache="default">
    <local-cache name="default" start="EAGER">
        <transaction mode="NONE"/>
        <compatibility />
    </local-cache>
</cache-container>
```

To specify a custom server-side compatibility marshaller use the "marshaller" attribute:

```
<compatibility marshaller="com.acme.CustomMarshaller"/>
```

Your custom marshaller needs to be on the classpath of the Infinispan module. You can add it by either:

- copying your jar to

  ```
  modules/system/layers/base/org/infinispan/main
  ```

  and editing the module definition to include the jar as resource-root:

  modules/system/layers/base/org/infinispan/main/modules.xml

  ```
  <resources>
      ...
      <resource-root path="acme-custom-marshallers.jar"/>
      ...
  </resources>
  ```

- or by creating a custom JBoss Module and adding it as a dependency to the Infinispan module:

modules/system/layers/base/org/infinispan/main/modules.xml

```
<dependencies>
    ...
    <module name="com.acme.custom.marshallers"/>
    ...
</dependencies>
```

Chapter 7. Security

7.1. General concepts

7.1.1. Authorization configuration

Just like embedded mode, the server supports cache authorization using the same configuration, e.g.:

```
<cache-container default-cache="secured">
   <security>
      <authorization>
     <identity-role-mapper/>
         <role name="admin" permissions="ALL" />
         <role name="reader" permissions="READ" />
         <role name="writer" permissions="WRITE" />
         <role name="supervisor" permissions="READ WRITE EXEC BULK"/>
      </authorization>
   </security>
   <local-cache name="secured">
      <security>
         <authorization roles="admin reader writer supervisor" />
      </security>
   </local-cache>
</cache-container>
```

7.1.2. Server Realms

Infinispan Server security is built around the features provided by the underlying server realm and security domains. Security Realms are used by the server to provide authentication and authorization information for both the management and application interfaces.

Security Realm configuration

```
<server xmlns="urn:jboss:domain:2.1">
   ...
   <management>
        ...
        <security-realm name="ApplicationRealm">
           <authentication>
              <properties path="application-users.properties" relative-to=
"jboss.server.config.dir"/>
           </authentication>
           <authorization>
              <properties path="application-roles.properties" relative-to=
"jboss.server.config.dir"/>
           </authorization>
        </security-realm>
        ...
    </management>
    ...
</server>
```

Infinispan Server comes with an add-user.sh script (add-user.bat for Windows) to ease the process of adding new user/role mappings to the above property files. An example invocation for adding a user to the ApplicationRealm with an initial set of roles:

./bin/add-user.sh -a -u myuser -p "qwer1234!" -ro supervisor,reader,writer

It is also possible to authenticate/authorize against alternative sources, such as LDAP, JAAS, etc. Refer to the WildFly security realms guide on how to configure the Security Realms. Bear in mind that the choice of authentication mechanism you select for the protocols limits the type of authentication sources, since the credentials must be in a format supported by the algorithm itself (e.g. pre-digested passwords for the digest algorithm)

7.2. Security Audit

The Infinispan subsystem security audit by default sends audit logs to the audit manager configured at the server level. Refer to the WildFly security subsystem guide on how to configure the server audit manager. Alternatively you can also set your custom audit logger by using the same configuration as for embedded mode. Refer to the The Security chapter in the user guide for details.

7.3. Hot Rod authentication

The Hot Rod protocol supports authentication since version 2.0 (Infinispan 7.0) by leveraging the SASL mechanisms. The supported SASL mechanisms (usually shortened as mechs) are:

- PLAIN - This is the most insecure mech, since credentials are sent over the wire in plain-text format, however it is the simplest to get to work. In combination with encryption (i.e. SSL) it can

be used safely

- DIGEST-MD5 - This mech hashes the credentials before sending them over the wire, so it is more secure than PLAIN
- GSSAPI - This mech uses Kerberos tickets, and therefore requires the presence of a properly configured Kerberos Domain Controller (such as Microsoft Active Directory)
- EXTERNAL - This mech obtains credentials from the underlying transport (i.e. from a X.509 client certificate) and therefore requires encryption using client-certificates to be enabled.

The following configuration enables authentication against ApplicationRealm, using the DIGEST-MD5 SASL mechanism:

Hot Rod connector configuration

```
<hotrod-connector socket-binding="hotrod" cache-container="default">
   <authentication security-realm="ApplicationRealm">
      <sasl server-name="myhotrodserver" mechanisms="DIGEST-MD5" qop="auth" />
   </authentication>
</hotrod-connector>
```

Notice the server-name attribute: it is the name that the server declares to incoming clients and therefore the client configuration must match.

Once you have configured a secured Hot Rod connector, you can connect to it using the Hot Rod client:

Hot Rod client configuration

```
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)
    .socketTimeout(1200000)
    .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.

7.3.1. Using GSSAPI/Kerberos

If you want to use GSSAPI/Kerberos, setup and configuration differs. First we need to define a Kerberos login module using the security domain subsystem:

Security domain configuration

```
<system-properties>
    <property name="java.security.krb5.conf" value="/tmp/infinispan/krb5.conf"/>
    <property name="java.security.krb5.debug" value="true"/>
    <property name="jboss.security.disable.secdomain.option" value="true"/>
</system-properties>

<security-domain name="infinispan-server" cache-type="default">
    <authentication>
        <login-module code="Kerberos" flag="required">
            <module-option name="debug" value="true"/>
            <module-option name="storeKey" value="true"/>
            <module-option name="refreshKrb5Config" value="true"/>
            <module-option name="useKeyTab" value="true"/>
            <module-option name="doNotPrompt" value="true"/>
            <module-option name="keyTab" value="/tmp/infinispan/infinispan.keytab"/>
            <module-option name="principal" value="HOTROD/localhost@INFINISPAN.ORG"/>
        </login-module>
    </authentication>
</security-domain>
```

Next we need to modify the Hot Rod connector

Hot Rod connector configuration

```
<hotrod-connector socket-binding="hotrod" cache-container="default">
   <authentication security-realm="ApplicationRealm">
      <sasl server-name="infinispan-server" server-context-name="infinispan-server"
mechanisms="GSSAPI" qop="auth" />
   </authentication>
</hotrod-connector>
```

On the client side you will also need to define a login module in a login configuration file:

gss.conf

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

Also you will 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.

Hot Rod client configuration

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

   public MyCallbackHandler() { }

   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);
         }
      }
   }
}

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

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .socketTimeout(1200000)
    .security()
```

```
        .authentication()
            .enable()
            .serverName("infinispan-server")
            .saslMechanism("GSSAPI")
            .clientSubject(clientSubject)
            .callbackHandler(new MyCallbackHandler());
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.

7.4. Hot Rod encryption (SSL)

The Hot Rod protocol also supports encryption using SSL/TLS with optional TLS/SNI support (Server Name Indication). To set this up you need to create a keystore using the keytool application which is part of the JDK to store your server certificate. Then add a <server-identities> element to your security realm:

Security Realm configuration for SSL

```
<security-realm name="ApplicationRealm">
    <server-identities>
        <ssl>
            <keystore path="keystore_server.jks" relative-to="jboss.server.config.dir"
keystore-password="secret" />
        </ssl>
    </server-identities>
</security-realm>
```

When using SNI support there might be multiple Security Realms configured.

Next modify the <hotrod-connector> element in the endpoint subsystem to require encryption. Optionally add SNI configuration:

Hot Rod connector SSL configuration

```
<hotrod-connector socket-binding="hotrod" cache-container="local">
    <topology-state-transfer lock-timeout="1000" replication-timeout="5000" />
    <encryption security-realm="ApplicationRealm" require-ssl-client-auth="false">
        <sni host-name="domain1" security-realm="Domain1ApplicationRealm" />
        <sni host-name="domain2" security-realm="Domain2ApplicationRealm" />
    </encryption>
</hotrod-connector>
```

In order to connect to the server, the client will need a trust store containing the public key of the

server(s) you are going to connect to:

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(hotrodServer.getPort())
        .socketTimeout(3000)
     .security()
        .ssl()
           .enabled(sslClient)
           .sniHostName("domain1")
           .trustStoreFileName("truststore.jks")
           .trustStorePassword("secret".toCharArray());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

Additionally, you might also want to enable client certificate authentication (and therefore also allow the use of the EXTERNAL SASL mech to authenticate clients).