{brandname} 9.4 Server Guide

The {brandname} community

Table of Contents

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.

Chapter 2. 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 3. Operating modes

{brandname} 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):

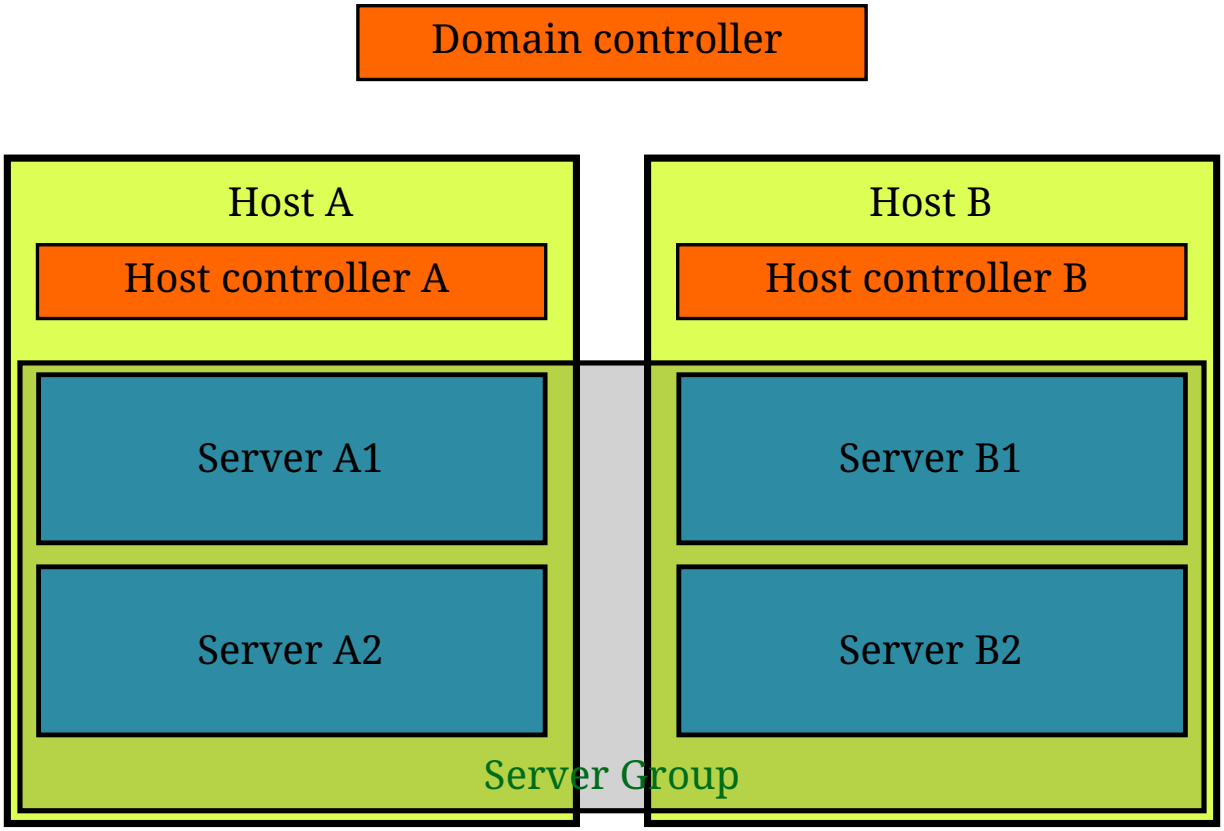


Figure 1. Domain-mode

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 {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.

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 {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".

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 {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.

3.2.4. 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).

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. Management

5.1. 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]
```

The CLI is extremely powerful and supports a number of useful features to navigate the management resource tree as well as inspecting single resources or entire subtrees. It is also possible to batch multiple commands together so that they are applied as a single operation.

5.2. Console

You can use the web console to perform management operations on servers running in either standalone or domain mode. The console only supports a subset of the operations provided by the CLI, however you can perform the following actions:

- View/Edit Cache Container Configuration
- Execute Tasks across Containers
- View/Edit Cache Configurations
- Create/Destroy Cache Instances
- View Cluster/Server/Cache Statistics
- View event logs
- Start/Stop servers/clusters (domain mode only)

To access the console start your server(s) in the required mode, navigate to `http://localhost:9990` and enter your user credentials. If you would like to contribute to the development of the console, the source code can be found here.

Before you can use the web console, you must first setup at least one user account via the `./bin/add-user.sh` script. Detailed instructions of this process are presented in your browser if you attempt to access the console before creating any user accounts.

5.3. JMX

You can monitor an {brandname} Server over JMX in two ways:

- Use JConsole or VisualVM running locally as the same user. This will use a local jvmstat connection and requires no additional setup
- Use JMX remoting (aka JSR-160) to connect from any host. This requires connecting through the management port (usually 9990) using a special protocol which respects the server security configuration

To setup a client for JMX remoting, add $ISPN_HOME/bin/client/jboss-client.jar to your client's classpath and use one of the following service URLs:

- `service:jmx:remote-http-jmx://host:port` for plain connections through the management interface
- `service:jmx:remote-https-jmx://host:port` for TLS connections through the management interface (although this requires having the appropriate keys available)
- `service:jmx:remoting-jmx://localhost:port` for connections through the remoting interface (necessary for connecting to individual servers in a domain)

The JMX subsystem registers a service with the Remoting endpoint so that remote access to JMX can be obtained over the exposed Remoting connector. This is switched on by default in standalone mode and accessible over port 9990 but in domain mode it is switched off so it needs to be enabled. In domain mode the port will be the port of the Remoting connector for the Server instance to be monitored.

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

5.4. Exposing JMX Beans to Prometheus

- Start {brandname} servers as follows:

```
$ ./standalone.sh -c cloud.xml --jmx 8180:my-config.yaml
```

5.5. Access Logs

Hot Rod and REST endpoints can record all inbound client requests as log entries with the following categories:

- `org.infinispan.HOTROD_ACCESS_LOG` logging category for the Hot Rod endpoint.

- `org.infinispan.REST_ACCESS_LOG` logging category for the REST endpoint.

5.5.1. Enabling Access Logs

Access logs for Hot Rod and REST endpoints are disabled by default. To enable either logging category, set the level to `TRACE` in the server configuration file, as in the following example:

```
<logger category="org.infinispan.HOTROD_ACCESS_LOG" use-parent-handlers="false">
    <level name="TRACE"/>
    <handlers>
       <handler name="HR-ACCESS-FILE"/>
    </handlers>
</logger>
```

5.5.2. Access Log Properties

The default format for access logs is as follows:

```
%X{address} %X{user} [%d{dd/MMM/yyyy:HH:mm:ss z}] "%X{method} %m %X{protocol}" %X{status} %X{requestSize} %X{responseSize} %X{duration}%n
```

The preceding format creates log entries such as the following:

```
127.0.0.1 - [30/Oct/2018:12:41:50 CET] "PUT /rest/default/key HTTP/1.1" 404 5 77 10
```

Logging properties use the `%X{name}` notation and let you modify the format of access logs. The following are the default logging properties:

Property	Description
`address`	Either the `X-Forwarded-For` header or the client IP address.
`user`	Principal name, if using authentication.
`method`	Method used. `PUT`, `GET`, and so on.
`protocol`	Protocol used. `HTTP/1.1`, `HTTP/2`, `HOTROD/2.9`, and so on.
`status`	An HTTP status code for the REST endpoint. `OK` or an exception for the Hot Rod endpoint.
`requestSize`	Size, in bytes, of the request.
`responseSize`	Size, in bytes, of the response.
`duration`	Number of milliseconds that the server took to handle the request.

Use the header name prefixed with `h:` to log headers that were included in requests; for example, `%X{h:User-Agent}`.

Chapter 6. Configuration

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

6.1. 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:9.4">
  <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"/>
  <protocol type="MFC"/>
  <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"/>
  <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"/>
  <protocol type="MFC"/>
  <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="org.jgroups.aws.s3.NATIVE_S3_PING" module=
"org.jgroups.aws.s3:{infinispanslot}">
  <property name="region_name">${jgroups.s3.region:}</property>
  <property name="bucket_name">${jgroups.s3.bucket_name:}</property>
  <property name="bucket_prefix">${jgroups.s3.bucket_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.

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>
</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>
```

6.2. {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>
```

6.2.1. 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"/>
```

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 {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>
```

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 `<memory>` element to your `<*-cache />`.

For more information about configuring the eviction strategy, see Eviction and Data Container in

the *User's Guide*.

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 {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.

6.2.6. 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.

6.2.7. 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`.

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:9.4">
 ...
</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. REST

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

6.3.4. Common Protocol Connector Settings

The HotRod and Memcached 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.

6.3.5. Starting and Stopping {brandname} Endpoints

Use the Command-Line Interface (CLI) to start and stop {brandname} endpoint connectors.

Commands to start and stop endpoint connectors:

- Apply to individual endpoints. To stop or start all endpoint connectors, you must run the

command on each endpoint connector.

- Take effect on single nodes only (not cluster-wide).

Procedure

1. Start the CLI and connect to {brandname}.
2. List the endpoint connectors in the `datagrid-infinispan-endpoint` subsystem, as follows:

```
[standalone@localhost:9990 /] ls subsystem=datagrid-infinispan-endpoint
hotrod-connector     memcached-connector  rest-connector       router-connector
```

3. Navigate to the endpoint connector you want to start or stop, for example:

```
[standalone@localhost:9990 /] cd subsystem=datagrid-infinispan-endpoint

[standalone@localhost:9990 subsystem=datagrid-infinispan-endpoint] cd rest-
connector=rest-connector
```

4. Use the `:stop-connector` and `:start-connector` commands as appropriate.

```
[standalone@localhost:9990 rest-connector=rest-connector] :stop-connector
{"outcome" => "success"}

[standalone@localhost:9990 rest-connector=rest-connector] :start-connector
{"outcome" => "success"}
```

6.3.6. 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.

6.3.7. 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.

Server Compatibility Mode

When using the Protostuff/Kryo bridges in compatibility mode, 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 follow the steps outlined in the Protocol Interoperability section, to place a jar containing all of their custom classes on the server's classpath.

When utilising a custom marshaller in compatibility mode, 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 compatibility mode. 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 7. Security

7.1. General concepts

7.1.1. Authorization

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

```
<cache-container default-cache="secured" name="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"/>
      </authorization>
   </security>
   <local-cache name="secured">
      <security>
         <authorization roles="admin reader writer supervisor" />
      </security>
   </local-cache>
</cache-container>
```

7.1.2. Server Realms

{brandname} 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>
```

{brandname} 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. Hot Rod authentication

The Hot Rod protocol supports authentication 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. TLS) 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 and only enables the **auth** QoP (see SASL Quality of Protection):

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. It is particularly important when using GSSAPI as it is equivalent to the Kerberos service name. You can specify multiple mechanisms and they will be attempted in order.

7.2.1. SASL Quality of Protection

While the main purpose of SASL is to provide authentication, some mechanisms also support integrity and privacy protection, also known as Quality of Protection (or qop). During authentication negotiation, ciphers are exchanged between client and server, and they can be used to add checksums and encryption to all subsequent traffic. You can tune the required level of qop as follows:

QOP	Description
auth	Authentication only
auth-int	Authentication with integrity protection
auth-conf	Authentication with integrity and privacy protection

7.2.2. SASL Policies

You can further refine the way a mechanism is chosen by tuning the SASL policies. This will effectively include / exclude mechanisms based on whether they match the desired policies.

Policy	Description
forward-secrecy	Specifies that the selected SASL mechanism must support forward secrecy between sessions. This means that breaking into one session will not automatically provide information for breaking into future sessions.
pass-credentials	Specifies that the selected SASL mechanism must require client credentials.
no-plain-text	Specifies that the selected SASL mechanism must not be susceptible to simple plain passive attacks.
no-active	Specifies that the selected SASL mechanism must not be susceptible to active (non-dictionary) attacks. The mechanism might require mutual authentication as a way to prevent active attacks.

Policy	Description
no-dictionary	Specifies that the selected SASL mechanism must not be susceptible to passive dictionary attacks.
no-anonymous	Specifies that the selected SASL mechanism must not accept anonymous logins.

Each policy's value is either "true" or "false". If a policy is absent, then the chosen mechanism need not have that characteristic (equivalent to setting the policy to "false"). One notable exception is the **no-anonymous** policy which, if absent, defaults to true, thus preventing anonymous connections.

It is possible to have mixed anonymous and authenticated connections to the endpoint, delegating actual access logic to cache authorization configuration. To do so, set the **no-anonymous** policy to false and turn on cache authorization.

The following configuration selects all available mechanisms, but effectively only enables GSSAPI, since it is the only one that respects all chosen policies:

Hot Rod connector policies

```
<hotrod-connector socket-binding="hotrod" cache-container="default">
   <authentication security-realm="ApplicationRealm">
      <sasl server-name="myhotrodserver" mechanisms="PLAIN DIGEST-MD5 GSSAPI EXTERNAL"
qop="auth">
         <policy>
            <no-active value="true" />
            <no-anonymous value="true" />
            <no-plain-text value="true" />
         </policy<>
      </sasl>
   </authentication>
</hotrod-connector>
```

7.2.3. 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>
```

7.3. Hot Rod and REST encryption (TLS/SSL)

Both Hot Rod and REST protocols support 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.

It is also possible to generate development certificates on server startup. In order to do this, just specify `generate-self-signed-certificate-host` in the keystore element as shown below:

Generating Keystore automatically

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

There are three basic principles that you should remember when using automatically generated keystores:

- They shouldn't be used on a production environment
- They are generated when necessary (e.g. while obtaining the first connection from the client)
- They contain also certificates so they might be used in a Hot Rod client directly

Next modify the <hotrod-connector> and/or <rest-connector> elements in the endpoint subsystem to require encryption. Optionally add SNI configuration:

Hot Rod connector SSL configuration

```
<hotrod-connector socket-binding="hotrod" cache-container="local">
    <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>
<rest-connector socket-binding="rest" cache-container="local">
    <encryption security-realm="ApplicationRealm" require-ssl-client-auth="false">
        <sni host-name="domain1" security-realm="Domain1ApplicationRealm" />
        <sni host-name="domain2" security-realm="Domain2ApplicationRealm" />
    </encryption>
</rest-connector>
```

To configure the client In order to connect to the server using the Hot Rod protocol, the client needs a trust store containing the public key of the server(s) you are going to connect to, unless the key was signed by a Certification Authority (CA) trusted by the JRE.

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
     .security()
        .ssl()
           .enabled(true)
           .sniHostName("domain1")
           .trustStoreFileName("truststore_client.jks")
           .trustStorePassword("secret".toCharArray());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

Additionally, you might also want to enable client certificate authentication (and optionally also allow the use of the EXTERNAL SASL mech to authenticate and authorize clients). To enable this you will need the security realm on the server to be able to trust incoming client certificates by adding a trust store:

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

And then tell the connector to require a client certificate:

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

The client, at this point, will also need to specify a keyStore which contains its certificate on top of the trustStore which trusts the server certificate. See the Hot Rod client encryption

section to learn how.

Chapter 8. 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)

8.1. 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`.

8.2. 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
    }
}
```

8.3. 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:

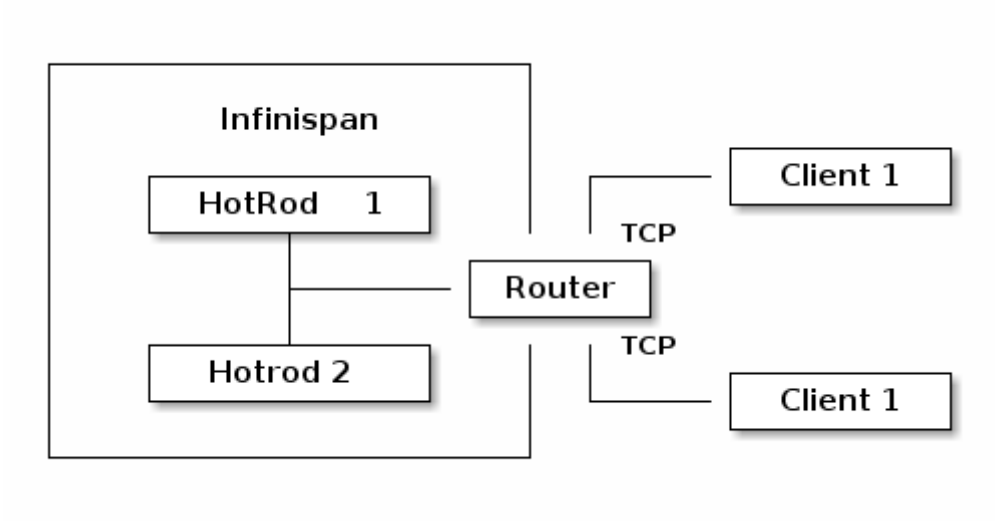
```
HTTP/1.1 200 OK
Connection: keep-alive
Authentication-Info:
nextnonce="AuZzFxz7uC4NMTQ3MDgyNTU1NTQ3OCfIJBHXVpPHPBdzGUy7Qts=",qop="auth",rspauth="b
518c3170e627bd732055c382ce5d970",cnonce="NGViOWM0NDY5OGJmNjY0MjcyOWE4NDkyZDU3YzNhYjY="
,nc=00000001
Content-Type: application/json; charset=utf-8
Content-Length: 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 9. 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.

9.1. 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.

9.2. 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());
```

9.2.1. 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`.