Deploying and Configuring Infinispan 10.0 Servers

Table of Contents

Infinispan server is a managed, distributed, and clusterable data grid that provides elastic scaling and high performance access to caches from multiple endpoints, such as Hot Rod and REST.

Chapter 1. Getting Started with Infinispan Server

Quickly set up Infinispan server and learn the basics.

1.1. Infinispan Server Requirements

Check host system requirements for the Infinispan server.

Infinispan server requires a Java Virtual Machine and supports:

- Java 8
- Java 11

1.2. Downloading Server Distributions

The Infinispan server distribution is an archive of Java libraries (`JAR` files), configuration files, and a `data` directory.

Procedure

Download the Infinispan 10.0 server from Infinispan downloads.

Verification

Use the checksum to verify the integrity of your download.

1. Run the `sha1sum` command with the server download archive as the argument, for example:

   ```
   $ sha1sum infinispan-server-${version}.zip
   ```

2. Compare with the `SHA-1` checksum value on the Infinispan downloads page.

Reference

Infinispan Server README describes the contents of the server distribution.

1.3. Installing Infinispan Server

Extract the Infinispan server archive to any directory on your host.

Procedure

Use any extraction tool with the server archive, for example:

```
$ unzip infinispan-server-${version}.zip
```

The resulting directory is your `$ISPN_HOME`.

1.4. Running Infinispan Servers

Spin up Infinispan server instances that automatically form clusters. Learn how to create cache definitions to store your data.

1.4.1. Starting Infinispan Servers

Launch Infinispan server with the startup script.

Procedure

1. Open a terminal in `$ISPN_HOME`.
2. Run the `server` script.

   Linux

   ```
   $ bin/server.sh
   ```

   Microsoft Windows

   ```
   bin\server.bat
   ```

   The server gives you these messages when it starts:

   ```
   INFO  [org.infinispan.SERVER] (main) ISPN080004: Protocol SINGLE_PORT listening
   on 127.0.0.1:11222
   INFO  [org.infinispan.SERVER] (main) ISPN080001: Infinispan Server ${version}
   started in 7453ms
   ```

Hello Infinispan!

- Open `127.0.0.1:11222` in any browser to see the Infinispan server welcome message.

Reference

Infinispan Server README describes command line arguments for the `server` script.

1.4.2. Verifying Infinispan Cluster Discovery

Infinispan servers running on the same network discover each other with the `MPING` protocol.

This procedure shows you how to use Infinispan server command arguments to start two instances on the same host and verify that the cluster view forms.

Prerequisites

Start a Infinispan server.

Procedure

1. Install and run a new Infinispan server instance.
    a. Open a terminal in `$ISPN_HOME`.
    b. Copy the root directory to `server2`.

    ```
    $ cp -r server server2
    ```

2. Specify a port offset and the location of the `server2` root directory.

    ```
    $ bin/server.sh -o 100 -s server2
    ```

Verification

Running servers return the following messages when new servers join clusters:

```
INFO  [org.infinispan.CLUSTER] (jgroups-11,<server_hostname>)
ISPN000094: Received new cluster view for channel cluster:
[<server_hostname>|3] (2) [<server_hostname>, <server2_hostname>]
INFO  [org.infinispan.CLUSTER] (jgroups-11,<server_hostname>)
ISPN100000: Node <server2_hostname> joined the cluster
```

Servers return the following messages when they join clusters:

```
INFO  [org.infinispan.remoting.transport.jgroups.JGroupsTransport] (main)
ISPN000078: Starting JGroups channel cluster
INFO  [org.infinispan.CLUSTER] (main)
ISPN000094: Received new cluster view for channel cluster:
[<server_hostname>|3] (2) [<server_hostname>, <server2_hostname>]
```

Reference

Infinispan Server README describes command line arguments for the `server` script.

1.4.3. Performing Operations with the Infinispan CLI

Connect to servers with the Infinispan command line interface (CLI) to access data and perform administrative functions.

Starting the Infinispan CLI

Start the Infinispan CLI as follows:

1. Open a terminal in `$ISPN_HOME`.
2. Run the CLI.

```
$ bin/cli.sh
[disconnected]>
```

Connecting to Infinispan Servers

Do one of the following:

- Run the `connect` command to connect to a Infinispan server on the default port of `11222`:

```
[disconnected]> connect
[hostname1@cluster//containers/default]>
```

- Specify the location of a Infinispan server. For example, connect to a local server that has a port offset of 100:

```
[disconnected]> connect 127.0.0.1:11322
[hostname2@cluster//containers/default]>
```

> Press the tab key to display available commands and options. Use the `-h` option to display help text.

Creating Caches from Templates

Use Infinispan cache templates to add caches with recommended default settings.

Procedure

1. Create a distributed, synchronous cache from a template and name it "mycache".

```
[//containers/default]> create cache --template=org.infinispan.DIST_SYNC mycache
```

> Press the tab key after the `--template=` argument to list available cache templates.

2. Retrieve the cache configuration.

```
[//containers/default]> describe caches/mycache
{
  "distributed-cache" : {
    "mode" : "SYNC",
    "remote-timeout" : 17500,
    "state-transfer" : {
      "timeout" : 60000
    },
    "transaction" : {
      "mode" : "NONE"
    },
    "locking" : {
      "concurrency-level" : 1000,
      "acquire-timeout" : 15000,
      "striping" : false
    }
  }
}
```

Adding Cache Entries

Add data to caches with the Infinispan CLI.

Prerequisites

- Create a cache named "mycache".

Procedure

1. Add a key/value pair to `mycache`.

```
[//containers/default]> put --cache=mycache hello world
```

> 💡 If the CLI is in the context of a cache, do `put k1 v1` for example:
>
> ```
> [//containers/default]> cd caches/mycache
> [//containers/default/caches/mycache]> put hello world
> ```

2. List keys in the cache.

```
[//containers/default]> ls caches/mycache
hello
```

3. Get the value for the `hello` key.

   a. Navigate to the cache.

```
[//containers/default]> cd caches/mycache
```

b. Use the `get` command to retrieve the key value.

```
[//containers/default/caches/mycache]> get hello
world
```

Shutting Down Infinispan Servers

Use the CLI to gracefully shutdown running servers. This ensures that Infinispan passivates all entries to disk and persists state.

- Use the `shutdown server` command to stop individual servers, for example:

```
[//containers/default]> shutdown server server_hostname
```

- Use the `shutdown cluster` command to stop all servers joined to the cluster, for example:

```
[//containers/default]> shutdown cluster
```

Infinispan servers log the following shutdown messages:

```
INFO  [org.infinispan.SERVER] (pool-3-thread-1) ISPN080002: Infinispan Server stopping
INFO  [org.infinispan.CONTAINER] (pool-3-thread-1) ISPN000029: Passivating all entries
to disk
INFO  [org.infinispan.CONTAINER] (pool-3-thread-1) ISPN000030: Passivated 28 entries
in 46 milliseconds
INFO  [org.infinispan.CLUSTER] (pool-3-thread-1) ISPN000080: Disconnecting JGroups
channel cluster
INFO  [org.infinispan.CONTAINER] (pool-3-thread-1) ISPN000390: Persisted state,
version=<Infinispan version> timestamp=YYYY-MM-DDTHH:MM:SS
INFO  [org.infinispan.SERVER] (pool-3-thread-1) ISPN080003: Infinispan Server stopped
INFO  [org.infinispan.SERVER] (Thread-0) ISPN080002: Infinispan Server stopping
INFO  [org.infinispan.SERVER] (Thread-0) ISPN080003: Infinispan Server stopped
```

When you shutdown Infinispan clusters, the shutdown messages include:

```
INFO  [org.infinispan.SERVER] (pool-3-thread-1) ISPN080029: Cluster shutdown
INFO  [org.infinispan.CLUSTER] (pool-3-thread-1) ISPN000080: Disconnecting JGroups
channel cluster
```

Chapter 2. Configuring Infinispan Server Endpoints

Infinispan server provides listener endpoints that handle inbound connections from client applications.

2.1. Hot Rod Server

Use the custom Hot Rod binary protocol to access Infinispan, which allows clients to do dynamic load balancing, failover, and smart routing.

- A variety of clients exist for this protocol.
- If your clients are running Java, this should be your defacto server module choice because it allows for dynamic load balancing and failover. This means that Hot Rod clients can dynamically detect changes in the topology of Hot Rod servers as long as these are clustered. When new nodes join or leave, clients update their Hot Rod server topology view. Also, when Hot Rod servers are configured with distribution, clients can detect where a particular key resides and can route requests smartly.
- Load balancing and failover are dynamically provided by Hot Rod client implementations using information provided by the server.

2.2. REST Server

Access Infinispan via a RESTful HTTP interface.

- To connect to it, you can use any HTTP client. There are many different client implementations available for many different languages and systems.
- This module is particularly recommended for those environments where the HTTP port is the only access method allowed between clients and servers.
- Clients wanting to load balance or failover between different Infinispan REST servers can do so using any standard HTTP load balancer such as mod_cluster . It’s worth noting, these load balancers maintain a static view of the servers in the backend and if a new one was to be added, it requires manual updates of the load balancer.

2.3. Memcached Server

Access Infinispan via an implementation of the Memcached text protocol.

- To connect to it, you can use any of the existing Memcached clients which are diverse.
- As opposed to Memcached servers, Infinispan based Memcached servers can actually be clustered and can replicate or distribute data using consistent hash algorithms around the cluster. This module is particularly of interest to those users that want to provide failover capabilities to the data stored in Memcached servers.
- In terms of load balancing and failover, there are a few clients that can load balance or failover

given a static list of server addresses (perl’s Cache::Memcached for example) but any server addition or removal would require manual intervention.

2.4. Server Protocol Comparison

Choosing the right protocol depends on a number of factors.

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

2.5. 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>
```

2.5.1. 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>
```

2.5.2. 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"/>
```

2.5.3. REST

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

2.5.4. 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.

Chapter 3. Monitoring Infinispan Server Logs

Infinispan uses JBoss LogManager to provide configurable logging mechanisms that capture details about the environment and record cache operations for troubleshooting purposes and root cause analysis.

3.1. Infinispan Log Files

Infinispan writes log messages to the following directory:
`$ISPN_HOME/${infinispan.server.root}/log`

`server.log`
: Messages in human readable format, including boot logs that relate to the server startup. Infinispan creates this file by default when you launch servers.

`server.log.json`
: Messages in JSON format that let you parse and analyze Infinispan logs. Infinispan creates this file when you enable the JSON-FILE log handler.

3.2. Configuring Infinispan Log Properties

You configure Infinispan logs with `logging.properties`, which is a standard Java properties file with the `property=value` format.

Procedure

1. Open `$ISPN_HOME/${infinispan.server.root}/conf/logging.properties` with any text editor.
2. Configure logging properties as appropriate.
3. Save and close `logging.properties`.

3.2.1. Log Levels

Log levels indicate the nature and severity of messages.

Log level	Description
TRACE	Provides detailed information about running state of applications. This is the most verbose log level.
DEBUG	Indicates the progress of individual requests or activities.
INFO	Indicates overall progress of applications, including lifecycle events.
WARN	Indicates circumstances that can lead to error or degrade performance.

Log level	Description
ERROR	Indicates error conditions that might prevent operations or activites from being successful but do not prevent applications from running.
FATAL	Indicates events that could cause critical service failure and application shutdown.

3.2.2. Infinispan Log Categories

Infinispan provides categories for INFO, WARN, ERROR, FATAL level messages that organize logs by functional area.

`org.infinispan.CLUSTER`
: Messages specific to Infinispan clustering that include state transfer operations, rebalancing events, partitioning, and so on.

`org.infinispan.CONFIG`
: Messages specific to Infinispan configuration.

`org.infinispan.CONTAINER`
: Messages specific to the data container that include expiration and eviction operations, cache listener notifications, transactions, and so on.

`org.infinispan.PERSISTENCE`
: Messages specific to cache loaders and stores.

`org.infinispan.SECURITY`
: Messages specific to Infinispan security.

`org.infinispan.SERVER`
: Messages specific to Infinispan servers.

`org.infinispan.XSITE`
: Messages specific to cross-site replication operations.

3.2.3. Root Logger

The root logger is `org.infinispan` and is always configured. This logger captures all messages for Infinispan log categories.

3.2.4. Log Handlers

Log handlers define how Infinispan records log messages.

CONSOLE
: Write log messages to the host standard out (`stdout`) or standard error (`stderr`) stream.
Uses the `org.jboss.logmanager.handlers.ConsoleHandler` class by default.

FILE
: Write log messages to a file.

Uses the `org.jboss.logmanager.handlers.PeriodicRotatingFileHandler` class by default.

JSON-FILE

Write log messages to a file in JSON format.
Uses the `org.jboss.logmanager.handlers.PeriodicRotatingFileHandler` class by default.

3.2.5. Log Formatters

Log formatters:

- Configure log handlers and define the appearance of log messages.
- Are strings that use syntax based on the `java.util.logging.Formatter` class.

An example is the default pattern for messages with the FILE log handler:
`%d{yyyy-MM-dd HH\:mm\:ss,SSS} %-5p [%c] (%t) %s%e%n`

- `%d` adds the current time and date.
- `%-5p` specifies the log level.
- `%c` specifies the logging category.
- `%t` adds the name of the current thread.
- `%s` specifies the simple log message.
- `%e` adds the exception stack trace.
- `%n` adds a new line.

Reference

java.util.logging.Formatter

3.2.6. Enabling and Configuring the JSON Log Handler

Infinispan provides a JSON log handler to write messages in JSON format.

Prerequisites

Ensure that Infinispan is not running. You cannot dynamically enable log handlers.

Procedure

1. Open `$ISPN_HOME/${infinispan.server.root}/conf/logging.properties` with any text editor.
2. Add `JSON-FILE` as a log handler, for example:

```
logger.handlers=CONSOLE,FILE,JSON-FILE
```

3. Optionally configure the JSON log handler and formatter.
    a. Use the `handler.JSON-FILE` property to configure the JSON log handler.
    b. Use the `formatter.JSON-FORMATTER` property to configure the JSON log formatter.
4. Save and close `logging.properties`.

When you start Infinispan, it writes each log message as a JSON map in the following file: `$ISPN_HOME/${infinispan.server.root}/log/server.log.json`

3.3. Logging Framework

Infinispan uses the JBoss Logging Framework and delegates to logging providers in the following order:

1. JBoss Logging, if you are running Infinispan servers.
2. Apache Log4j, if `org.apache.log4j.LogManager` and `org.apache.log4j.Hierarchy` are on the classpath.
3. LogBack, if `ch.qos.logback.classic.Logger` is on the classpath.
4. JDK logging (`java.util.logging`), if no other logging provider is available.

3.4. 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.

3.4.1. 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 Infinispan server configuration, 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>
```

3.4.2. Access Log Properties

The default format for access logs is as follows:

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

The preceding format creates log entries such as the following:

`127.0.0.1 - [DD/MM/YYYY:HH:MM:SS] "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 4. Monitoring Server Health

Monitor the health of your Infinispan clusters in the following ways:

- Programmatically with `embeddedCacheManager.getHealth()` method calls.
- JMX MBeans
- Infinispan REST Server

4.1. Accessing the Health API via JMX

Retrieve Infinispan cluster health statistics via JMX.

Procedure

1. Connect to Infinispan server using any JMX capable tool such as JConsole and navigate to the following object:

   ```
   org.infinispan:type=CacheManager,name="default",component=CacheContainerHealth
   ```

2. Select available MBeans to retrieve cluster health statistics.

4.2. Accessing the Health API via REST

Get Infinispan cluster health via the REST API.

Procedure

- Invoke a `GET` request to retrieve cluster health.

  ```
  GET /rest/v2/cache-managers/{cacheManagerName}/health
  ```

Infinispan responds with a `JSON` document such as the following:

```
{
    "cluster_health":{
        "cluster_name":"ISPN",
        "health_status":"HEALTHY",
        "number_of_nodes":2,
        "node_names":[
            "NodeA-36229",
            "NodeB-28703"
        ]
    },
    "cache_health":[
        {
            "status":"HEALTHY",
            "cache_name":"___protobuf_metadata"
        },
        {
            "status":"HEALTHY",
            "cache_name":"cache2"
        },
        {
            "status":"HEALTHY",
            "cache_name":"mycache"
        },
        {
            "status":"HEALTHY",
            "cache_name":"cache1"
        }
    ]

}
```

Get cache manager status as follows:

```
GET /rest/v2/cache-managers/{cacheManagerName}/health/status
```

Reference

See the *REST v2 (version 2) API* documentation for more information.

Chapter 5. Monitoring Infinispan Servers

5.1. Monitoring Infinispan Servers Over JMX

You can monitor an Infinispan 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 you need to add the $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>
```

5.1.1. JMX Monitoring

Management of Infinispan instances is all about exposing as much relevant statistical information that allows administrators to get a view of the state of each Infinispan instance. Taking in account that a single installation could be made up of several tens or hundreds Infinispan instances, providing clear and concise information in an efficient manner is imperative. The following sections dive into the range of management tooling that Infinispan provides.

> Any management tool that supports JMX already has basic support for Infinispan. However, custom plugins could be written to adapt the JMX information for easier consumption.

JMX

Over the years, JMX has become the de facto standard for management and administration of middleware and as a result, the Infinispan team has decided to standardize on this technology for the exposure of management and statistical information.

Understanding The Exposed MBeans

By connecting to the VM(s) where Infinispan is running with a standard JMX GUI such as JConsole or VisualVM you should find the following MBeans:

- For CacheManager level JMX statistics, without further configuration, you should see an MBean called *org.infinispan:type=CacheManager,name="DefaultCacheManager"* with properties specified by the CacheManager MBean .
- Using the cacheManagerName attribute in globalJmxStatistics XML element, or using the corresponding GlobalJmxStatisticsConfigurationBuilder.cacheManagerName(String cacheManagerName) call, you can name the cache manager in such way that the name is used as part of the JMX object name. So, if the name had been "Hibernate2LC", the JMX name for the cache manager would have been: *org.infinispan:type=CacheManager,name="Hibernate2LC"* . This offers a nice and clean way to manage environments where multiple cache managers are deployed, which follows JMX best practices .
- For Cache level JMX statistics, you should see several different MBeans depending on which configuration options have been enabled. For example, if you have configured a write behind cache store, you should see an MBean exposing properties belonging to the cache store component. All Cache level MBeans follow the same format though which is the following: `org.infinispan:type=Cache,name="${name-of-cache}(${cache-mode})",manager="${name-of-cache-manager}",component=${component-name}` where:
- ${name-of-cache} has been substituted by the actual cache name. If this cache represents the default cache, its name will be `___defaultCache`.
- ${cache-mode} has been substituted by the cache mode of the cache. The cache mode is represented by the lower case version of the possible enumeration values shown here.
- ${name-of-cache-manager} has been substituted by the name of the cache manager to which this cache belongs. The name is derived from the *cacheManagerName* attribute value in `globalJmxStatistics` element.
- ${component-name} has been substituted by one of the JMX component names in the JMX reference documentation .

For example, the cache store JMX component MBean for a default cache configured with synchronous distribution would have the following name: `org.infinispan:type=Cache,name="___defaultcache(dist_sync)",manager="DefaultCacheManager",component=CacheStore`

Please note that cache and cache manager names are quoted to protect against illegal characters being used in these user-defined names.

Enabling JMX Statistics

The MBeans mentioned in the previous section are always created and registered in the

MBeanServer allowing you to manage your caches but some of their attributes do not expose meaningful values unless you take the extra step of enabling collection of statistics. Gathering and reporting statistics via JMX can be enabled at 2 different levels:

CacheManager level

The CacheManager is the entity that governs all the cache instances that have been created from it. Enabling CacheManager statistics collections differs depending on the configuration style:

- If configuring the CacheManager via XML, make sure you add the following XML under the `<cache-container />` element:

  ```
  <cache-container statistics="true"/>
  ```

- If configuring the CacheManager programmatically, simply add the following code:

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...
globalConfigurationBuilder.globalJmxStatistics().enable();
```

Cache level

At this level, you will receive management information generated by individual cache instances. Enabling Cache statistics collections differs depending on the configuration style:

- If configuring the Cache via XML, make sure you add the following XML under the one of the top level cache elements, such as `<local-cache />`:

```
<local-cache statistics="true"/>
```

- If configuring the Cache programmatically, simply add the following code:

```
ConfigurationBuilder configurationBuilder = ...
configurationBuilder.jmxStatistics().enable();
```

Monitoring cluster health

It is also possible to monitor Infinispan cluster health using JMX. On CacheManager there's an additional object called `CacheContainerHealth`. It contains the following attributes:

- cacheHealth - a list of caches and corresponding statuses (HEALTHY, DEGRADED or HEALTHY_REBALANCING)
- clusterHealth - overall cluster health
- clusterName - cluster name
- freeMemoryKb - Free memory obtained from JVM runtime measured in KB
- numberOfCpus - The number of CPUs obtained from JVM runtime

- numberOfNodes - The number of nodes in the cluster
- totalMemoryKb - Total memory obtained from JVM runtime measured in KB

Multiple JMX Domains

There can be situations where several CacheManager instances are created in a single VM, or Cache names belonging to different CacheManagers under the same VM clash.

Using different JMX domains for multi cache manager environments should be last resort. Instead, it’s possible to name a cache manager in such way that it can easily be identified and used by monitoring tools. For example:

- Via XML:

```
<cache-container statistics="true" name="Hibernate2LC"/>
```

- Programmatically:

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...
globalConfigurationBuilder.globalJmxStatistics()
    .enable()
    .cacheManagerName("Hibernate2LC");
```

Using either of these options should result on the CacheManager MBean name being: `org.infinispan:type=CacheManager,name="Hibernate2LC"`

For the time being, you can still set your own jmxDomain if you need to and we also allow duplicate domains, or rather duplicate JMX names, but these should be limited to very special cases where different cache managers within the same JVM are named equally.

Registering MBeans In Non-Default MBean Servers

Let’s discuss where Infinispan registers all these MBeans. By default, Infinispan registers them in the standard JVM MBeanServer platform . However, users might want to register these MBeans in a different MBeanServer instance. For example, an application server might work with a different MBeanServer instance to the default platform one. In such cases, users should implement the MBeanServerLookup interface provided by Infinispan so that the getMBeanServer() method returns the MBeanServer under which Infinispan should register the management MBeans. Once you have your implementation ready, simply configure Infinispan with the fully qualified name of this class. For example:

- Via XML:

```
<cache-container statistics="true">
   <jmx mbean-server-lookup="com.acme.MyMBeanServerLookup" />
</cache-container>
```

- Programmatically:

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...
globalConfigurationBuilder.globalJmxStatistics()
    .enable()
    .mBeanServerLookup(new com.acme.MyMBeanServerLookup());
```

Available MBeans

For a complete list of available MBeans, refer to the JMX reference documentation

5.1.2. JMX with Prometheus

You can also expose JMX beans using Prometheus. In order to do this, just run the server with additional parameter `--jmx`, for example: `./standalone.xml -c cloud.xml --jmx`. Prometheus configuration is stored in `prometheus_config.yaml` file. It is possible to override this file by specifying it after `--jmx` parameter. For example: `./standalone.sh -c cloud.xml --jmx my-config.yaml`.

Chapter 6. Securing Infinispan Servers

6.1. Security Concepts

6.1.1. Authorization

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

```xml
<infinispan>
   <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>

</infinispan>
```

6.1.2. 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.

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)

6.2. 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.

6.2.1. 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

6.2.2. 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.

> ℹ 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>
```

6.2.3. 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>
```

6.3. 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>
```

ℹ 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.

Chapter 7. Single Port

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

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

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

7.1. Single-Port router

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

```xml
<!-- TLS/ALPN negotiation -->
<router-connector name="router-ssl" single-port-socket-binding="rest-ssl">
    <single-port security-realm="SSLRealm1">
        <hotrod name="hotrod" />
        <rest name="rest" />
    </single-port>
</router-connector>
<!-- HTTP 1.1/Upgrade procedure -->
<router-connector name="router" single-port-socket-binding="rest">
    <single-port>
        <hotrod name="hotrod" />
        <rest name="rest" />
    </single-port>
</router-connector>
```

With the configuration above, the Single-Port Router will operate on `rest` and `rest-ssl` socket bindings. The router named `router` should typically operate on port `8080` and will use HTTP/1.1 Upgrade (also known as *cleartext upgrade*) procedure. The other router instance (called `router-ssl`) should typically operate on port `8443` and will use TLS/ALPN.

7.1.1. Testing the Single-Port router

A tool such as `curl` can be used to access cache using both *cleartext upgrade* or TLS/ALPN. Here's an example:

```
$ curl -v -k --http2-prior-knowledge https://127.0.0.1:8443/rest/default/test
```

The `--http2-prior-knowledge` can be exchanged with `--http2` switch allowing to control how the switch procedure is being done (via Plain-Text Upgrade or TLS/ALPN).

7.2. Hot Rod Protocol Detection

The single-port router has built-in automatic detection of Hot Rod messages which trigger a transparent "upgrade" to the Hot Rod protocol. This means that no changes are required on the client side to connect to a single-port endpoint. It also means that older clients will also be able to function seamlessly.

7.2.1. TLS/ALPN protocol selection

Another supported way to select the protocol is to use TLS/ALPN which uses the Application-Layer Protocol Negotiation spec. This feature requires that you have configured your endpoint to enable TLS.

Enabling ALPN

If you are using JDK 9 or greater, ALPN is supported by default. However, if you are using JDK 8, you will need to use Netty’s BoringSSL library, which leverages native libraries to enable ALPN.

1. Add Netty dependencies.

```
<dependencyManagement>
      <dependency>
         <groupId>io.netty</groupId>
         <artifactId>netty-bom</artifactId>
         <!-- Pulled from Infinispan BOM -->
         <version>${version.netty}</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>

<dependency>
   <groupId>io.netty</groupId>
   <artifactId>netty-tcnative-boringssl-static</artifactId>
   <!-- The version is defined in Netty BOM -->
</dependency>
```

2. Configure your trust store accordingly:

```
ConfigurationBuilder builder = new ConfigurationBuilder()
      .addServers("127.0.0.1:8443");

builder.security().ssl().enable()
      .trustStoreFileName("truststore.pkcs12")
      .trustStorePassword(DEFAULT_TRUSTSTORE_PASSWORD.toCharArray());

RemoteCacheManager remoteCacheManager = new RemoteCacheManager(builder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("default"");
```

Chapter 8. Adding Custom Marshaller Bridges

Infinispan 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.

8.1. Protostuff

Add the protostuff marshaller dependency to your pom:

```xml
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-marshaller-protostuff</artifactId>
  <!-- Replace ${version.infinispan} with the
  version of Infinispan that you're using. -->
  <version>${version.infinispan}</version>
</dependency>
```

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:

```java
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.

8.2. Kryo

Add the kryo marshaller dependency to your pom:

```xml
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-marshaller-kryo</artifactId>
  <!-- Replace ${version.infinispan} with the
  version of Infinispan that you're using. -->
  <version>${version.infinispan}</version>
</dependency>
```

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.

8.3. Storing deserialized objects

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

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

Bundle jar downloads:

- Kryo Bundle
- Protostuff Bundle

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

8.4. Registering Custom Schemas/Serializers

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

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

Chapter 9. Implementing Server Tasks

Create custom Infinispan server jobs that you can invoke from clients.

9.1. Custom Server Task Implementations

Custom server tasks are classes that extend the `org.infinispan.tasks.ServerTask` interface, defined in `infinispan-tasks-api` module.

Server task implementations typically include the following method calls:

- `setTaskContext()` allows access to execution context information including task parameters, cache references on which tasks are executed, and so on. In most cases, implementations store this information locally and use it when tasks are actually executed.
- `getName()` returns unique names for tasks. Clients invoke tasks with the names.
- `getExecutionMode()` determines if the task is invoked on a single node in a cluster of N nodes or is invoked on N nodes. For example, server tasks that invoke stream processing need to be executed on a single node because stream processing is distributed to all nodes.
- `call()` in the `java.util.concurrent.Callable` interface is invoked when users invoke server tasks.

Reference

- `org.infinispan.tasks.ServerTask`
- `call()`

9.2. Creating and Deploying Server Tasks

Learn how to implement and deploy server tasks to Infinispan servers.

Review the following example `HelloTask` class implementation that takes the name of a person to greet as a parameter:

```
package example;

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

public class HelloTask implements ServerTask<String> {

   private TaskContext ctx;

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

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

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

}
```

To deploy the `HelloTask` class to Infinispan server, do the following:

1. Package the `HelloTask` class in a JAR file.

   Server task implementations must adhere to service loader pattern requirements. For example, implementations must have a zero-argument constructor.

2. Add a `META-INF/services/org.infinispan.tasks.ServerTask` file to the JAR.

   The file must contain the fully qualified names of server tasks, for example:

   ```
   example.HelloTask
   ```

3. Place the JAR file in the `$ISPN_HOME/server/lib` directory of your Infinispan server.

4. Execute `HelloTask` as follows:

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

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

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

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

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

Chapter 10. Scripting

10.1. Scripting

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

10.2. Installing Scripts

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

10.2.1. Script Metadata

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

Script Metadata Comments

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

The following are examples of valid metadata comments:

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

Metadata Properties

The following metadata property keys are available

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

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

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

Script Bindings

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

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

Script Parameters

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

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

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

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

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

10.3. Running Scripts with the Hot Rod Java Client

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

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

10.4. Distributed Script Execution

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

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

Chapter 11. Performing Rolling Upgrades

Upgrade Infinispan without downtime or data loss. You can perform rolling upgrades for Infinispan servers to start using a more recent version of Infinispan.

> This section explains how to upgrade Infinispan servers, see the appropriate documentation for your Hot Rod client for upgrade procedures.

11.1. Rolling Upgrades

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

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

11.2. Setting Up Target Clusters

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

   `RemoteCacheStore` settings

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

Example RemoteCacheStore *Configuration*

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

3. Configure the target cluster to handle all client requests instead of the source cluster:

   a. Configure all clients to point to the target cluster instead of the source cluster.

   b. Restart each client node.

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

11.3. Synchronizing Data from Source Clusters

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

   JMX

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

   CLI

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

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

   Use the following parameters to tune the operation:

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

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

  For example:

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

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

   JMX

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

   CLI

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

3. Decommission the source cluster.