Deploying and Configuring Infinispan 11.0 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 11.0 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 Networking

Infinispan servers let you configure interfaces and ports to make endpoints available across your network.

By default, Infinispan servers multiplex endpoints to a single TCP/IP port and automatically detect protocols of inbound client requests.

2.1. Server Interfaces

Infinispan servers can use different strategies for binding to IP addresses.

2.1.1. Address Strategy

Uses an `inet-address` strategy that maps a single `public` interface to the IPv4 loopback address (`127.0.0.1`).

```
<interfaces>
  <interface name="public">
    <inet-address value="${infinispan.bind.address:127.0.0.1}"/>
  </interface>
</interfaces>
```

You can use the CLI `-b` argument or the `infinispan.bind.address` property to select a specific address from the command-line. See Changing the Default Bind Address.

2.1.2. Loopback Strategy

Selects a loopback address.

- **IPv4** the address block `127.0.0.0/8` is reserved for loopback addresses.
- **IPv6** the address block `::1` is the only loopback address.

```
<interfaces>
    <interface name="public">
        <loopback/>
    </interface>
</interfaces>
```

2.1.3. Non-Loopback Strategy

Selects a non-loopback address.

```
<interfaces>
    <interface name="public">
        <non-loopback/>
    </interface>
</interfaces>
```

2.1.4. Network Address Strategy

Selects networks based on IP address.

```
<interfaces>
    <interface name="public">
        <inet-address value="10.1.2.3"/>
    </interface>
</interfaces>
```

2.1.5. Any Address Strategy

Selects the `INADDR_ANY` wildcard address. As a result Infinispan servers listen on all interfaces.

```
<interfaces>
    <interface name="public">
        <any-address/>
    </interface>
</interfaces>
```

2.1.6. Link Local Strategy

Selects a *link-local* IP address.

- **IPv4** the address block `169.254.0.0/16` (`169.254.0.0 – 169.254.255.255`) is reserved for link-local addressing.
- **IPv6** the address block `fe80::/10` is reserved for link-local unicast addressing.

```
<interfaces>
    <interface name="public">
        <inet-address value="10.1.2.3"/>
    </interface>
</interfaces>
```

2.1.7. Site Local Strategy

Selects a *site-local* (private) IP address.

- **IPv4** the address blocks `10.0.0.0/8`, `172.16.0.0/12`, and `192.168.0.0/16` are reserved for site-local

addressing.

- **IPv6** the address block `fc00::/7` is reserved for site-local unicast addressing.

```
<interfaces>
    <interface name="public">
        <inet-address value="10.1.2.3"/>
    </interface>
</interfaces>
```

2.1.8. Match Host Strategy

Resolves the host name and selects one of the IP addresses that is assigned to any network interface.

Infinispan servers enumerate all available operating system interfaces to locate IP addresses resolved from the host name in your configuration.

```
<interfaces>
    <interface name="public">
        <match-host value="my_host_name"/>
    </interface>
</interfaces>
```

2.1.9. Match Interface Strategy

Selects an IP address assigned to a network interface that matches a regular expression.

Infinispan servers enumerate all available operating system interfaces to locate the interface name in your configuration.

Use regular expressions with this strategy for additional flexibility.

```
<interfaces>
    <interface name="public">
        <match-interface value="eth0"/>
    </interface>
</interfaces>
```

2.1.10. Match Address Strategy

Similar to `inet-address` but selects an IP address using a regular expression.

Infinispan servers enumerate all available operating system interfaces to locate the IP address in your configuration.

Use regular expressions with this strategy for additional flexibility.

```
<interfaces>
    <interface name="public">
        <match-address value="132\..*"/>
    </interface>
</interfaces>
```

2.1.11. Fallback Strategy

Interface configurations can include multiple strategies. Infinispan servers try each strategy in the declared order.

For example, with the following configuration, Infinispan servers first attempt to match a host, then an IP address, and then fall back to the `INADDR_ANY` wildcard address:

```
<interfaces>
    <interface name="public">
        <match-host value="my_host_name"/>
        <match-address value="132\..*"/>
        <any-address/>
    </interface>
</interfaces>
```

2.1.12. Changing the Default Bind Address for Infinispan Servers

You can use the server `-b` switch or the `infinispan.bind.address` system property to bind to a different address.

For example, bind the `public` interface to `127.0.0.2` as follows:

Linux

```
$ bin/server.sh -b 127.0.0.2
```

Windows

```
bin\server.bat -b 127.0.0.2
```

2.2. Socket Bindings

Socket bindings map endpoint connectors to server interfaces and ports.

By default, Infinispan servers provide the following socket bindings:

```
<socket-bindings default-interface="public" port-offset=
"${infinispan.socket.binding.port-offset:0}">
    <socket-binding name="default" port="${infinispan.bind.port:11222}"/>
    <socket-binding name="memcached" port="11221"/>
</socket-bindings>
```

- `socket-bindings` declares the default interface and port offset.
- `default` binds to hotrod and rest connectors to the default port `11222`.
- `memcached` binds the memcached connector to port `11221`.

The memcached endpoint is disabled by default.

To override the default interface for `socket-binding` declarations, specify the `interface` attribute.

For example, you add an `interface` declaration named "private":

```
<interfaces>
  ...
  <interface name="private">
    <inet-address value="10.1.2.3"/>
  </interface>
</interfaces>
```

You can then specify `interface="private"` in a `socket-binding` declaration to bind to the private IP address, as follows:

```
<socket-bindings default-interface="public" port-offset=
"${infinispan.socket.binding.port-offset:0}">
  ...
  <socket-binding name="private_binding" interface="private" port="1234"/>
</socket-bindings>
```

2.2.1. Specifying Port Offsets

Configure port offsets with Infinispan servers when running multiple instances on the same host. The default port offset is `0`.

Use the `-o` switch with the Infinispan CLI or the `infinispan.socket.binding.port-offset` system property to set port offsets.

For example, start a server instance with an offset of `100` as follows. With the default configuration, this results in the Infinispan server listening on port `11322`.

Linux

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

Windows

```
bin\server.bat -o 100
```

2.3. Infinispan Protocol Handling

Infinispan servers use a router connector to expose multiple protocols over the same TCP port, `11222`. Using a single port for multiple protocols simplifies configuration and management and increases security by reducing the attack surface for unauthorized users.

Infinispan servers handle HTTP/1.1, HTTP/2, and Hot Rod protocol requests via port `11222` as follows:

HTTP/1.1 upgrade headers

Client requests can include the `HTTP/1.1 upgrade` header field to initiate HTTP/1.1 connections with Infinispan servers. Client applications can then send the `Upgrade: protocol` header field, where `protocol` is a Infinispan server endpoint.

Application-Layer Protocol Negotiation (ALPN)/Transport Layer Security (TLS)

Client applications specify Server Name Indication (SNI) mappings for Infinispan server endpoints to negotiate protocols in a secure manner.

Automatic Hot Rod detection

Client requests that include Hot Rod headers automatically route to Hot Rod endpoints if the single port router configuration includes Hot Rod.

2.3.1. Configuring Clients for ALPN

Configure clients to provide ALPN messages for protocol negotiation during TLS handshakes with Infinispan servers.

Prerequisites

- Enable Infinispan server endpoints with encryption.

Procedure

1. Provide your client application with the appropriate libraries to handle ALPN/TLS exchanges with Infinispan servers.

   

   Infinispan uses Wildfly OpenSSL bindings for Java.

2. Configure clients with trust stores as appropriate.

Programmatically

```
ConfigurationBuilder builder = new ConfigurationBuilder()
      .addServers("127.0.0.1:11222");

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"");
```

Hot Rod client properties

```
infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.use_ssl = true
infinispan.client.hotrod.trust_store_file_name = truststore.pkcs12
infinispan.client.hotrod.trust_store_password = trust_store_password
```

Reference

- Infinispan Endpoint Connectors
- Wildfly OpenSSL
- SslConfigurationBuilder
- Hot Rod client configuration properties

Chapter 3. Configuring Infinispan Server Endpoints

Infinispan servers provide listener endpoints that handle requests from remote client applications.

3.1. Infinispan Endpoints

Infinispan endpoints expose the `CacheManager` interface over different connector protocols so you can remotely access data and perform operations to manage and maintain Infinispan clusters.

You can define multiple endpoint connectors on different socket bindings.

3.1.1. Hot Rod

Hot Rod is a binary TCP client-server protocol designed to provide faster data access and improved performance in comparison to text-based protocols.

Infinispan provides Hot Rod client libraries in Java, C++, C#, Node.js and other programming languages.

Topology state transfer

Infinispan uses topology caches to provide clients with cluster views. Topology caches contain entries that map internal JGroups transport addresses to exposed Hot Rod endpoints.

When client send requests, Infinispan servers compare the topology ID in request headers with the topology ID from the cache. Infinispan servers send new topology views if client have older topology IDs.

Cluster topology views allow Hot Rod clients to immediately detect when nodes join and leave, which enables dynamic load balancing and failover.

In distributed cache modes, the consistent hashing algorithm also makes it possible to route Hot Rod client requests directly to primary owners.

Reference

- Infinispan Hot Rod Server
- Hot Rod client implementations

3.1.2. REST

Infinispan exposes a RESTful interface that allows HTTP clients to access data, monitor and maintain clusters, and perform administrative operations.

You can use standard HTTP load balancers to provide clients with load balancing and failover capabilities. However, HTTP load balancers maintain static cluster views and require manual updates when cluster topology changes occur.

Reference

- Infinispan REST Server
- mod_cluster HTTP load balancer

3.1.3. Memcached

Infinispan provides an implementation of the Memcached text protocol for remote client access.

The Infinispan Memcached server supports clustering with replicated and distributed cache modes.

There are some Memcached client implementations, such as the Cache::Memcached Perl client, that can offer load balancing and failover detection capabilities with static lists of Infinispan server addresses that require manual updates when cluster topology changes occur.

Reference

- Infinispan Memcached Server
- Memcached clients
- Memcached text protocol

3.1.4. Protocol Comparison

	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	**Y**	**N**	**N**
Listeners	**Y**	**N**	**N**
Query	**Y**	**Y**	**N**
Execution	**Y**	**N**	**N**
Cross-site failover	**Y**	**N**	**N**

3.2. Endpoint Connectors

You configure Infinispan server endpoints with connector declarations that specify socket bindings, authentication mechanisms, and encryption configuration.

The default endpoint connector configuration is as follows:

```
<endpoints socket-binding="default">
   <hotrod-connector name="hotrod"/>
   <rest-connector name="rest"/>
   <memcached-connector socket-binding="memcached"/>
</endpoints>
```

- `endpoints` contains endpoint connector declarations and defines global configuration for endpoints such as default socket bindings, security realms, and whether clients must present valid TLS certificates.
- `<hotrod-connector name="hotrod"/>` declares a Hot Rod connector.
- `<rest-connector name="rest"/>` declares a Hot Rod connector.
- `<memcached-connector socket-binding="memcached"/>` declares a Memcached connector that uses the memcached socket binding.

Reference

urn:infinispan:server schema provides all available endpoint configuration.

3.2.1. Hot Rod Connectors

Hot Rod connector declarations enable Hot Rod servers.

```
<hotrod-connector name="hotrod">
  <topology-state-transfer />
  <authentication>
    ...
  </authentication>
  <encryption>
    ...
  </encryption>
</hotrod-connector>
```

- `name="hotrod"` logically names the Hot Rod connector.
- `topology-state-transfer` tunes the state transfer operations that provide Hot Rod clients with cluster topology.
- `authentication` configures SASL authentication mechanisms.
- `encryption` configures TLS settings for client connections.

Reference

urn:infinispan:server schema provides all available Hot Rod connector configuration.

3.2.2. REST Connectors

REST connector declarations enable REST servers.

```
<rest-connector name="rest">
  <authentication>
    ...
  </authentication>
  <cors-rules>
    ...
  </cors-rules>
  <encryption>
    ...
  </encryption>
</rest-connector>
```

- `name="rest"` logically names the REST connector.
- `authentication` configures authentication mechanisms.
- `cors-rules` specifies CORS (Cross Origin Resource Sharing) rules for cross-domain requests.
- `encryption` configures TLS settings for client connections.

Reference

urn:infinispan:server schema provides all available REST connector configuration.

3.2.3. Memcached Connectors

Memcached connector declarations enable Memcached servers.

Infinispan servers do not enable Memcached connectors by default.

```
<memcached-connector name="memcached" socket-binding="memcached" cache="mycache" />
```

- `name="memcached"` logically names the Memcached connector.
- `socket-binding="memcached"` declares a unique socket binding for the Memcached connector.
- `cache="mycache"` names the cache that the Memcached connector exposes. The default is `memcachedCache`.

  Memcached connectors expose a single cache only. To expose multiple caches through the Memcached endpoint, you must declare additional connectors. Each Memcached connector must also have a unique socket binding.

Reference

urn:infinispan:server schema provides all available Memcached connector configuration.

Chapter 4. Monitoring Infinispan Servers

4.1. Working with Infinispan Server Logs

Infinispan uses Apache Log4j 2 to provide configurable logging mechanisms that capture details about the environment and record cache operations for troubleshooting purposes and root cause analysis.

4.1.1. 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` appender.

4.1.2. Configuring Infinispan Log Properties

You configure Infinispan logs with `log4j2.xml`, which is described in the Log4j 2 manual.

Procedure

1. Open `$ISPN_HOME/${infinispan.server.root}/conf/log4j2.xml` with any text editor.
2. Change logging configuration as appropriate.
3. Save and close `log4j2.xml`.

Log Levels

Log levels indicate the nature and severity of messages.

Log level	Description
`TRACE`	Fine-grained debug messages, capturing the flow of individual requests through the application.
`DEBUG`	Messages for general debugging, not related to an individual request.
`INFO`	Messages about the overall progress of applications, including lifecycle events.
`WARN`	Events that can lead to error or degrade performance.

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

In addition to the levels of individual messages presented above, the configuration allows two more values: `ALL` to include all messages, and `OFF` to exclude all messages.

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.

Log Appenders

Log appenders define how Infinispan records log messages.

CONSOLE
: Write log messages to the host standard out (`stdout`) or standard error (`stderr`) stream.
Uses the `org.apache.logging.log4j.core.appender.ConsoleAppender` class by default.

FILE
: Write log messages to a file.
Uses the `org.apache.logging.log4j.core.appender.RollingFileAppender` class by default.

JSON-FILE

Write log messages to a file in JSON format.
Uses the `org.apache.logging.log4j.core.appender.RollingFileAppender` class by default.

Log Patterns

The `CONSOLE` and `FILE` appenders use a `PatternLayout` to format the log messages according to a **pattern**.

An example is the default pattern in the FILE appender:
`%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p (%t) [%c{1}] %m%throwable%n`

- `%d{yyyy-MM-dd HH:mm:ss,SSS}` adds the current time and date.
- `%-5p` specifies the log level, aligned to the right.
- `%t` adds the name of the current thread.
- `%c{1}` adds the short name of the logging category.
- `%m` adds the log message.
- `%throwable` adds the exception stack trace.
- `%n` adds a new line.

Patterns are fully described in the `PatternLayout` documentation .

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/log4j2.xml` with any text editor.
2. Uncomment the `JSON-FILE` appender and comment out the `FILE` appender:

```
<!--<AppenderRef ref="FILE"/>-->
<AppenderRef ref="JSON-FILE"/>
```

3. Optionally configure the JSON appender and layout.
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`

4.1.3. 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.

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 logging configuration, as in the following example:

```
<Logger name="org.infinispan.HOTROD_ACCESS_LOG" additivity="false" level="INFO">
   <AppenderRef ref="HR-ACCESS-FILE"/>
</Logger>
```

Access Log Properties

The default format for access logs is as follows:

```
`%X{address} %X{user} [%d{dd/MMM/yyyy:HH:mm:ss Z}] &amp;quot;%X{method} %m
%X{protocol}&amp;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 +0000] "PUT /rest/v2/caches/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}`.

4.2. Configuring Statistics, Metrics, and JMX

Enable statistics that Infinispan exports to a MicroProfile Metrics endpoint or via JMX MBeans. You can also register JMX MBeans to perform management operations.

4.2.1. Enabling Infinispan Statistics

Infinispan lets you enable statistics for Cache Managers and specific cache instances.

Infinispan servers provide default `infinispan.xml` configuration files that enable statistics for Cache Managers. If you use the default Infinispan server configuration, you only need to enable statistics when you configure caches.

Procedure

- Enable statistics declaratively or programmatically.

Declaratively

```
<cache-container statistics="true"> ①
  <local-cache name="mycache" statistics="true"/> ②
</cache-container>
```

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .statistics().enable() ①
  .build();

 ...

Configuration config = new ConfigurationBuilder()
  .statistics().enable() ②
  .build();
```

① collects statistics about the Cache Manager.

② collects statistics about the named cache.

Enabling statistics for Cache Managers does not globally enable statistics for caches. This configuration only collects statistics for the Cache Manager.

4.2.2. Enabling Infinispan Metrics

Infinispan is compatible with the Eclipse MicroProfile Metrics API and can generate gauge and histogram metrics.

- Infinispan metrics are provided at the `vendor` scope. Metrics related to the JVM are provided in the `base` scope for Infinispan server.
- Gauges provide values such as the average number of nanoseconds for write operations or JVM uptime. Gauges are enabled by default. If you enable statistics, Infinispan automatically generates gauges.
- Histograms provide details about operation execution times such as read, write, and remove times. Infinispan does not enable histograms by default because they require additional computation.

Procedure

- Configure metrics declaratively or programmatically.

Declaratively

```
<cache-container statistics="true"> ①
  <metrics gauges="true" histograms="true" /> ②
</cache-container>
```

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .statistics().enable() ①
  .metrics().gauges(true).histograms(true) ②
  .build();
```

① computes and collects statistics about the Cache Manager.

② exports collected statistics as gauge and histogram metrics.

Reference

- Eclipse MicroProfile Metrics
- Enabling Infinispan Statistics

4.2.3. Collecting Infinispan Metrics

Collect Infinispan metrics with monitoring tools such as Prometheus.

Prerequisites

- Enable statistics. If you do not enable statistics, Infinispan provides `0` and `-1` values for metrics.
- Optionally enable histograms. By default Infinispan generates gauges but not histograms.

Procedure

- Get metrics in Prometheus (OpenMetrics) format:

```
$ curl -v http://localhost:11222/metrics
```

- Get metrics in MicroProfile JSON format:

```
$ curl --header "Accept: application/json" http://localhost:11222/metrics
```

Next steps

Configure monitoring applications to collect Infinispan metrics. For example, add the following to `prometheus.yml`:

```
static_configs:
    - targets: ['localhost:11222']
```

Reference

- Prometheus Configuration
- Enabling Infinispan Statistics

4.2.4. Enabling and Configuring JMX

You can enable JMX with Infinispan servers to collect statistics and perform administrative operations.

You can enable JMX without enabling statistics if you want to use management operations only. In this case, Infinispan provides `0` values for all statistics.

Procedure

- Enable JMX declaratively or programmatically.

Declaratively

```
<cache-container>
  <jmx enabled="true" /> ①
</cache-container>
```

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .jmx().enable() ①
  .build();
```

① registers Infinispan JMX MBeans.

Reference

- Enabling Infinispan Statistics

Naming Multiple Cache Managers

In cases where multiple Infinispan Cache Managers run on the same JVM, you should uniquely identify each Cache Manager to prevent conflicts.

Procedure

- Uniquely identify each cache manager in your environment.

For example, the following examples specify "Hibernate2LC" as the cache manager name, which results in a JMX MBean named `org.infinispan:type=CacheManager,name="Hibernate2LC"`.

Declaratively

```
<cache-container name="Hibernate2LC">
  <jmx enabled="true" />
  ...
</cache-container>
```

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .cacheManagerName("Hibernate2LC")
  .jmx().enable()
  .build();
```

Reference

- GlobalConfigurationBuilder
- Infinispan Configuration Schema

Registering MBeans In Custom MBean Servers

Infinispan includes an `MBeanServerLookup` interface that you can use to register MBeans in custom MBeanServer instances.

Procedure

1. Create an implementation of `MBeanServerLookup` so that the `getMBeanServer()` method returns the custom MBeanServer instance.
2. Configure Infinispan with the fully qualified name of your class, as in the following example:

Declaratively

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

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .jmx().enable().mBeanServerLookup(new com.acme.MyMBeanServerLookup())
  .build();
```

Reference

- Infinispan Configuration Schema
- MBeanServerLookup

Infinispan MBeans

Infinispan exposes JMX MBeans that represent manageable resources.

`org.infinispan:type=Cache`
: Attributes and operations available for cache instances.

`org.infinispan:type=CacheManager`
: Attributes and operations available for cache managers, including Infinispan cache and cluster health statistics.

For a complete list of available JMX MBeans along with descriptions and available operations and attributes, see the *Infinispan JMX Components* documentation.

Reference

Infinispan JMX Components

4.3. Retrieving Server Health Statistics

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

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

4.3.1. 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.

4.3.2. 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. Securing Infinispan Servers

Protect Infinispan servers against network attacks and unauthorized access.

5.1. Cache Authorization

Infinispan can restrict access to data by authorizing requests to perform cache operations.

Infinispan maps identities, or Principals of type `java.security.Principal`, to security roles in your configuration. For example, a Principal named `reader` maps to a security role named `reader`.

Infinispan lets you assign permissions to the various roles to authorize cache operations. For example, `Cache.get()` requires read permission while `Cache.put()` requires write permission.

In this case, iff a client with the `reader` role attempts to write an entry, Infinispan denies the request and throws a security exception. However, if a client with the `writer` role sends a write request, Infinispan validates authorization and issues the client with a token for subsequent operations.

5.1.1. Cache Authorization Configuration

Infinispan configuration for cache authorization is as follows:

```
<infinispan>
   <cache-container default-cache="secured" name="secured">
      <security>
         <authorization> ①
            <identity-role-mapper /> ②
            <role name="admin" permissions="ALL" /> ③
            <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" /> ④
         </security>
      </local-cache>
   </cache-container>

</infinispan>
```

① configures cache authorization for the cache container.

② specifies an implementation of `PrincipalRoleMapper` that converts Principal names to roles.

③ names roles and assigns permissions that control access to data.

④ defines the authorized roles for the cache.

Reference

- Infinispan Configuration
- org.infinispan.security.PrincipalRoleMapper

5.2. Defining Infinispan Server Security Realms

Security realms provide identity, encryption, authentication, and authorization information to Infinispan server endpoints.

5.2.1. Property Realms

Property realms use property files to define users and groups.

`users.properties` maps usernames to passwords in plain-text format. Passwords can also be pre-digested if you use the `DIGEST-MD5` SASL mechanism or `Digest` HTTP mechanism.

```
myuser=a_password
user2=another_password
```

`groups.properties` maps users to roles.

```
supervisor=myuser,user2
reader=myuser
writer=myuser
```

Property realm configuration

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
         <properties-realm groups-attribute="Roles"> ①
            <user-properties path="users.properties" ②
                             relative-to="infinispan.server.config.path" ③
                             plain-text="true"/> ④
            <group-properties path="groups.properties" ⑤
                              relative-to="infinispan.server.config.path"/>
         </properties-realm>
      </security-realm>
   </security-realms>
</security>
```

① Defines groups as roles for Infinispan server authorization.

② Specifies the `users.properties` file.

③ Specifies that the file is relative to the `$ISPN_HOME/server/conf` directory.

④ Specifies that the passwords in `users.properties` are in plain-text format.

⑤ Specifies the `groups.properties` file.

Supported authentication mechanisms

Property realms support the following authentication mechanisms:

- **SASL:** `PLAIN`, `DIGEST-*`, and `SCRAM-*`
- **HTTP (REST):** `Basic` and `Digest`

Adding Users to Property Realms

Infinispan server provides a `user-tool` script that lets you easily add new user/role mappings to properties files.

Procedure

1. Navigate to your `$ISPN_HOME` directory.
2. Run the `user-tool` script in the `bin` folder.

For example, create a new user named "myuser" with a password of "qwer1234!" that belongs to the "supervisor", "reader", and "writer" groups:

Linux

```
$ bin/user-tools.sh -a -u myuser -p "qwer1234!" -g supervisor,reader,writer
```

Microsoft Windows

```
$ bin\user-tools.bat -a -u myuser -p "qwer1234!" -g supervisor,reader,writer
```

5.2.2. LDAP Realms

LDAP realms connect to LDAP servers, such as OpenLDAP, Red Hat Directory Server, Apache Directory Server, or Microsoft Active Directory, to authenticate users and obtain membership information.

LDAP servers can have different entry layouts, depending on the type of server and deployment. For this reason, LDAP realm configuration is complex. It is beyond the scope of this document to provide examples for all possibile configurations.

LDAP realm configuration

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
        <ldap-realm name="ldap" ①
                    url="ldap://my-ldap-server:10389" ②
                    principal="uid=admin,ou=People,dc=infinispan,dc=org" ③
                    credential="strongPassword"
                    connection-timeout="3000" read-timeout="30000" ④
                    connection-pooling="true" referral-mode="ignore"
                    page-size="30"
                    direct-verification="true"> ⑤
           <identity-mapping rdn-identifier="uid" ⑥
                             search-dn="ou=People,dc=infinispan,dc=org"> ⑦
              <attribute-mapping> ⑧
                 <attribute from="cn"
                            to="Roles"
                            filter="(&amp;(objectClass=groupOfNames)(member={1}))"
                            filter-dn="ou=Roles,dc=infinispan,dc=org"/>
              </attribute-mapping>
           </identity-mapping>
         </ldap-realm>
      </security-realm>
   </security-realms>
</security>
```

① Names the LDAP realm.

② Specifies the LDAP server connection URL.

③ Specifies a principal and credentials to connect to the LDAP server.

The principal for LDAP connections must have necessary privileges to perform LDAP queries and access specific attributes.

④ Optionally tunes LDAP server connections by specifying connection timeouts and so on.

⑤ Verifies user credentials. Infinispan attempts to connect to the LDAP server using the configured credentials. Alternatively, you can use the `user-password-mapper` element that specifies a password.

⑥ Maps LDAP entries to identities. The `rdn-identifier` specifies an LDAP attribute that finds the user entry based on a provided identifier, which is typically a username; for example, the `uid` or `sAMAccountName` attribute.

⑦ Defines a starting context that limits searches to the LDAP subtree that contains the user entries.

⑧ Retrieves all the groups of which the user is a member. There are typically two ways in which membership information is stored:

- Under group entries that usually have class `groupOfNames` in the `member` attribute. In this case, you can use an attribute filter as in the preceding example configuration. This filter searches for entries that match the supplied filter, which locates groups with a `member` attribute equal to the user's DN. The filter then extracts the group entry's CN as specified by `from`, and adds it to the user's `Roles`.
- In the user entry in the `memberOf` attribute. In this case you should use an attribute reference such as the following:

  ```
  <attribute-reference reference="memberOf" from="cn" to="Roles" />
  ```

  This reference gets all `memberOf` attributes from the user's entry, extracts the CN as specified by `from`, and adds them to the user's `Roles`.

Supported authentication mechanisms

LDAP realms support the following authentication mechanisms directly:

- **SASL:** `PLAIN`, `DIGEST-*`, and `SCRAM-*`
- **HTTP (REST):** `Basic` and `Digest`

LDAP Realm Principal Rewriting

Some SASL authentication mechanisms, such as `GSSAPI`, `GS2-KRB5` and `Negotiate`, supply a username that needs to be *cleaned up* before you can use it to search LDAP servers.

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
         <ldap-realm name="ldap"
                     url="ldap://${org.infinispan.test.host.address}:10389"
                     principal="uid=admin,ou=People,dc=infinispan,dc=org"
                     credential="strongPassword">
            <name-rewriter> ①
               <regex-principal-transformer name="domain-remover"
                                            pattern="(.*)@INFINISPAN\.ORG"
                                            replacement="$1"/>
            </name-rewriter>
            <identity-mapping rdn-identifier="uid"
                              search-dn="ou=People,dc=infinispan,dc=org">
               <attribute-mapping>
                  <attribute from="cn" to="Roles"
                             filter="(&amp;(objectClass=groupOfNames)(member={1}))"
                             filter-dn="ou=Roles,dc=infinispan,dc=org" />
               </attribute-mapping>
               <user-password-mapper from="userPassword" />
            </identity-mapping>
         </ldap-realm>
      </security-realm>
   </security-realms>
</security>
```

① Defines a rewriter that extracts the username from the principal using a regular expression.

5.2.3. Trust Store Realms

Trust store realms use keystores that contain the public certificates of all clients that are allowed to connect to Infinispan server.

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <ssl>
               <keystore path="server.p12" ①
                         relative-to="infinispan.server.config.path" ②
                         keystore-password="secret" ③
                         alias="server"/> ④
            </ssl>
         </server-identities>
         <truststore-realm path="trust.p12" ⑤
                           relative-to="infinispan.server.config.path"
                           keystore-password="secret"/>
      </security-realm>
   </security-realms>
</security>
```

① Provides an SSL server identity with a keystore that contains server certificates.

② Specifies that the file is relative to the `$ISPN_HOME/server/conf` directory.

③ Specifies a keystore password.

④ Specifies a keystore alias.

⑤ Provides a keystore that contains public certificates of all clients.

Supported authentication mechanisms

Trust store realms work with client-certificate authentication mechanisms:

- **SASL:** `EXTERNAL`
- **HTTP (REST):** `CLIENT_CERT`

5.2.4. Token Realms

Token realms use external services to validate tokens and require providers that are compatible with RFC-7662 (OAuth2 Token Introspection), such as KeyCloak.

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
         <token-realm name="token"
                      auth-server-url="https://oauth-server/auth/"> ①
            <oauth2-introspection
                     introspection-url="https://oauth-
server/auth/realms/infinispan/protocol/openid-connect/token/introspect" ②
                     client-id="infinispan-server" ③
                     client-secret="1fdca4ec-c416-47e0-867a-3d471af7050f"/> ④
         </token-realm>
      </security-realm>
   </security-realms>
</security>
```

① Specifies the URL of the authentication server.

② Specifies the URL of the token introspection endpoint.

③ Names the client identifier for Infinispan server.

④ Specifies the client secret for Infinispan server.

Supported authentication mechanisms

Token realms support the following authentication mechanisms:

- **SASL:** `OAUTHBEARER`
- **HTTP (REST):** `Bearer`

5.3. Creating Infinispan Server Identities

Server identities are defined within security realms and enable Infinispan servers to prove their identity to clients.

5.3.1. Setting Up SSL Identities

SSL identities use keystores that contain either a certificate or chain of certificates.

If security realms contain SSL identities, Infinispan servers automatically enable encryption for the endpoints that use those security realms.

Procedure

1. Create a keystore for Infinispan server.

Infinispan server supports the following keystore formats: JKS, JCEKS, PKCS12, BKS, BCFKS and UBER.

In production environments, server certificates should be signed by a trusted Certificate Authority, either Root or Intermediate CA.

2. Add the keystore to the `$ISPN_HOME/server/conf` directory.
3. Add a `server-identities` definition to the Infinispan server security realm.
4. Specify the name of the keystore along with the password and alias.

SSL Identity Configuration

The following example configures an SSL identity for Infinispan server:

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
         <server-identities> ①
            <ssl> ②
               <keystore path="server.p12" ③
                         relative-to="infinispan.server.config.path" ④
                         keystore-password="secret" ⑤
                         alias="server"/> ⑥
            </ssl>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

① Defines identities for Infinispan server.

② Configures an SSL identity for Infinispan server.

③ Names a keystore that contains Infinispan server SSL certificates.

④ Specifies that the keystore is relative to the `server/conf` directory in `$ISPN_HOME`.

⑤ Specifies a keystore password.

⑥ Specifies a keystore alias.

Automatically Generating Keystores

Configure Infinispan servers to automatically generate keystores at startup.

Automatically generated keystores:

- Should not be used in production environments.
- Are generated whenever necessary; for example, while obtaining the first connection from a client.
- Contain certificates that you can use directly in Hot Rod clients.

Procedure

1. Include the `generate-self-signed-certificate-host` attribute for the `keystore` element in the server configuration.
2. Specify a hostname for the server certificate as the value.

SSL server identity with a generated keystore

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <ssl>
               <keystore path="server.p12"
                         relative-to="infinispan.server.config.path"
                         keystore-password="secret"
                         alias="server"
                         generate-self-signed-certificate-host="localhost"/> ①
            </ssl>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

① generates a keystore using `localhost`

Tuning SSL Protocols and Cipher Suites

You can configure the SSL engine, via the Infinispan server SSL identity, to use specific protocols and ciphers.

You must ensure that you set the correct ciphers for the protocol features you want to use; for example HTTP/2 ALPN.

Procedure

1. Add the `engine` element to your Infinispan server SSL identity.
2. Configure the SSL engine with the `enabled-protocols` and `enabled-ciphersuites` attributes.

SSL engine configuration

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:11.0
          https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <ssl>
               <keystore path="server.p12"
                         relative-to="infinispan.server.config.path"
                         keystore-password="secret" alias="server"/>
               <engine enabled-protocols="TLSv1.2 TLSv1.1" ①
                       enabled-ciphersuites="SSL_RSA_WITH_AES_128_GCM_SHA256 ②
                       SSL_RSA_WITH_AES_128_CBC_SHA256"/>
            </ssl>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

① Configures the SSL engine to use TLS v1 and v2 protocols.

② Configures the SSL engine to use the specified cipher suites.

5.3.2. Setting Up Kerberos Identities

Kerberos identities use *keytab* files that contain service principal names and encrypted keys, derived from Kerberos passwords.

keytab files can contain both user and service account principals. However, Infinispan servers use service account principals only. As a result, Infinispan servers can provide identity to clients and allow clients to authenticate with Kerberos servers.

In most cases, you create unique principals for the Hot Rod and REST connectors. For example, you have a "datagrid" server in the "INFINISPAN.ORG" domain. In this case you should create the following service principals:

- `hotrod/datagrid@INFINISPAN.ORG` identifies the Hot Rod service.
- `HTTP/datagrid@INFINISPAN.ORG` identifies the REST service.

Procedure

1. Create keytab files for the Hot Rod and REST services.

   Linux

```
$ ktutil
ktutil:  addent -password -p datagrid@INFINISPAN.ORG -k 1 -e aes256-cts
Password for datagrid@INFINISPAN.ORG: [enter your password]
ktutil:  wkt http.keytab
ktutil:  quit
```

Microsoft Windows

```
$ ktpass -princ HTTP/datagrid@INFINISPAN.ORG -pass * -mapuser
INFINISPAN\USER_NAME
$ ktab -k http.keytab -a HTTP/datagrid@INFINISPAN.ORG
```

2. Copy the keytab files to the `$ISPN_HOME/server/conf` directory.
3. Add a `server-identities` definition to the Infinispan server security realm.
4. Specify the location of keytab files that provide service principals to Hot Rod and REST connectors.
5. Name the Kerberos service principals.

Kerberos Identity Configuration

The following example configures Kerberos identities for Infinispan server:

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
         <server-identities> ①
            <kerberos keytab-path="hotrod.keytab" ②
                      principal="hotrod/datagrid@INFINISPAN.ORG" ③
                      required="true"/> ④
            <kerberos keytab-path="http.keytab" ⑤
                      principal="HTTP/localhost@INFINISPAN.ORG" ⑥
                      required="true"/>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

① Defines identities for Infinispan server.

② Specifies a keytab file that provides a Kerberos identity for the Hot Rod connector.

③ Names the Kerberos service principal for the Hot Rod connector.

④ Specifies that the keytab file must exist when Infinispan server starts.

⑤ Specifies a keytab file that provides a Kerberos identity for the REST connector.

⑥ Names the Kerberos service principal for the REST connector.

5.4. Configuring Endpoint Authentication Mechanisms

Configure Hot Rod and REST connectors with SASL authentication mechanisms to authenticate with clients.

5.4.1. Setting Up Hot Rod Authentication

Configure Hot Rod connectors to authenticate with clients to Infinispan server security realms using specific SASL authentication mechanisms .

Procedure

1. Add an `authentication` definition to the Hot Rod connector configuration.
2. Specify which Infinispan security realm the Hot Rod connector uses for authentication.
3. Specify the SASL authentication mechanisms for the Hot Rod endpoint to use.
4. Configure SASL authentication properties as appropriate.

Hot Rod Authentication Configuration

Hot Rod connector with SCRAM, DIGEST, and PLAIN authentication

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:11.0
           https://infinispan.org/schemas/infinispan-server-11.0.xsd"
           xmlns="urn:infinispan:server:11.0"
           socket-binding="default" security-realm="default"> ①
   <hotrod-connector name="hotrod">
      <authentication>
         <sasl mechanisms="SCRAM-SHA-512 SCRAM-SHA-384 SCRAM-SHA-256 ②
                           SCRAM-SHA-1 DIGEST-SHA-512 DIGEST-SHA-384
                           DIGEST-SHA-256 DIGEST-SHA DIGEST-MD5 PLAIN"
               server-name="infinispan" ③
               qop="auth"/> ④
      </authentication>
   </hotrod-connector>
</endpoints>
```

① enables authentication against the security realm named "default".

② specifies SASL mechanisms to use for authentication.

③ defines the name that Infinispan servers declare to clients. The server name should match the client configuration.

④ enables **auth** QoP.

Hot Rod connector with Kerberos authentication

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
           xmlns="urn:infinispan:server:11.0"
           socket-binding="default" security-realm="default">
   <hotrod-connector name="hotrod">
      <authentication>
         <sasl mechanisms="GSSAPI GS2-KRB5" ①
               server-name="datagrid" ②
               server-principal="hotrod/datagrid@INFINISPAN.ORG"/> ③
      </authentication>
   </hotrod-connector>
</endpoints>
```

① Enables the `GSSAPI` and `GS2-KRB5` mechanisms for Kerberos authentication.

② Defines the Infinispan server name, which is equivalent to the Kerberos service name.

③ Specifies the Kerberos identity for the server.

Hot Rod Endpoint Authentication Mechanisms

Infinispan supports the following SASL authentications mechanisms with the Hot Rod connector:

Authentication mechanism	Description	Related details
`PLAIN`	Uses credentials in plain-text format. You should use `PLAIN` authentication with encrypted connections only.	Similar to the `Basic` HTTP mechanism.
`DIGEST-*`	Uses hashing algorithms and nonce values. Hot Rod connectors support `DIGEST-MD5`, `DIGEST-SHA`, `DIGEST-SHA-256`, `DIGEST-SHA-384`, and `DIGEST-SHA-512` hashing algorithms, in order of strength.	Similar to the `Digest` HTTP mechanism.
`SCRAM-*`	Uses *salt* values in addition to hashing algorithms and nonce values. Hot Rod connectors support `SCRAM-SHA`, `SCRAM-SHA-256`, `SCRAM-SHA-384`, and `SCRAM-SHA-512` hashing algorithms, in order of strength.	Similar to the `Digest` HTTP mechanism.

Authentication mechanism	Description	Related details
GSSAPI	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding kerberos server identity in the realm configuration. In most cases, you also specify an ldap-realm to provide user membership information.	Similar to the SPNEGO HTTP mechanism.
GS2-KRB5	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding kerberos server identity in the realm configuration. In most cases, you also specify an ldap-realm to provide user membership information.	Similar to the SPNEGO HTTP mechanism.
EXTERNAL	Uses client certificates.	Similar to the CLIENT_CERT HTTP mechanism.
OAUTHBEARER	Uses OAuth tokens and requires a token-realm configuration.	Similar to the BEARER_TOKEN HTTP mechanism.

SASL Quality of Protection (QoP)

If SASL mechanisms support integrity and privacy protection settings, you can add them to your Hot Rod connector configuration with the qop attribute.

QoP setting	Description
auth	Authentication only.
auth-int	Authentication with integrity protection.
auth-conf	Authentication with integrity and privacy protection.

SASL Policies

SASL policies let you control which authentication mechanisms Hot Rod connectors can use.

Policy	Description	Default value
forward-secrecy	Use only SASL mechanisms that support forward secrecy between sessions. This means that breaking into one session does not automatically provide information for breaking into future sessions.	false

Policy	Description	Default value
pass-credentials	Use only SASL mechanisms that require client credentials.	false
no-plain-text	Do not use SASL mechanisms that are susceptible to simple plain passive attacks.	false
no-active	Do not use SASL mechanisms that are susceptible to active, non-dictionary, attacks.	false
no-dictionary	Do not use SASL mechanisms that are susceptible to passive dictionary attacks.	false
no-anonymous	Do not use SASL mechanisms that accept anonymous logins.	true

Infinispan cache authorization restricts access to caches based on roles and permissions. If you configure cache authorization, you can then set `<no-anonymous value=false />` to allow anonymous login and delegate access logic to cache authorization.

Hot Rod connector with SASL policy configuration

```
<hotrod-connector socket-binding="hotrod" cache-container="default">
   <authentication security-realm="ApplicationRealm">
      <sasl server-name="myhotrodserver"
            mechanisms="PLAIN DIGEST-MD5 GSSAPI EXTERNAL" ①
            qop="auth">
         <policy> ②
            <no-active value="true" />
            <no-anonymous value="true" />
            <no-plain-text value="true" />
         </policy>
      </sasl>
   </authentication>
</hotrod-connector>
```

① Specifies multiple SASL authentication mechanisms for the Hot Rod connector.

② Defines policies for SASL mechanisms.

As a result of the preceding configuration, the Hot Rod connector uses the `GSSAPI` mechanism because it is the only mechanism that complies with all policies.

5.4.2. Setting Up REST Authentication

Configure REST connectors to authenticate with clients to Infinispan server security realms using specific SASL authentication mechanisms .

Procedure

1. Add an `authentication` definition to the REST connector configuration.

2. Specify which Infinispan security realm the REST connector uses for authentication.
3. Specify the SASL authentication mechanisms for the REST endpoint to use.
4. Configure SASL authentication properties as appropriate.

REST Authentication Configuration

REST connector with BASIC and DIGEST authentication

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
           xmlns="urn:infinispan:server:11.0"
           socket-binding="default" security-realm="default"> ①
   <rest-connector name="rest">
      <authentication mechanisms="DIGEST BASIC"/> ②
   </rest-connector>
</endpoints>
```

① Enables authentication against the security realm named "default".

② Specifies SASL mechanisms to use for authentication

REST connector with Kerberos authentication

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
           xmlns="urn:infinispan:server:11.0"
           socket-binding="default" security-realm="default">
   <rest-connector name="rest">
      <authentication mechanisms="SPNEGO" ①
                      server-principal="HTTP/localhost@INFINISPAN.ORG"/> ②
   </rest-connector>
</endpoints>
```

① Enables the `SPENGO` mechanism for Kerberos authentication.

② Specifies the Kerberos identity for the server.

REST Endpoint Authentication Mechanisms

Infinispan supports the following authentications mechanisms with the REST connector:

Authentication mechanism	Description	Related details
`BASIC`	Uses credentials in plain-text format. You should use `BASIC` authentication with encrypted connections only.	Corresponds to the `Basic` HTTP authentication scheme and is similar to the `PLAIN` SASL mechanism.

Authentication mechanism	Description	Related details
`DIGEST`	Uses hashing algorithms and nonce values. REST connectors support `SHA-512`, `SHA-256` and `MD5` hashing algorithms.	Corresponds to the `Digest` HTTP authentication scheme and is similar to `DIGEST-*` SASL mechanisms.
`SPNEGO`	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding `kerberos` server identity in the realm configuration. In most cases, you also specify an `ldap-realm` to provide user membership information.	Corresponds to the `Negotiate` HTTP authentication scheme and is similar to the `GSSAPI` and `GS2-KRB5` SASL mechanisms.
`BEARER_TOKEN`	Uses OAuth tokens and requires a `token-realm` configuration.	Corresponds to the `Bearer` HTTP authentication scheme and is similar to `OAUTHBEARER` SASL mechanism.
`CLIENT_CERT`	Uses client certificates.	Similar to the `EXTERNAL` SASL mechanism.

Chapter 6. Remotely Executing Server-Side Tasks

Define and add tasks to Infinispan servers that you can invoke from the Infinispan command line interface, REST API, or from Hot Rod clients.

You can implement tasks as custom Java classes or define scripts in languages such as JavaScript.

6.1. Creating Server Tasks

Create custom task implementations and add them to Infinispan servers.

6.1.1. Server Tasks

Infinispan server tasks are classes that extend the `org.infinispan.tasks.ServerTask` interface and generally 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 these names.

`getExecutionMode()`
: Returns the execution mode for tasks.

  - `TaskExecutionMode.ONE_NODE` only the node that handles the request executes the script. Although scripts can still invoke clustered operations.
  - `TaskExecutionMode.ALL_NODES` Infinispan uses clustered executors to run scripts across 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()`
: Computes a result. This method is defined in the `java.util.concurrent.Callable` interface and is invoked with server tasks.

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

The following `HelloTask` class implementation provides an example task that has one 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";
   }

}
```

Reference

- `org.infinispan.tasks.ServerTask`
- `java.util.concurrent.Callable.call()`
- `java.util.ServiceLoader`

6.1.2. Deploying Server Tasks to Infinispan Servers

Add your custom server task classes to Infinispan servers.

Prerequisites

Stop any running Infinispan servers. Infinispan does not support runtime deployment of custom classes.

Procedure

1. Package your server task implementation in a JAR file.
2. Add a `META-INF/services/org.infinispan.tasks.ServerTask` file that contains the fully qualified names of server tasks, for example:

```
example.HelloTask
```

3. Copy the JAR file to the `$ISPN_HOME/server/lib` directory of your Infinispan server.
4. Add your classes to the deserialization whitelist in your Infinispan configuration. Alternatively set the whitelist using system properties.

Reference

- Adding Java Classes to Deserialization White Lists
- Infinispan 11.0 Configuration Schema

6.2. Creating Server Scripts

Create custom scripts and add them to Infinispan servers.

6.2.1. Server Scripts

Infinispan server scripting is based on the `javax.script` API and is compatible with any JVM-based ScriptEngine implementation. Nashorn is the default JDK ScriptEngine and provides JavaScript capabilities.

Hello World Script Example

The following script provides a simple example that runs on a single Infinispan server, has one parameter, and uses JavaScript:

```
// mode=local,language=javascript,parameters=[greetee]
"Hello " + greetee
```

When you run the preceding script, you pass a value for the `greetee` parameter and Infinispan returns `"Hello ${value}"`.

Script Metadata

Metadata provides additional information about scripts that Infinispan servers use when running scripts.

Script metadata are `property=value` pairs that you add to comments in the first lines of scripts, such as the following example:

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

- Use comment styles that match the scripting language (`//`, `;;`, `#`).
- Separate `property=value` pairs with commas.
- Separate values with single (') or double (") quote characters.

Table 1. Metadata Properties

Property	Description
`mode`	Defines the exection mode and has the following values: `local` only the node that handles the request executes the script. Although scripts can still invoke clustered operations. `distributed` Infinispan uses clustered executors to run scripts across nodes.
`language`	Specifies the ScriptEngine that executes the script.
`extension`	Specifies filename extensions as an alternative method to set the ScriptEngine.
`role`	Specifies roles that users must have to execute scripts.
`parameters`	Specifies an array of valid parameter names for this script. Invocations which specify parameters not included in this list cause exceptions.
`datatype`	Optionally sets the MediaType (MIME type) for storing data as well as parameter and return values. This property is useful for remote clients that support particular data formats only. Currently you can set only `text/plain; charset=utf-8` to use the String UTF-8 format for data.

Script Bindings

Infinispan exposes internal objects as bindings for script execution.

Binding	Description
`cache`	Specifies the cache against which the script is run.
`marshaller`	Specifies the marshaller to use for serializing data to the cache.
`cacheManager`	Specifies the `cacheManager` for the cache.
`scriptingManager`	Specifies the instance of the script manager that runs the script. You can use this binding to run other scripts from a script.

Script Parameters

Infinispan lets you pass named parameters as bindings for running scripts.

Parameters are `name,value` pairs, where `name` is a string and `value` is any value that the marshaller can interpret.

The following example script has two parameters, `multiplicand` and `multiplier`. The script takes the value of `multiplicand` and multiplies it with the value of `multiplier`.

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

When you run the preceding script, Infinispan responds with the result of the expression evaluation.

6.2.2. Adding Scripts to Infinispan Servers

Use the command line interface to add scripts to Infinispan servers.

Prerequisites

Infinispan servers store scripts in the `___script_cache` cache. If you enable cache authorization, users must have the `___script_manager` role to access `___script_cache`.

Procedure

1. Define scripts as required.

   For example, create a file named `multiplication.js` that runs on a single Infinispan server, has two parameters, and uses JavaScript to multiply a given value:

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

2. Open a CLI connection to Infinispan and use the `task` command to upload your scripts as in the following example:

   ```
   [//containers/default]> task upload --file=multiplication.js multiplication
   ```

3. Verify that your scripts are available.

   ```
   [//containers/default]> ls tasks
   multiplication
   ```

6.2.3. Programmatically Creating Scripts

Add scripts with the Hot Rod `RemoteCache` interface as in the following example:

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

Reference

org.infinispan.client.hotrod.RemoteCache

6.3. Running Server-Side Tasks and Scripts

Execute tasks and custom scripts on Infinispan servers.

6.3.1. Running Tasks and Scripts

Use the command line interface to run tasks and scripts on Infinispan servers.

Prerequisites

- Open a CLI connection to Infinispan.

Procedure

Use the `task` command to run tasks and scripts on Infinispan servers, as in the following examples:

- Execute a script named `multipler.js` and specify two parameters:

```
[//containers/default]> task exec multipler.js -Pmultiplicand=10 -Pmultiplier=20
200.0
```

- Execute a task named `@@cache@names` to retrieve a list of all available caches:

```
//containers/default]> task exec @@cache@names
["___protobuf_metadata","mycache","___script_cache"]
```

6.3.2. Programmatically Running Scripts

Call the `execute()` method to run scripts with the Hot Rod `RemoteCache` interface, as in the following example:

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

Reference

org.infinispan.client.hotrod.RemoteCache

6.3.3. Programmatically Running Tasks

Call the `execute()` method to run tasks with the Hot Rod `RemoteCache` interface, as in the following example:

```
// 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());

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

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

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

Reference

org.infinispan.client.hotrod.RemoteCache

Chapter 7. 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.

7.1. 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.

7.2. 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`.

7.3. 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.