# Deploying and Configuring Infinispan 10.1 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 10.1 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 | **N** | **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 JBoss LogManager 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 log handler.

### 4.1.2. 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`.

#### 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. |

#### 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.

#### Root Logger

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

#### 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.

### 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

### 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`

### 4.1.3. 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.

### 4.1.4. 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.

#### 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>
```

#### 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] "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. 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

## 4.2.1. 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.

### 4.2.2. 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.

## 4.3. Collecting Infinispan Metrics

Infinispan servers provide monitoring data through an HTTP `metrics` endpoint that exposes OS, JVM, and application-level statistics.

*Procedure*

1. Start at least one Infinispan server.
2. Test the metrics endpoint with any HTTP client, as in the following examples:

   To get metrics in Prometheus or OpenMetrics format:

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

   To get metrics in Microprofile JSON format:

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

   

   If you configure Infinispan servers with security, you should include the appropriate credentials or client certificates to access the metrics endpoint.

   Infinispan responds with monitoring data.

*Next steps*

Configure monitoring applications to collect Infinispan metrics from the endpoint as appropriate. For example, add the following to your `prometheus.yml` file:

```
static_configs:
    - targets: ['localhost:9090', 'localhost:11222']
```

*Reference*

- Eclipse Microprofile Metrics
- Prometheus Configuration

## 4.4. 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>
```

### 4.4.1. 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

# Chapter 5. Securing Infinispan Servers

## 5.1. Security Concepts

### 5.1.1. Authorization

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

```
<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>
```

### 5.1.2. Server Realms

Infinispan Server security is built around the concept of security realms. Security Realms are used by the server to provide identity, encryption, authentication and authorization information for the server endpoints. A security realm is identified by a name and combines any number of identities and sub-realms, each one also identified by a name. 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).

#### Server identities

The Infinispan Server supports two kinds of identities:

- SSL identities, backed by a *keystore*.
- Kerberos identities, backed by a *keytab* file.

#### SSL identities and endpoint encryption

A SSL identity is represented by a certificate (or a chain of certificates) stored within a *keystore*. Infinispan Server automatically recognizes and supports multiple keystore formats: JKS, JCEKS, PKCS12, BKS, BCFKS and UBER. When the server endpoint is configured with a realm which has an SSL identity it will automatically enable encryption.

*SSL Identity*

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <ssl>
               <keystore path="server.p12" relative-to="infinispan.server.config.path"
keystore-password="secret" alias="server"/>
            </ssl>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

The ideal server certificate is one which is signed by a trusted Certificate Authority (be it a Root or Intermediate CA), and configuring clients to trust that authority. The following sequence of shell commands uses Java's `keytool` to create a CA certificate, a Server certificate signed by the CA and imports both into a keystore to be used by the server.

*Creating certificates with keytool*

```
# Generate a CA certificate and keystore
keytool -genkeypair -validity 365 -keyalg RSA -keysize 2048 -noprompt -storepass
secret -alias ca -dname "CN=CA,OU=Operations,O=Organization,L=ACME" -keystore ca.p12
-ext bc:c

# Export the CA certificate from the keystore
keytool -exportcert -validity 365 -keyalg RSA -keysize 2048 -noprompt -storepass
secret -alias ca -keystore ca.p12 -file ca.cer

# Generate a server certificate and keystore
keytool -genkeypair -validity 365 -keyalg RSA -keysize 2048 -noprompt -storepass
secret -alias server -dname "CN=Server,OU=Operations,O=Organization,L=ACME" -keystore
server.p12

# Create a certificate signing request (CSR) for the server
keytool -certreq -alias server -dname "CN=Server,OU=Operations,O=Organization,L=ACME"
-keystore server.p12 -storepass secret -file server.csr

# Sign the CSR with the CA certificate
keytool -gencert -alias ca -keystore ca.p12 -storepass secret -infile server.csr
-outfile server.cer

# Import the CA certificate to the server keystore
keytool -importcert -alias ca -keystore server.p12 -noprompt -storepass secret -file
ca.cer

# Import the signed server certificate to the server keystore
keytool -importcert -alias server -keystore server.p12 -noprompt -storepass secret
-file server.cer

# Create a client truststore that contains the CA public key
keytool -importcert -alias ca -keystore client_truststore.p12 -storepass secret -file
ca.cer

# Optionally verify that the server keystore contains the CA and server certificates
keytool -list -v -keystore server.p12 -storepass secret
```

Place the `server.p12` file in the server configuration directory.

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) or of the CA that has signed the key of the server you are going to connect to, unless the key was signed by a Certification Authority (CA) trusted by the JRE.

The following code snippet shows how to use the `client_truststore.p12` we generated above in your clients so that they will trust the server's certificate:

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         .trustStoreFileName("client_truststore.p12")
         .trustStorePassword("secret".toCharArray());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

You can require that clients also present a certificate by adding `require-ssl-client-auth="true"` to the `endpoints` element:

*Requiring client certificates*

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
           xmlns="urn:infinispan:server:10.1"
           socket-binding="default" security-realm="default"  require-ssl-client-auth
="true">
   <hotrod-connector name="hotrod"/>
   <rest-connector name="rest" />
</endpoints>
```

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 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <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"/>
            </ssl>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

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 also contain certificates so they might be used in a Hot Rod client directly

The SSL identity also defines the way in which the SSL engine is configured, in terms of available protocols and ciphers. The default configuration should work for most scenarios, but it can be fine-tuned if necessary:

*SSL engine configuration*

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <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" enabled-ciphersuites=
"SSL_RSA_WITH_AES_128_GCM_SHA256 SSL_RSA_WITH_AES_128_CBC_SHA256"/>
            </ssl>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

Make sure you have the correct ciphers for the protocol features you need (e.g. HTTP/2 ALPN)

#### Kerberos identities

A Kerberos identity is represented by a *keytab* file which contains pairs of Kerberos principals and encrypted keys (derived from Kerberos password). While a *keytab* can contain both user and service account principals, in the context of a Infinispan server we only use the latter. This allows the server to identify itself to clients as well as acting as an intermediary authentication service between the clients and the Kerberos server. Since a Kerberos Service Account includes the name of the service in the principal, you usually need to supply one identity per service, i.e. one for HTTP and one for Hot Rod. For example, you may have a `datagrid` server in the `INFINISPAN.ORG` domain, you'd need both a `hotrod/datagrid@INFINISPAN.ORG` service principal for Hot Rod (note the lowercase service name) and an `HTTP/datagrid@INFINISPAN.ORG` service principal for HTTP (note the uppercase service name).

*Kerberos service identities*

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <kerberos keytab-path="hotrod.keytab" principal=
"hotrod/datagrid@INFINISPAN.ORG" required="true"/>
            <kerberos keytab-path="http.keytab" principal=
"HTTP/localhost@INFINISPAN.ORG" required="true"/>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

After creating the service principals in your Kerberos server, you will need to create the relevant *keytab* files:

**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
```

Once you have obtained the keytab files, place them in the server's configuration directory.

#### Property realm

Property realms are the simplest type of realm that you can configure in Infinispan:

*Property Realm configuration*

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <security-realms>
      <security-realm name="default">
         <properties-realm groups-attribute="Roles">
            <user-properties path="users.properties" relative-to=
"infinispan.server.config.path" plain-text="true"/>
            <group-properties path="groups.properties" relative-to=
"infinispan.server.config.path"/>
         </properties-realm>
      </security-realm>
   </security-realms>
</security>
```

They use two property files:

- a users property file which maps user names to passwords. The latter can be either in plain text or hashed (recommended).

  ```
  user1=a_password
  user2=another_password
  ```

- a groups property file which maps users to group names.

  ```
  group1=user1,user2
  group2=user2
  ```

  Infinispan Server comes with a `user-tool.sh` script (`user-tool.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 with an initial set of 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
```

Property realms support the following authentication mechanisms:

- **SASL (Hot Rod)**: `PLAIN`, `DIGEST-*`, `SCRAM-*`

- **HTTP (REST)**: `Basic`, `Digest`

#### LDAP realm

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. Since the layout of entries in an LDAP server varies depending on the type and deployment, the LDAP realm configuration is quite articulate. The following is an example

*LDAP Realm configuration*

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <security-realms>
      <security-realm name="default">
         <ldap-realm name="ldap" url="ldap://my-ldap-server:10389"
                     principal="uid=admin,ou=People,dc=infinispan,dc=org" credential=
"strongPassword"
                     direct-verification="true">
            <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>
            </identity-mapping>
         </ldap-realm>
      </security-realm>
   </security-realms>
</security>
```

The `ldap-realm` element specifies the LDAP connection url as well as a principal and credential to use. The principal must be able to perform LDAP queries and access specific attributes, so it should have the necessary privileges. You can also set additional properties which tune the connection, e.g. timeouts and pooling:

*LDAP Connection configuration*

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <security-realms>
      <security-realm name="default">
         <ldap-realm name="ldap" url="ldap://my-ldap-server:10389"
                     principal="uid=admin,ou=People,dc=infinispan,dc=org" credential=
"strongPassword"
                     connection-timeout="3000" read-timeout="30000" connection-
pooling="true" referral-mode="ignore" page-size="30">
            <name-rewriter>
               <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>
```

It is then necessary to provide a mapping from LDAP entries to identities using the `identity-mapping` element. The first part is to find the user's entry based on a provided identifier, usually a username. The `rdn-identifier` specifies the LDAP attribute which contains the username (e.g. `uid` or `sAMAccountName`). It is also useful to limit the search only to the LDAP subtree which contains the user entries, otherwise it will search the entire tree. This is achieved by passing a starting context in the `search-dn` attribute.

Then we need to decide how to verify a user's credentials. This can be done in two ways:

- by adding the `direct-verification="true"` attribute to the `ldap-realm` element, the realm will try to connect to the LDAP server using the supplied credentials. This is the recommended configuration.
- by adding a `user-password-mapper` element to the `identity-mapping` element specifying which attribute contains the password (e.g. `userPassword`)

Some authentication mechanisms (most notably the Kerberos-based ones: `GSSAPI`, `GS2-KRB5` and `Negotiate`) supply a username which needs to be *cleaned up* before it can be used to search LDAP. In these situation you should add a `name-rewriter` to the `ldap-realm` which extracts the username from

the principal. For example, the following rewriter uses a regular expression and group capturing to remove the domain part:

*LDAP realm with principal rewriting*

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <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>
               <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>
```

Next, we will want to retrieve all the groups a user is a member of, since this information may not be stored as part of a user entry. There are usually two ways in which this is implemented:

- Membership information is stored under group entries (which usually have class `groupOfNames`) in the `member` attribute (this is the typical layout in OpenLDAP). In this case you will need to use an attribute filter as follows:

  ```
  <attribute            filter="(&(objectClass=groupOfNames)(member={1}))"            filter-
  dn="ou=Roles,dc=infinispan,dc=org" from="cn" to="Roles" />
  ```

  The above will search for entries matching the supplied `filter` (which matches groups with a `member` attribute equal to the user's DN) under the `filter-dn`. It will then extract the group entry's CN (as specified by `from`) and add it to the user's `Roles`.

- Membership information is stored in the user entry in the `memberOf` attribute (typical in Microsoft Active Directory). In this case use an attribute reference as follows:

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

The above will get all `memberOf` attributes from the user and extract the CN (as specified by `from`) and add them to the user's `Roles`.

*LDAP realm for Active Directory*

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <security-realms>
      <security-realm name="default">
         <ldap-realm name="ldap" url="ldap://my-ldap-server:10389"
                     principal="CN=LdapAdmin,CN=Users,DC=INFINISPAN,DC=ORG"
credential="strongPassword"
                     direct-verification="true">
            <identity-mapping rdn-identifier="sAMAccountName" search-dn=
"cn=Users,dc=INFINISPAN,dc=ORG">
               <attribute-mapping>
                  <attribute-reference reference="memberOf" from="cn" to="Roles"/>
               </attribute-mapping>
            </identity-mapping>
         </ldap-realm>
      </security-realm>
   </security-realms>
</security>
```

LDAP Realms support the following authentication mechanisms directly:

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

LDAP Realms can also be used to *fill-in* authorization information for users obtained through realms which don't provide this directly (e.g. Trust, Kerberos)

#### Trust Store realm

A Trust Store realm is backed by a keystore containing the public certificates of all the clients you want to allow to connect. It is used with client-certificate authentication mechanisms (EXTERNAL for Hot Rod and CLIENT_CERT for HTTP) so it requires that the server's realm also includes an SSL server identity as well as requiring client certificate authentication on the endpoint.

*Trust Store realm configuration*

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <ssl>
               <keystore path="server.p12" relative-to="infinispan.server.config.path"
keystore-password="secret"
                         alias="server"/>
            </ssl>
         </server-identities>
         <truststore-realm path="trust.p12" relative-to="
infinispan.server.config.path" keystore-password="secret"/>
      </security-realm>
   </security-realms>
</security>
```

You should use `keytool` to import all of the client certificates into the trust store.

#### Token realm

The Token realm relies on external validation services to validate tokens. It requires a provider compatible with RFC-7662 (OAuth2 Token Introspection), such as KeyCloak.

*Token realm configuration with OAuth introspection*

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
          xmlns="urn:infinispan:server:10.1">
   <security-realms>
      <security-realm name="default">
         <token-realm name="token" auth-server-url="https://oauth-server/auth/">
            <oauth2-introspection
                    introspection-url="https://oauth-
server/auth/realms/infinispan/protocol/openid-connect/token/introspect"
                    client-id="infinispan-server" client-secret="1fdca4ec-c416-47e0-
867a-3d471af7050f"/>
         </token-realm>
      </security-realm>
   </security-realms>
</security>
```

The `auth-server-url` attribute on the realm specifies the URL of the authentication server. The `oauth2-introspection` element specifies the `introspection-url` as well as a `client-id` and `client-secret` which will be used for interpreting the tokens.

Token realms support the following authentication mechanisms:

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

## 5.2. 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. It is equivalent to the `BASIC` HTTP mechanism.
- `DIGEST-*` - This family of mechs hashes the credentials before sending them over the wire, so it is more secure than `PLAIN`. The supported implementations, which differ in the hashing algorithm, are: `DIGEST-MD5`, `DIGEST-SHA`, `DIGEST-SHA-256`, `DIGEST-SHA-384` and `DIGEST-SHA-512` ordered by increasing strength. These mechs are comparable to the `DIGEST` HTTP mechanism.
- `SCRAM-*` - This family of mechs is similar to `DIGEST-*`, but increases security by using an additional *salt* for increased security. The supported implementations, which differ in the hashing algorithm, are: `SCRAM-SHA`, `SCRAM-SHA-256`, `SCRAM-SHA-384` and `SCRAM-SHA-512` ordered by increasing strength.
- `GSSAPI` - This mech uses Kerberos tickets, and therefore requires the presence of a properly configured Kerberos Domain Controller (such as Microsoft Active Directory), and a corresponding `kerberos` server identity in the realm. It is also usually used in combination with an `ldap-realm` for retrieving user membership information. These mechs are equivalent with the `SPNEGO` HTTP scheme.
- `GS2-KRB5` - An improvement over `GSSAPI`, it has the same requirements and is configured in the same way.
- `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. It is equivalent to the `CLIENT_CERT` HTTP mechanism.
- `OAUTHBEARER` - This mech enables the use of OAuth tokens as a way to. It requires a token realm. It is equivalent to the `BEARER_TOKEN` HTTP mechanism.

The following configuration enables authentication against the `default` realm, using various SCRAM and DIGEST SASL mechanisms and only enables the **auth** QoP (see SASL Quality of Protection):

*Hot Rod connector configuration*

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
           xmlns="urn:infinispan:server:10.1"
           socket-binding="default" security-realm="default">
   <hotrod-connector name="hotrod">
      <authentication>
         <sasl mechanisms="SCRAM-SHA-512 SCRAM-SHA-384 SCRAM-SHA-256 SCRAM-SHA-1
DIGEST-SHA-512 DIGEST-SHA-384 DIGEST-SHA-256 DIGEST-SHA DIGEST-MD5 PLAIN"
               server-name="infinispan" qop="auth"/>
      </authentication>
   </hotrod-connector>
</endpoints>
```

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` or `GS2-KRB5` mechs as it is equivalent to the Kerberos service name. The Kerberos mechanisms also need to indicate the identity that will be used:

*Hot Rod connector configuration*

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
           xmlns="urn:infinispan:server:10.1"
           socket-binding="default" security-realm="default">
   <hotrod-connector name="hotrod">
      <authentication>
         <sasl mechanisms="GSSAPI GS2-KRB5" server-name="datagrid" server-principal=
"hotrod/datagrid@INFINISPAN.ORG"/>
      </authentication>
   </hotrod-connector>
</endpoints>
```

### 5.2.1. 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 |

| QOP | Description |
|---|---|
| auth-conf | Authentication with integrity and privacy protection |

### 5.2.2. 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. |
| no-dictionary | Specifies that the selected SASL mechanism must not be susceptible to passive dictionary attacks. |
| no-anonymous | Specifies that the selected SASL mechanism must not accept anonymous logins. |

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

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

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

*Hot Rod connector policies*

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

## 5.3. REST Authentication

The REST connector supports authentication by leveraging the HTTP authentication schemes. The supported mechanism names are:

- `BASIC` - This mechanism corresponds to the `Basic` HTTP scheme. This is the most insecure scheme, 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. It is comparable to the `PLAIN` SASL mechanism.
- `DIGEST` - This mechanism corresponds to the `Digest` HTTP scheme. It is more secure than `BASIC` because it uses hashing and a nonce to protect against replay attacks. The server supports multiple hashing algorithms: `SHA-512`, `SHA-256` and `MD5`. It is comparable to the `DIGEST-*` family of SASL mechanisms.
- `SPNEGO` - This mechanism corresponds to the `Negotiate` HTTP scheme and uses Kerberos tickets, and therefore requires the presence of a properly configured Kerberos Domain Controller (such as Microsoft Active Directory), and a corresponding `kerberos` server identity in the realm. It is also usually used in combination with an `ldap-realm` for retrieving user membership information. It is comparable to the `GSSAPI` and `GS2-KRB5` SASL mechanisms.
- `BEARER_TOKEN` - This mechanism corresponds to the `Bearer` HTTP scheme and enables the use of OAuth tokens as a way to authenticate users. It requires a token realm. It is equivalent to the `OAUTHBEARER` SASL mechanism.
- `CLIENT_CERT` - This is a pseudo-mechanism which relies on the client certificate to provide authentication information. It is equivalent to the `EXTERNAL` SASL mechanism.

The following configuration enables authentication against the `default` realm, using both `BASIC` and `DIGEST` mechanisms:

*REST authentication configuration*

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
           xmlns="urn:infinispan:server:10.1"
           socket-binding="default" security-realm="default">
   <rest-connector name="rest">
      <authentication mechanisms="DIGEST BASIC"/>
   </rest-connector>
</endpoints>
```

The `SPNEGO` mechanism also needs to indicate the Kerberos identity that will be used:

*REST authentication configuration for SPNEGO*

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:10.1
https://infinispan.org/schemas/infinispan-server-10.1.xsd"
           xmlns="urn:infinispan:server:10.1"
           socket-binding="default" security-realm="default">
   <rest-connector name="rest">
      <authentication mechanisms="SPNEGO" server-principal=
"HTTP/localhost@INFINISPAN.ORG"/>
   </rest-connector>
</endpoints>
```

# 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 10.1 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:<br><br>`local` only the node that handles the request executes the script. Although scripts can still invoke clustered operations.<br><br>`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.<br><br>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.