Deploying and Configuring Infinispan 12.1 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.

[Get started icon] You can also visit our Get Started with Infinispan tutorial and run the server image in 4 easy steps.

1.1. Infinispan Server Requirements

Infinispan Server requires a Java Virtual Machine and works with Java 11 and later.

Infinispan Server does not support Java 8. However, you can use Java 8 with Hot Rod Java clients.

1.2. Downloading Server Distributions

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

Procedure

1. Download Infinispan 12.1 Server from Infinispan downloads.
2. Run the `sha1sum` command with the server download archive as the argument, for example:

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

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

Reference

The Infinispan Server README, available in the distribution, provides example commands for running the server, describes folders in the `$ISPN_HOME` directory, and lists system properties you can use to customize the filesystem.

1.3. Installing Infinispan Server

Install the Infinispan Server distribution on a host system.

Prerequisites

Download a Infinispan Server distribution archive.

Procedure

- Use any appropriate tool to extract the Infinispan Server archive to the host filesystem.

```
$ unzip infinispan-server-12.1.0.CR2.zip
```

The resulting directory is your `$ISPN_HOME`.

1.4. Creating and Modifying Users

Infinispan Server requires users to authenticate against a default property realm. Before you can access Infinispan Server, you must add credentials by creating at least one user and a password. You can also add and modify the security authorization groups to which users belong.

Procedure

1. Open a terminal in `$ISPN_HOME`.
2. Create and modify Infinispan users with the `user` command.

Run `help user` for more details about using the command.

Creating users and passwords

- Linux

```
$ bin/cli.sh user create myuser -p "qwer1234!"
```

- Microsoft Windows

```
$ bin\cli.bat user create myuser -p "qwer1234!"
```

Creating users with group membership

- Linux

```
$ bin/cli.sh user create myuser -p "qwer1234!" -g supervisor,reader,writer
```

- Microsoft Windows

```
$ bin\cli.bat user create myuser -p "qwer1234!" -g supervisor,reader,writer
```

1.5. Starting Infinispan Server

Run Infinispan Server on your local host.

Prerequisites

- Create at least one Infinispan user.

Procedure

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

Linux

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

Microsoft Windows

```
bin\server.bat
```

Infinispan Server is running successfully when it logs the following messages:

```
ISPN080004: Protocol SINGLE_PORT listening on 127.0.0.1:11222
ISPN080034: Server '...' listening on http://127.0.0.1:11222
ISPN080001: Infinispan Server <version> started in <mm>ms
```

Verification

1. Open `127.0.0.1:11222/console/` in any browser.
2. Enter your credentials at the prompt then continue to Infinispan Console.

1.6. Verifying Cluster Views

Infinispan nodes on the same network automatically discover each other and form clusters.

Complete this procedure to observe cluster discovery with the `MPING` protocol in the default `TCP` stack with locally running Infinispan Server instances. If you want to adjust cluster transport for custom network requirements, see the documentation for setting up Infinispan clusters.

This procedure is intended to demonstrate the principle of cluster discovery and is not intended for production environments. Doing things like specifying a port offset on the command line is not a reliable way to configure cluster transport for production.

Prerequisites

Have one instance of Infinispan Server running.

Procedure

1. Open a terminal in `$ISPN_HOME`.
2. Copy the root directory to `server2`.

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

3. Specify a port offset and the `server2` directory.

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

Verification

You can view cluster membership in the console at `127.0.0.1:11222/console/cluster-membership`.

Infinispan also logs the following messages when nodes 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
```

Reference

Setting Up Infinispan Clusters

1.7. Shutting Down Infinispan Server

Stop individually running servers or bring down clusters gracefully.

Procedure

1. Create a CLI connection to Infinispan.
2. Shut down Infinispan Server in one of the following ways:
   - Stop all nodes in a cluster with the `shutdown cluster` command, for example:

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

     This command saves cluster state to the `data` folder for each node in the cluster. If you use a cache store, the `shutdown cluster` command also persists all data in the cache.

   - Stop individual server instances with the `shutdown server` command and the server hostname, for example:

     ```
     [//containers/default]> shutdown server <my_server01>
     ```

The `shutdown server` command does not wait for rebalancing operations to complete, which can lead to data loss if you specify multiple hostnames at the same time.

Run `help shutdown` for more details about using the command.

Verification

Infinispan logs the following messages when you shut down servers:

```
ISPN080002: Infinispan Server stopping
ISPN000080: Disconnecting JGroups channel cluster
ISPN000390: Persisted state, version=<$version> timestamp=YYYY-MM-DDTHH:MM:SS
ISPN080003: Infinispan Server stopped
```

1.7.1. Restarting Infinispan Clusters

When you bring Infinispan clusters back online after shutting them down, you should wait for the cluster to be available before adding or removing nodes or modifying cluster state.

If you shutdown clustered nodes with the `shutdown server` command, you must restart each server in reverse order.
For example, if you shutdown `server1` and then shutdown `server2`, you should first start `server2` and then start `server1`.

If you shutdown a cluster with the `shutdown cluster` command, clusters become fully operational only after all nodes rejoin.
You can restart nodes in any order but the cluster remains in DEGRADED state until all nodes that were joined before shutdown are running.

1.8. Infinispan Server Filesystem

Infinispan Server uses the following folders on the host filesystem under `$ISPN_HOME`:

```
├─── bin
├─── boot
├─── docs
├─── lib
├─── server
└─── static
```

See the Infinispan Server README, available in the distribution, for descriptions of the each folder in your `$ISPN_HOME` directory as well as system properties you can use to customize the filesystem.

1.8.1. Server Root Directory

Apart from resources in the `bin` and `docs` folders, the only folder under `$ISPN_HOME` that you should interact with is the server root directory, which is named `server` by default.

You can create multiple nodes under the same `$ISPN_HOME` directory or in different directories, but

each Infinispan Server instance must have its own server root directory. For example, a cluster of 5 nodes could have the following server root directories on the filesystem:

```
├──── server
├──── server1
├──── server2
├──── server3
└──── server4
```

Each server root directory should contain the following folders:

```
├──── server
│    ├──── conf
│    ├──── data
│    ├──── lib
│    └──── log
```

`server/conf`

Holds `infinispan.xml` configuration files for a Infinispan Server instance.

Infinispan separates configuration into two layers:

Dynamic

Create mutable cache configurations for data scalability.
Infinispan Server permanently saves the caches you create at runtime along with the cluster state that is distributed across nodes. Each joining node receives a complete cluster state that Infinispan Server synchronizes across all nodes whenever changes occur.

Static

Add configuration to `infinispan.xml` for underlying server mechanisms such as cluster transport, security, and shared datasources.

`server/data`

Provides internal storage that Infinispan Server uses to maintain cluster state.

Never directly delete or modify content in `server/data`.

Modifying files such as `caches.xml` while the server is running can cause corruption. Deleting content can result in an incorrect state, which means clusters cannot restart after shutdown.

`server/lib`

Contains extension `JAR` files for custom filters, custom event listeners, JDBC drivers, custom `ServerTask` implementations, and so on.

`server/log`

Holds Infinispan Server log files.

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 Memcached endpoint is deprecated and planned for removal in a future release.

The Infinispan Memcached endpoint 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 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" security-realm="default"/>
```

- `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/>` declares a Hot Rod connector.
- `<rest-connector/>` declares a REST connector.
- `<memcached-connector socket-binding="memcached"/>` declares a Memcached connector that uses the memcached socket binding.

Declaring an empty `<endpoints/>` element implicitly enables the Hot Rod and REST connectors.

It is possible to have multiple `endpoints` bound to different sockets. These can use different security realms and offer different authentication and encryption configurations. The following configuration enables two endpoints on distinct socket bindings, each one with a dedicated security realm. Additionally, the `public` endpoint disables administrative features, such as the console and CLI.

```
<endpoints socket-binding="public" security-realm="application-realm" admin="false">
   <hotrod-connector/>
   <rest-connector/>
</endpoints>
<endpoints socket-binding="private" security-realm="management-realm">
   <hotrod-connector/>
   <rest-connector/>
</endpoints>
```

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. By default the name is derived from the socket binding name, for example *hotrod-default*.
- `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. By default the name is derived from the socket binding name, for example *rest-default*.
- `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.

3.3. Infinispan Server Ports and Protocols

Infinispan Server exposes endpoints on your network for remote client access.

Port	Protocol	Description
11222	TCP	Hot Rod and REST endpoint
11221	TCP	Memcached endpoint, which is disabled by default.

3.3.1. Configuring Network Firewalls for Remote Connections

Adjust any firewall rules to allow traffic between the server and external clients.

Procedure

On Red Hat Enterprise Linux (RHEL) workstations, for example, you can allow traffic to port 11222 with firewalld as follows:

```
# firewall-cmd --add-port=11222/tcp --permanent
success
# firewall-cmd --list-ports | grep 11222
11222/tcp
```

To configure firewall rules that apply across a network, you can use the nftables utility.

Chapter 4. Securing Access to Infinispan Servers

Configure authentication and encryption mechanisms to secure access to Infinispan servers and protect your data.

4.1. Defining Infinispan Server Security Realms

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

4.1.1. 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.

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

Property realm configuration

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <security-realms>
      <security-realm name="default">
         <!-- Defines groups as roles for server authorization. -->
         <properties-realm groups-attribute="Roles">
            <!-- Specifies the properties file that holds usernames and passwords. -->
            <!-- The plain-text="true" attribute stores passwords in plain text. -->
            <user-properties path="users.properties"
                             relative-to="infinispan.server.config.path"
                             plain-text="true"/>
            <!-- Specifies the properties file that defines roles for users. -->
            <group-properties path="groups.properties"
                              relative-to="infinispan.server.config.path"/>
         </properties-realm>
      </security-realm>
   </security-realms>
</security>
```

Supported authentication mechanisms

Property realms support the following authentication mechanisms:

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

Creating and Modifying Users

Infinispan Server requires users to authenticate against a default property realm. Before you can access Infinispan Server, you must add credentials by creating at least one user and a password. You can also add and modify the security authorization groups to which users belong.

Procedure

1. Open a terminal in `$ISPN_HOME`.
2. Create and modify Infinispan users with the `user` command.

Run `help user` for more details about using the command.

Creating users and passwords

- Linux

```
$ bin/cli.sh user create myuser -p "qwer1234!"
```

- Microsoft Windows

```
$ bin\cli.bat user create myuser -p "qwer1234!"
```

Creating users with group membership

- Linux

```
$ bin/cli.sh user create myuser -p "qwer1234!" -g supervisor,reader,writer
```

- Microsoft Windows

```
$ bin\cli.bat user create myuser -p "qwer1234!" -g supervisor,reader,writer
```

4.1.2. 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. It is beyond the scope of this document to provide examples for all possible configurations.

LDAP realm configuration

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <security-realms>
      <security-realm name="default">
        <!-- Names an LDAP realm and specifies connection properties. -->
        <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"
                    direct-verification="true">
           <!-- Defines how principals are mapped to LDAP entries. -->
           <identity-mapping rdn-identifier="uid"
                             search-dn="ou=People,dc=infinispan,dc=org">
              <!-- Retrieves all the groups of which the user is a member. -->
              <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 principal for LDAP connections must have necessary privileges to perform LDAP queries and access specific attributes.

As an alternative to verifying user credentials with the `direct-verification` attribute, you can specify a LDAP password with the `user-password-mapper` element.

The `rdn-identifier` attribute 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.

The `attribute-mapping` element 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:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <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>
               <!-- Defines a rewriter that extracts the username from the principal
using a regular expression. -->
               <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>
```

4.1.3. Trust Store Realms

Trust store realms use keystores that contain certificates, or certificate chains, that clients must present to connect to Infinispan Server.

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <ssl>
               <!-- Provides an SSL server identity with a keystore that contains
server certificates. -->
               <keystore path="server.p12"
                         relative-to="infinispan.server.config.path"
                         keystore-password="secret"
                         alias="server"/>
               <!-- Provides a trust store that verifies client identities to the
server. -->
               <truststore path="trust.p12"
                           relative-to="infinispan.server.config.path"
                           password="secret"/>
            </ssl>
         </server-identities>
         <!-- Authenticates client identities against the trust store, if required.
-->
         <truststore-realm/>
      </security-realm>
   </security-realms>
</security>
```

If you include the `truststore-realm` element, the trust store must contain public certificates for all clients. If you do not include the `truststore-realm` element, the trust store needs only a certificate chain to verify client identities.

Supported authentication mechanisms

Trust store realms work with client-certificate authentication mechanisms:

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

4.1.4. 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:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <security-realms>
      <security-realm name="default">
         <!-- Specifies the URL of the authentication server. -->
         <token-realm name="token"
                      auth-server-url="https://oauth-server/auth/">
            <!-- Specifies the URL of the token introspection endpoint. -->
            <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>
```

Supported authentication mechanisms

Token realms support the following authentication mechanisms:

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

4.2. Creating Infinispan Server Identities

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

4.2.1. Setting Up TLS/SSL Identities

Use certificates, or chains of certificates, to verify the identity of Infinispan Server to clients.

If security realms contain TLS/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.
5. If required, add a trust store that contains client certificates.

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <ssl>
               <!-- Adds a keystore that verifies the server identity to clients. -->
               <keystore path="server.p12"
                         relative-to="infinispan.server.config.path"
                         keystore-password="secret"
                         alias="server"/>
               <!-- Adds a trust store that verifies client identities to the server.
-->
               <truststore path="trust.p12"
                           relative-to="infinispan.server.config.path"
                           password="secret"/>
            </ssl>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

Credential Keystore Configuration

Review example configurations for credential keystores in Infinispan Server configuration.

Credential keystore

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <!-- Uses a keystore to manage server credentials. -->
   <credential-stores>
      <!-- Specifies the name and filesystem location of a keystore. -->
      <credential-store name="credentials" path="credentials.pfx">
         <!-- Specifies the password for the credential keystore. -->
         <clear-text-credential clear-text="secret1234!"/>
      </credential-store>
   </credential-stores>
</security>
```

Datasource connection

```
<data-sources xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
              xmlns="urn:infinispan:server:12.1">
   <data-source name="postgres" jndi-name="jdbc/postgres">
      <!-- Specifies the database username in the connection factory. -->
      <connection-factory driver="org.postgresql.Driver"
                          username="dbuser"
                          url="${org.infinispan.server.test.postgres.jdbcUrl}">
         <!-- Specifies the credential keystore that contains an encrypted password
and the alias for it. -->
         <credential-reference store="credentials" alias="dbpassword"/>
      </connection-factory>
      <connection-pool max-size="10" min-size="1" background-validation="1000" idle-
removal="1" initial-size="1" leak-detection="10000"/>
   </data-source>
</data-sources>
```

LDAP connection

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <credential-stores>
      <credential-store name="credentials" path="credentials.pfx">
         <clear-text-credential clear-text="secret1234!"/>
      </credential-store>
   </credential-stores>
   <security-realms>
      <security-realm name="default">
         <!-- Specifies the LDAP principal in the connection factory. -->
         <ldap-realm name="ldap" url="ldap://my-ldap-server:10389"
                     principal="uid=admin,ou=People,dc=infinispan,dc=org"
                     connection-timeout="3000"
                     read-timeout="30000"
                     connection-pooling="true"
                     referral-mode="ignore"
                     page-size="30">
            <!-- Specifies the credential keystore that contains an encrypted password
and the alias for it. -->
            <credential-reference store="credentials" alias="ldappassword"/>
         </ldap-realm>
      </security-realm>
   </security-realms>
</security>
```

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:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <ssl>
               <!-- Generates a keystore that includes a self-signed certificate with
the specified hostname. -->
               <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>
```

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:12.1
          https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <ssl>
               <keystore path="server.p12"
                         relative-to="infinispan.server.config.path"
                         keystore-password="secret" alias="server"/>
               <!-- Configures the SSL engine to use TLS v1 and v2 protocols with
specific cipher suites. -->
               <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>
```

4.2.2. 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:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <security-realms>
      <security-realm name="default">
         <server-identities>
            <!-- Specifies a keytab file that provides a Kerberos identity for the Hot
Rod connector. -->
            <!-- Names the Kerberos service principal for the Hot Rod connector. -->
            <!-- The required="true" attribute specifies that the keytab file must be
present when the server starts. -->
            <kerberos keytab-path="hotrod.keytab"
                      principal="hotrod/datagrid@INFINISPAN.ORG"
                      required="true"/>
            <!-- Specifies a keytab file that provides a Kerberos identity for the
REST connector. -->
            <!-- Names the Kerberos service principal for the REST connector. -->
            <kerberos keytab-path="http.keytab"
                      principal="HTTP/localhost@INFINISPAN.ORG"
                      required="true"/>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

4.3. Storing Infinispan Server Credentials in Keystores

External services require credentials to authenticate with Infinispan Server. To protect sensitive text strings such as passwords, add them to a credential keystore rather than directly in Infinispan Server configuration files.

You can then configure Infinispan Server to decrypt passwords for establishing connections with services such as databases or LDAP directories.

Plain-text passwords in `$ISPN_HOME/server/conf` are unencrypted. Any user account with read access to the host filesystem can view plain-text passwords.

While credential keystores are password-protected store encrypted passwords, any user account with write access to the host filesystem can tamper with the keystore itself.

To completely secure Infinispan Server credentials, you should grant read-write access only to user accounts that can configure and run Infinispan Server.

4.3.1. Setting Up Credential Keystores

Create keystores that encrypt credential for Infinispan Server access.

A credential keystore contains at least one alias that is associated with an encrypted password. After you create a keystore, you specify the alias in a connection configuration such as a database connection pool. Infinispan Server then decrypts the password for that alias from the keystore when the service attempts authentication.

You can create as many credential keystores with as many aliases as required.

Procedure

1. Open a terminal in `$ISPN_HOME`.
2. Create a keystore and add credentials to it with the `credentials` command.

By default, keystores are of type PKCS12. Run `help credentials` for details on changing keystore defaults.

The following example shows how to create a keystore that contains an alias of "dbpassword" for the password "changeme". When you create a keystore you also specify a password for the keystore with the `-p` argument.

Linux

```
$ bin/cli.sh credentials add dbpassword -c changeme -p "secret1234!"
```

Microsoft Windows

```
$ bin\cli.bat credentials add dbpassword -c changeme -p "secret1234!"
```

3. Check that the alias is added to the keystore.

```
$ bin/cli.sh credentials ls -p "secret1234!"
dbpassword
```

4. Configure Infinispan to use the credential keystore.
   a. Specify the name and location of the credential keystore in the `credential-stores` configuration.
   b. Provide the credential keystore and alias in the `credential-reference` configuration.

Attributes in the `credential-reference` configuration are optional.

- `store` is required only if you have multiple keystores.
- `alias` is required only if the keystore contains multiple aliases.

Reference

- Credential Keystore Configuration

4.3.2. Credential Keystore Configuration

Review example configurations for credential keystores in Infinispan Server configuration.

Credential keystore

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <!-- Uses a keystore to manage server credentials. -->
   <credential-stores>
      <!-- Specifies the name and filesystem location of a keystore. -->
      <credential-store name="credentials" path="credentials.pfx">
         <!-- Specifies the password for the credential keystore. -->
         <clear-text-credential clear-text="secret1234!"/>
      </credential-store>
   </credential-stores>
</security>
```

Datasource connection

```
<data-sources xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
              xmlns="urn:infinispan:server:12.1">
   <data-source name="postgres" jndi-name="jdbc/postgres">
      <!-- Specifies the database username in the connection factory. -->
      <connection-factory driver="org.postgresql.Driver"
                          username="dbuser"
                          url="${org.infinispan.server.test.postgres.jdbcUrl}">
         <!-- Specifies the credential keystore that contains an encrypted password
and the alias for it. -->
         <credential-reference store="credentials" alias="dbpassword"/>
      </connection-factory>
      <connection-pool max-size="10" min-size="1" background-validation="1000" idle-
removal="1" initial-size="1" leak-detection="10000"/>
   </data-source>
</data-sources>
```

LDAP connection

```
<security xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
          xmlns="urn:infinispan:server:12.1">
   <credential-stores>
      <credential-store name="credentials" path="credentials.pfx">
         <clear-text-credential clear-text="secret1234!"/>
      </credential-store>
   </credential-stores>
   <security-realms>
      <security-realm name="default">
         <!-- Specifies the LDAP principal in the connection factory. -->
         <ldap-realm name="ldap" url="ldap://my-ldap-server:10389"
                     principal="uid=admin,ou=People,dc=infinispan,dc=org"
                     connection-timeout="3000"
                     read-timeout="30000"
                     connection-pooling="true"
                     referral-mode="ignore"
                     page-size="30">
            <!-- Specifies the credential keystore that contains an encrypted password
and the alias for it. -->
            <credential-reference store="credentials" alias="ldappassword"/>
         </ldap-realm>
      </security-realm>
   </security-realms>
</security>
```

4.4. Configuring Endpoint Authentication Mechanisms

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

Infinispan servers require user authentication to access the command line interface (CLI) and console as well as the Hot Rod and REST endpoints. Infinispan servers also automatically configure authentication mechanisms based on the security realms that you define.

4.4.1. Infinispan Server Authentication

Infinispan servers automatically configure authentication mechanisms based on the security realm that you assign to endpoints.

SASL Authentication Mechanisms

The following SASL authentication mechanisms apply to Hot Rod endpoints:

Security Realm	SASL Authentication Mechanism
Property Realms and LDAP Realms	SCRAM-*, DIGEST-*, CRAM-MD5

Security Realm	SASL Authentication Mechanism
Token Realms	OAUTHBEARER
Trust Realms	EXTERNAL
Kerberos Identities	GSSAPI, GS2-KRB5
SSL/TLS Identities	PLAIN

HTTP Authentication Mechanisms

The following HTTP authentication mechanisms apply to REST endpoints:

Security Realm	HTTP Authentication Mechanism
Property Realms and LDAP Realms	DIGEST
Token Realms	BEARER_TOKEN
Trust Realms	CLIENT_CERT
Kerberos Identities	SPNEGO
SSL/TLS Identities	BASIC

Default Configuration

Infinispan servers provide a security realm named "default" that uses a property realm with plain text credentials defined in `$ISPN_HOME/server/ conf/users.properties`, as shown in the following snippet:

```
<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>
```

The `endpoints` configuration assigns the "default" security realm to the Hot Rod and REST connectors, as follows:

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

As a result of the preceding configuration, Infinispan servers require authentication with a mechanism that the property realm supports.

4.4.2. Manually Configuring Hot Rod Authentication

Explicitly configure Hot Rod connector authentication to override the default SASL authentication mechanisms that Infinispan servers use for security realms.

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:12.1
           https://infinispan.org/schemas/infinispan-server-12.1.xsd"
           xmlns="urn:infinispan:server:12.1"
           socket-binding="default"
           security-realm="default">
   <hotrod-connector>
      <authentication>
         <!-- Specifies SASL mechanisms to use for authentication. -->
         <!-- Defines the name that the server declares to clients. -->
         <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>
```

Hot Rod connector with Kerberos authentication

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
           xmlns="urn:infinispan:server:12.1"
           socket-binding="default"
           security-realm="default">
   <hotrod-connector>
      <authentication>
         <!-- Enables the GSSAPI and GS2-KRB5 mechanisms for Kerberos authentication.
-->
         <!-- Defines the server name, which is equivalent to the Kerberos service
name, and specifies the Kerberos identity for the server. -->
         <sasl mechanisms="GSSAPI GS2-KRB5"
               server-name="datagrid"
               server-principal="hotrod/datagrid@INFINISPAN.ORG"/>
      </authentication>
   </hotrod-connector>
</endpoints>
```

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">
      <!-- Specifies multiple SASL authentication mechanisms for the Hot Rod
connector. -->
      <sasl server-name="myhotrodserver"
            mechanisms="PLAIN DIGEST-MD5 GSSAPI EXTERNAL"
            qop="auth">
         <!-- Defines policies for SASL mechanisms. -->
         <policy>
            <no-active value="true" />
            <no-anonymous value="true" />
            <no-plain-text value="true" />
         </policy>
      </sasl>
   </authentication>
</hotrod-connector>
```

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.

4.4.3. Manually Configuring REST Authentication

Explicitly configure REST connector authentication to override the default HTTP authentication mechanisms that Infinispan servers use for security realms.

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 authentication mechanisms for the REST endpoint to use.

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:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
           xmlns="urn:infinispan:server:12.1"
           socket-binding="default"
           security-realm="default">
   <rest-connector>
      <!-- Specifies SASL mechanisms to use for authentication. -->
      <authentication mechanisms="DIGEST BASIC"/>
   </rest-connector>
</endpoints>
```

REST connector with Kerberos authentication

```
<endpoints xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="urn:infinispan:server:12.1
https://infinispan.org/schemas/infinispan-server-12.1.xsd"
           xmlns="urn:infinispan:server:12.1"
           socket-binding="default"
           security-realm="default">
   <rest-connector>
      <!-- Enables the `SPENGO` mechanism for Kerberos authentication and specifies an
identity for the server. -->
      <authentication mechanisms="SPNEGO"
                      server-principal="HTTP/localhost@INFINISPAN.ORG"/>
   </rest-connector>
</endpoints>
```

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.

4.4.4. Disabling Infinispan Server Authentication

In local development environments or on isolated networks you can configure Infinispan servers to allow unauthenticated client requests.

Procedure

1. Remove any security-realm attributes from the endpoints configuration.
2. Ensure that the Hot Rod and REST connectors do not include any authentication definitions.

For example, the following configuration allows unauthenticated access to Infinispan:

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

4.5. Endpoint IP Filtering

Configure IP Filtering rules on the endpoints to accept or reject connections based on the client address.

4.5.1. Infinispan Server IP Filter Configuration

Infinispan endpoints and connectors can specify one or more IP filtering rules. These rules specify the type of action to take when a client which matches a supplied CIDR block connects. IP filtering rules are applied in order up until the first one that matches.

A CIDR block is a compact representation of an IP address and its associated network mask. CIDR notation specifies an IP address, a slash ('/') character, and a decimal number. The decimal number is the count of leading 1 bits in the network mask. The number can also be thought of as the width, in bits, of the network prefix. The IP address in CIDR notation is always represented according to the standards for IPv4 or IPv6.

The address can denote a specific interface address, including a host identifier, such as `10.0.0.1/8`, or it can be the beginning address of an entire network interface range using a host identifier of 0, as in `10.0.0.0/8` or `10/8`.

For example:

- `192.168.100.14/24` represents the IPv4 address `192.168.100.14` and its associated network prefix `192.168.100.0`, or equivalently, its subnet mask `255.255.255.0`, which has 24 leading 1-bits.
- the IPv4 block `192.168.100.0/22` represents the 1024 IPv4 addresses from `192.168.100.0` to `192.168.103.255`.
- the IPv6 block `2001:db8::/48` represents the block of IPv6 addresses from `2001:db8:0:0:0:0:0:0` to `2001:db8:0:ffff:ffff:ffff:ffff:ffff`.
- `::1/128` represents the IPv6 loopback address. Its prefix length is 128 which is the number of bits in the address.

```
<endpoints socket-binding="default" security-realm="default">
  <ip-filter>
    <accept from="192.168.0.0/16"/>
    <accept from="10.0.0.0/8"/>
    <reject from="/0"/>
  </ip-filter>
  <hotrod-connector name="hotrod"/>
  <rest-connector name="rest"/>
</endpoints>
```

As a result of the preceding configuration, Infinispan servers accept connections only from addresses in the `192.168.0.0/16` and `10.0.0.0/8` CIDR blocks. Infinispan servers reject all other connections.

4.5.2. Inspecting and Modifying Infinispan Server IP Filter Rules

Server IP filter rules can be manipulated via the CLI.

Procedure

1. Open a terminal in `$ISPN_HOME`.

2. Inspect and modify the IP filter rules `server connector ipfilter` command as required.
   a. List all IP filtering rules active on a connector across the cluster:

```
[//containers/default]> server connector ipfilter ls endpoint-default
```

   b. Set IP filtering rules across the cluster.

This command replaces any existing rules.

```
[//containers/default]> server connector ipfilter set endpoint-default
--rules=ACCEPT/192.168.0.0/16,REJECT/10.0.0.0/8`
```

   c. Remove all IP filtering rules on a connector across the cluster.

```
[//containers/default]> server connector ipfilter clear endpoint-default
```

4.6. Configuring Security Authorization

Authorization restricts the ability to perform operations with Infinispan and access data. You assign users with roles that have different permission levels.

4.6.1. Restricting Access to Caches

Control access to caches with Infinispan authorization (RBAC).

This procedure shows you how to use the default Infinispan roles and permissions that are suitable for most use cases.

Procedure

1. Open your `infinispan.xml` configuration for editing.
2. If it is not already declared, add the `<authorization />` tag inside the `security` elements for the `cache-container`.

   This enables authorization for the Cache Manager and provides a global set of roles and permissions that caches can inherit.

3. Add the `<authorization />` tag to each cache for which Infinispan restricts access based on user roles.

The following configuration example shows how to use implicit authorization configuration with default roles and permissions:

```
<infinispan>
   <cache-container default-cache="rbac-cache" name="restricted">
      <security>
         <!-- Enable authorization with the default roles and permissions. -->
         <authorization />
      </security>
      <local-cache name="rbac-cache">
         <security>
            <!-- Inherit authorization settings from the cache-container. -->
            <authorization/>
         </security>
      </local-cache>
   </cache-container>
</infinispan>
```

4.6.2. Default Roles and Permissions

Infinispan includes a default set of default roles and permissions for security authorization.

Default roles and permissions come from the cluster role mapper that associates each role to a permission that authorizes operations on Cache Managers and caches.

Use the Infinispan CLI to dynamically edit these roles with the `user role` command.

Run `help user` for command usage examples.

Role	Permissions
`admin`	Superuser with all permissions including Cache Managers.
`deployer`	Access and modify resources such as caches and counters. Create and delete resources such as caches, counters, schemas, and scripts.
`application`	Access and modify resources such as caches and counters.
`observer`	Read-only access to the system.

Reference

- Infinispan Configuration Schema Reference

4.6.3. How Security Authorization Works

Infinispan authorization secures your installation by restricting user access.

User applications or clients must belong to a role that is assigned with sufficient permissions before they can perform operations on Cache Managers or caches.

For example, you configure authorization on a specific cache instance so that invoking `Cache.get()` requires an identity to be assigned a role with read permission while `Cache.put()` requires a role with write permission.

In this scenario, if a user application or client with the `io` role attempts to write an entry, Infinispan denies the request and throws a security exception. If a user application or client with the `writer` role sends a write request, Infinispan validates authorization and issues a token for subsequent operations.

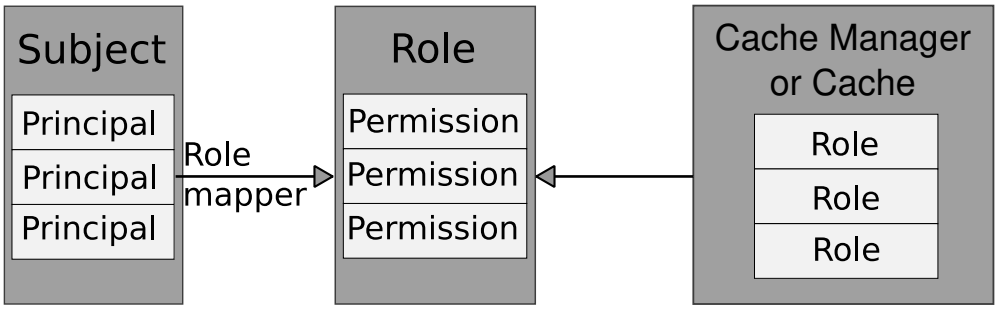
Identities

Identities are security Principals of type `java.security.Principal`. Subjects, implemented with the `javax.security.auth.Subject` class, represent a group of security Principals. In other words, a Subject represents a user and all groups to which it belongs.

Identities to roles

Infinispan uses role mappers so that security principals correspond to roles, which you assign one or more permissions.

The following image illustrates how security principals correspond to roles:



Permissions

Authorization roles have different permissions with varying levels of access to Infinispan. Permissions let you restrict user access to both Cache Managers and caches.

Cache Manager permissions

Permission	Function	Description
CONFIGURATION	`defineConfiguration`	Defines new cache configurations.
LISTEN	`addListener`	Registers listeners against a Cache Manager.
LIFECYCLE	`stop`	Stops the Cache Manager.
CREATE	`createCache`, `removeCache`	Create and remove container resources such as caches, counters, schemas, and scripts.
ALL	-	Includes all Cache Manager permissions.

Cache permissions

Permission	Function	Description
READ	get, contains	Retrieves entries from a cache.
WRITE	put, putIfAbsent, replace, remove, evict	Writes, replaces, removes, evicts data in a cache.
EXEC	distexec, streams	Allows code execution against a cache.
LISTEN	addListener	Registers listeners against a cache.
BULK_READ	keySet, values, entrySet, query	Executes bulk retrieve operations.
BULK_WRITE	clear, putAll	Executes bulk write operations.
LIFECYCLE	start, stop	Starts and stops a cache.
ADMIN	getVersion, addInterceptor*, removeInterceptor, getInterceptorChain, getEvictionManager, getComponentRegistry, getDistributionManager, getAuthorizationManager, evict, getRpcManager, getCacheConfiguration, getCacheManager, getInvocationContextContainer, setAvailability, getDataContainer, getStats, getXAResource	Allows access to underlying components and internal structures.
ALL	-	Includes all cache permissions.
ALL_READ	-	Combines the READ and BULK_READ permissions.
ALL_WRITE	-	Combines the WRITE and BULK_WRITE permissions.

Reference

- Infinispan Security API

Role Mappers

Infinispan includes a `PrincipalRoleMapper` API that maps security Principals in a Subject to authorization roles that you can assign to users.

Cluster role mappers

`ClusterRoleMapper` uses a persistent replicated cache to dynamically store principal-to-role mappings for the default roles and permissions.

By default uses the Principal name as the role name and implements `org.infinispan.security.MutableRoleMapper` which exposes methods to change role mappings at runtime.

- Java class: `org.infinispan.security.mappers.ClusterRoleMapper`
- Declarative configuration: `<cluster-role-mapper />`

Identity role mappers

`IdentityRoleMapper` uses the Principal name as the role name.

- Java class: `org.infinispan.security.mappers.IdentityRoleMapper`
- Declarative configuration: `<identity-role-mapper />`

CommonName role mappers

`CommonNameRoleMapper` uses the Common Name (CN) as the role name if the Principal name is a Distinguished Name (DN).

For example this DN, `cn=managers,ou=people,dc=example,dc=com`, maps to the `managers` role.

- Java class: `org.infinispan.security.mappers.CommonRoleMapper`
- Declarative configuration: `<common-name-role-mapper />`

Custom role mappers

Custom role mappers are implementations of `org.infinispan.security.PrincipalRoleMapper`.

- Declarative configuration: `<custom-role-mapper class="my.custom.RoleMapper" />`

Reference

- Infinispan Security API
- org.infinispan.security.PrincipalRoleMapper

4.6.4. Customizing Roles and Permissions

You can customize authorization settings in your Infinispan configuration to use role mappers with different combinations of roles and permissions.

Procedure

1. Open your `infinispan.xml` configuration for editing.
2. Configure authorization for the `cache-container` by declaring a role mapper and a set of roles and permissions.
3. Configure authorization for caches to restrict access based on user roles.

The following configuration example shows how to configure security authorization with roles and permissions:

```
<infinispan>
   <cache-container default-cache="restricted" name="custom-authorization">
      <security>
         <authorization>
            <!-- Declare a role mapper that associates a security principal to each
role. -->
            <identity-role-mapper />
            <!-- Specify user roles and corresponding permissions. -->
            <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="implicit-authorization">
         <security>
            <!-- Inherit roles and permissions from the cache-container. -->
            <authorization/>
         </security>
      </local-cache>
      <local-cache name="restricted">
         <security>
            <!-- Explicitly define which roles can access the cache. -->
            <authorization roles="admin supervisor"/>
         </security>
      </local-cache>
   </cache-container>
</infinispan>
```

Chapter 5. Setting Up Infinispan Clusters

Infinispan requires a transport layer so nodes can automatically join and leave clusters. The transport layer also enables Infinispan nodes to replicate or distribute data across the network and perform operations such as re-balancing and state transfer.

5.1. Getting Started with Default Stacks

Infinispan uses JGroups protocol stacks so nodes can send each other messages on dedicated cluster channels.

Infinispan provides preconfigured JGroups stacks for `UDP` and `TCP` protocols. You can use these default stacks as a starting point for building custom cluster transport configuration that is optimized for your network requirements.

Procedure

1. Locate the default JGroups stacks, `default-jgroups-*.xml`, in the `default-configs` directory inside the `infinispan-core-12.1.0.CR2.jar` file.

   The `jar` file is in the `$ISPN_HOME/lib` directory.

2. Do one of the following:

   - Use the `stack` attribute in your `infinispan.xml` file.

     ```
     <infinispan>
       <cache-container default-cache="replicatedCache">
         <!-- Use the default UDP stack for cluster transport. -->
         <transport cluster="${infinispan.cluster.name}"
                    stack="udp"
                    node-name="${infinispan.node.name:}"/>
       </cache-container>
     </infinispan>
     ```

   - Use the `cluster-stack` argument when you start the server:

     ```
     $ bin/server.sh --cluster-stack=udp
     ```

Infinispan logs the following message to indicate which stack it uses:

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack udp
```

5.1.1. Default JGroups Stacks

Learn about default JGroups stacks that configure cluster transport.

File name	Stack name	Description
default-jgroups-udp.xml	udp	Uses UDP for transport and UDP multicast for discovery. Suitable for larger clusters (over 100 nodes) or if you are using replicated caches or invalidation mode. Minimizes the number of open sockets.
default-jgroups-tcp.xml	tcp	Uses TCP for transport and the MPING protocol for discovery, which uses UDP multicast. Suitable for smaller clusters (under 100 nodes) *only if* you are using distributed caches because TCP is more efficient than UDP as a point-to-point protocol.
default-jgroups-ec2.xml	ec2	Uses TCP for transport and S3_PING for discovery. Suitable for Amazon EC2 nodes where UDP multicast is not available.
default-jgroups-kubernetes.xml	kubernetes	Uses TCP for transport and DNS_PING for discovery. Suitable for Kubernetes and Red Hat OpenShift nodes where UDP multicast is not always available.
default-jgroups-google.xml	google	Uses TCP for transport and GOOGLE_PING2 for discovery. Suitable for Google Cloud Platform nodes where UDP multicast is not available.
default-jgroups-azure.xml	azure	Uses TCP for transport and AZURE_PING for discovery. Suitable for Microsoft Azure nodes where UDP multicast is not available.

Reference

- JGroups Protocols

5.1.2. TCP and UDP Ports for Cluster Traffic

Infinispan uses the following ports for cluster transport messages:

Default Port	Protocol	Description
7800	TCP/UDP	JGroups cluster bind port
46655	UDP	JGroups multicast

Cross-Site Replication

Infinispan uses the following ports for the JGroups RELAY2 protocol:

`7900`
: For Infinispan clusters running on Kubernetes.

`7800`

If using UDP for traffic between nodes and TCP for traffic between clusters.

`7801`

If using TCP for traffic between nodes and TCP for traffic between clusters.

5.2. Customizing JGroups Stacks

Adjust and tune properties to create a cluster transport configuration that works for your network requirements.

Infinispan provides attributes that let you extend the default JGroups stacks for easier configuration. You can inherit properties from the default stacks while combining, removing, and replacing other properties.

Procedure

1. Create a new JGroups stack declaration in your `infinispan.xml` file.
2. Add the `extends` attribute and specify a JGroups stack to inherit properties from.
3. Use the `stack.combine` attribute to modify properties for protocols configured in the inherited stack.
4. Use the `stack.position` attribute to define the location for your custom stack.
5. Specify the stack name as the value for the `stack` attribute in the `transport` configuration.

   For example, you might evaluate using a Gossip router and symmetric encryption with the default TCP stack as follows:

```
<infinispan>
  <jgroups>
    <!-- Creates a custom JGroups stack named "my-stack". -->
    <!-- Inherits properties from the default TCP stack. -->
    <stack name="my-stack" extends="tcp">
      <!-- Uses TCPGOSSIP as the discovery mechanism instead of MPING -->
      <TCPGOSSIP initial_hosts=
"${jgroups.tunnel.gossip_router_hosts:localhost[12001]}"
             stack.combine="REPLACE"
             stack.position="MPING" />
      <!-- Removes the FD_SOCK protocol from the stack. -->
      <FD_SOCK stack.combine="REMOVE"/>
      <!-- Modifies the timeout value for the VERIFY_SUSPECT protocol. -->
      <VERIFY_SUSPECT timeout="2000"/>
      <!-- Adds SYM_ENCRYPT to the stack after VERIFY_SUSPECT. -->
      <SYM_ENCRYPT sym_algorithm="AES"
                   keystore_name="mykeystore.p12"
                   keystore_type="PKCS12"
                   store_password="changeit"
                   key_password="changeit"
                   alias="myKey"
                   stack.combine="INSERT_AFTER"
                   stack.position="VERIFY_SUSPECT" />
    </stack>
    <cache-container name="default" statistics="true">
      <!-- Uses "my-stack" for cluster transport. -->
      <transport cluster="${infinispan.cluster.name}"
                 stack="my-stack"
                 node-name="${infinispan.node.name:}"/>
   </cache-container>
  </jgroups>
</infinispan>
```

6. Check Infinispan logs to ensure it uses the stack.

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack
my-stack
```

5.2.1. Inheritance Attributes

When you extend a JGroups stack, inheritance attributes let you adjust protocols and properties in the stack you are extending.

- `stack.position` specifies protocols to modify.
- `stack.combine` uses the following values to extend JGroups stacks:

Value	Description
COMBINE	Overrides protocol properties.
REPLACE	Replaces protocols.
INSERT_AFTER	Adds a protocol into the stack after another protocol. Does not affect the protocol that you specify as the insertion point. Protocols in JGroups stacks affect each other based on their location in the stack. For example, you should put a protocol such as NAKACK2 after the SYM_ENCRYPT or ASYM_ENCRYPT protocol so that NAKACK2 is secured.
INSERT_BEFORE	Inserts a protocols into the stack before another protocol. Affects the protocol that you specify as the insertion point.
REMOVE	Removes protocols from the stack.

5.3. Using JGroups System Properties

Pass system properties to Infinispan at startup to tune cluster transport.

Procedure

- Use `-D<property-name>=<property-value>` arguments to set JGroups system properties as required.

For example, set a custom bind port and IP address as follows:

```
$ bin/server.sh -Djgroups.bind.port=1234 -Djgroups.bind.address=192.0.2.0
```

5.3.1. System Properties for JGroups Stacks

Set system properties that configure JGroups cluster transport stacks.

System Property	Description	Default Value	Required/Optional
jgroups.bind.address	Bind address for cluster transport.	SITE_LOCAL	Optional
jgroups.bind.port	Bind port for the socket.	7800	Optional
jgroups.mcast_addr	IP address for multicast, both discovery and inter-cluster communication. The IP address must be a valid "class D" address that is suitable for IP multicast.	228.6.7.8	Optional
jgroups.mcast_port	Port for the multicast socket.	46655	Optional

System Property	Description	Default Value	Required/Optional
jgroups.ip_ttl	Time-to-live (TTL) for IP multicast packets. The value defines the number of network hops a packet can make before it is dropped.	2	Optional
jgroups.thread_pool.min_threads	Minimum number of threads for the thread pool.	0	Optional
jgroups.thread_pool.max_threads	Maximum number of threads for the thread pool.	200	Optional
jgroups.join_timeout	Maximum number of milliseconds to wait for join requests to succeed.	2000	Optional
jgroups.thread_dumps_threshold	Number of times a thread pool needs to be full before a thread dump is logged.	10000	Optional

Amazon EC3

The following system properties apply only to `default-jgroups-ec2.xml`:

System Property	Description	Default Value	Required/Optional
jgroups.s3.access_key	Amazon S3 access key for an S3 bucket.	No default value.	Optional
jgroups.s3.secret_access_key	Amazon S3 secret key used for an S3 bucket.	No default value.	Optional
jgroups.s3.bucket	Name of the Amazon S3 bucket. The name must exist and be unique.	No default value.	Optional

Kubernetes

The following system properties apply only to `default-jgroups-kubernetes.xml`:

System Property	Description	Default Value	Required/Optional
jgroups.dns.query	Sets the DNS record that returns cluster members.	No default value.	Required

Google Cloud Platform

The following system properties apply only to `default-jgroups-google.xml`:

System Property	Description	Default Value	Required/Optional
`jgroups.google.bucket_name`	Name of the Google Compute Engine bucket. The name must exist and be unique.	No default value.	Required

Reference

- JGroups System Properties
- JGroups Protocol List

5.4. Using Inline JGroups Stacks

You can insert complete JGroups stack definitions into `infinispan.xml` files.

Procedure

- Embed a custom JGroups stack declaration in your `infinispan.xml` file.

```
<infinispan>
  <!-- Contains one or more JGroups stack definitions. -->
  <jgroups>
    <!-- Defines a custom JGroups stack named "prod". -->
    <stack name="prod">
      <TCP bind_port="7800" port_range="30" recv_buf_size="20000000" send_buf_size
="640000"/>
      <MPING break_on_coord_rsp="true"
             mcast_addr="${jgroups.mping.mcast_addr:228.2.4.6}"
             mcast_port="${jgroups.mping.mcast_port:43366}"
             num_discovery_runs="3"
             ip_ttl="${jgroups.udp.ip_ttl:2}"/>
      <MERGE3 />
      <FD_SOCK />
      <FD_ALL timeout="3000" interval="1000" timeout_check_interval="1000" />
      <VERIFY_SUSPECT timeout="1000" />
      <pbcast.NAKACK2 use_mcast_xmit="false" xmit_interval="100"
xmit_table_num_rows="50"
                      xmit_table_msgs_per_row="1024"
xmit_table_max_compaction_time="30000" />
      <UNICAST3 xmit_interval="100" xmit_table_num_rows="50"
xmit_table_msgs_per_row="1024"
                xmit_table_max_compaction_time="30000" />
      <pbcast.STABLE stability_delay="200" desired_avg_gossip="2000" max_bytes="1M"
/>
      <pbcast.GMS print_local_addr="false" join_timeout=
"${jgroups.join_timeout:2000}" />
      <UFC max_credits="4m" min_threshold="0.40" />
      <MFC max_credits="4m" min_threshold="0.40" />
      <FRAG3 />
    </stack>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Uses "prod" for cluster transport. -->
    <transport cluster="${infinispan.cluster.name}"
           stack="prod"
           node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

5.5. Using External JGroups Stacks

Reference external files that define custom JGroups stacks in `infinispan.xml` files.

Procedure

1. Add custom JGroups stack files to the `$ISPN_HOME/server/conf` directory.

   Alternatively you can specify an absolute path when you declare the external stack file.

2. Reference the external stack file with the `stack-file` element.

```
<infinispan>
  <jgroups>
     <!-- Creates a "prod-tcp" stack that references an external file. -->
     <stack-file name="prod-tcp" path="prod-jgroups-tcp.xml"/>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Use the "prod-tcp" stack for cluster transport. -->
    <transport stack="prod-tcp" />
    <replicated-cache name="replicatedCache"/>
  </cache-container>
  ...
</infinispan>
```

5.6. Cluster Discovery Protocols

Infinispan supports different protocols that allow nodes to automatically find each other on the network and form clusters.

There are two types of discovery mechanisms that Infinispan can use:

- Generic discovery protocols that work on most networks and do not rely on external services.
- Discovery protocols that rely on external services to store and retrieve topology information for Infinispan clusters.
  For instance the DNS_PING protocol performs discovery through DNS server records.

Running Infinispan on hosted platforms requires using discovery mechanisms that are adapted to network constraints that individual cloud providers impose.

Reference

- JGroups Discovery Protocols

5.6.1. PING

PING, or UDPPING is a generic JGroups discovery mechanism that uses dynamic multicasting with the UDP protocol.

When joining, nodes send PING requests to an IP multicast address to discover other nodes already in the Infinispan cluster. Each node responds to the PING request with a packet that contains the address of the coordinator node and its own address. C=coordinator's address and A=own address. If no nodes respond to the PING request, the joining node becomes the coordinator node in a new cluster.

PING configuration example

```
<config>
  <PING num_discovery_runs="3"/>
  ...
</config>
```

Reference

- JGroups PING

5.6.2. TCPPING

TCPPING is a generic JGroups discovery mechanism that uses a list of static addresses for cluster members.

With TCPPING, you manually specify the IP address or hostname of each node in the Infinispan cluster as part of the JGroups stack, rather than letting nodes discover each other dynamically.

TCPPING configuration example

```
<config>
  <TCP bind_port="7800" />
  <TCPPING timeout="3000"
           initial_hosts=
"${jgroups.tcpping.initial_hosts:hostname1[port1],hostname2[port2]}"
          port_range="0"
          num_initial_members="3"/>
  ...
</config>
```

Reference

- JGroups TCPPING

5.6.3. MPING

MPING uses IP multicast to discover the initial membership of Infinispan clusters.

You can use MPING to replace TCPPING discovery with TCP stacks and use multicasing for discovery instead of static lists of initial hosts. However, you can also use MPING with UDP stacks.

MPING configuration example

```
<config>
  <MPING mcast_addr="${jgroups.mcast_addr:228.6.7.8}"
         mcast_port="${jgroups.mcast_port:46655}"
         num_discovery_runs="3"
         ip_ttl="${jgroups.udp.ip_ttl:2}"/>
  ...
</config>
```

Reference

- JGroups MPING

5.6.4. TCPGOSSIP

Gossip routers provide a centralized location on the network from which your Infinispan cluster can retrieve addresses of other nodes.

You inject the address (`IP:PORT`) of the Gossip router into Infinispan nodes as follows:

1. Pass the address as a system property to the JVM; for example, `-DGossipRouterAddress="10.10.2.4[12001]"`.
2. Reference that system property in the JGroups configuration file.

Gossip router configuration example

```
<config>
  <TCP bind_port="7800" />
  <TCPGOSSIP timeout="3000"
             initial_hosts="${GossipRouterAddress}"
             num_initial_members="3" />
  ...
</config>
```

Reference

- JGroups Gossip Router

5.6.5. JDBC_PING

JDBC_PING uses shared databases to store information about Infinispan clusters. This protocol supports any database that can use a JDBC connection.

Nodes write their IP addresses to the shared database so joining nodes can find the Infinispan cluster on the network. When nodes leave Infinispan clusters, they delete their IP addresses from the shared database.

JDBC_PING configuration example

```
<config>
  <JDBC_PING connection_url="jdbc:mysql://localhost:3306/database_name"
             connection_username="user"
             connection_password="password"
             connection_driver="com.mysql.jdbc.Driver"/>
  ...
</config>
```

Add the appropriate JDBC driver to the classpath so Infinispan can use JDBC_PING.

Reference

- JDBC_PING
- JDBC_PING Wiki

5.6.6. DNS_PING

JGroups DNS_PING queries DNS servers to discover Infinispan cluster members in Kubernetes environments such as OKD and Red Hat OpenShift.

DNS_PING configuration example

```
<config>
  <dns.DNS_PING dns_query="myservice.myproject.svc.cluster.local" />
  ...
</config>
```

Reference

- JGroups DNS_PING
- DNS for Services and Pods (Kubernetes documentation for adding DNS entries)

5.7. Encrypting Cluster Transport

Secure cluster transport so that nodes communicate with encrypted messages. You can also configure Infinispan clusters to perform certificate authentication so that only nodes with valid identities can join.

5.7.1. Infinispan Cluster Security

To secure cluster traffic, you configure Infinispan nodes to encrypt JGroups message payloads with secret keys.

Infinispan nodes can obtain secret keys from either:

- The coordinator node (asymmetric encryption).

- A shared keystore (symmetric encryption).

Retrieving secret keys from coordinator nodes

You configure asymmetric encryption by adding the `ASYM_ENCRYPT` protocol to a JGroups stack in your Infinispan configuration. This allows Infinispan clusters to generate and distribute secret keys.

When using asymmetric encryption, you should also provide keystores so that nodes can perform certificate authentication and securely exchange secret keys. This protects your cluster from man-in-the-middle (MitM) attacks.

Asymmetric encryption secures cluster traffic as follows:

1. The first node in the Infinispan cluster, the coordinator node, generates a secret key.
2. A joining node performs certificate authentication with the coordinator to mutually verify identity.
3. The joining node requests the secret key from the coordinator node. That request includes the public key for the joining node.
4. The coordinator node encrypts the secret key with the public key and returns it to the joining node.
5. The joining node decrypts and installs the secret key.
6. The node joins the cluster, encrypting and decrypting messages with the secret key.

Retrieving secret keys from shared keystores

You configure symmetric encryption by adding the `SYM_ENCRYPT` protocol to a JGroups stack in your Infinispan configuration. This allows Infinispan clusters to obtain secret keys from keystores that you provide.

1. Nodes install the secret key from a keystore on the Infinispan classpath at startup.
2. Node join clusters, encrypting and decrypting messages with the secret key.

Comparison of asymmetric and symmetric encryption

`ASYM_ENCRYPT` with certificate authentication provides an additional layer of encryption in comparison with `SYM_ENCRYPT`. You provide keystores that encrypt the requests to coordinator nodes for the secret key. Infinispan automatically generates that secret key and handles cluster traffic, while letting you specify when to generate secret keys. For example, you can configure clusters to generate new secret keys when nodes leave. This ensures that nodes cannot bypass certificate authentication and join with old keys.

`SYM_ENCRYPT`, on the other hand, is faster than `ASYM_ENCRYPT` because nodes do not need to exchange keys with the cluster coordinator. A potential drawback to `SYM_ENCRYPT` is that there is no configuration to automatically generate new secret keys when cluster membership changes. Users are responsible for generating and distributing the secret keys that nodes use to encrypt cluster traffic.

5.7.2. Configuring Cluster Transport with Asymmetric Encryption

Configure Infinispan clusters to generate and distribute secret keys that encrypt JGroups messages.

Procedure

1. Create a keystore with certificate chains that enables Infinispan to verify node identity.
2. Place the keystore on the classpath for each node in the cluster.

   For Infinispan Server, you put the keystore in the $ISPN_HOME directory.

3. Add the `SSL_KEY_EXCHANGE` and `ASYM_ENCRYPT` protocols to a JGroups stack in your Infinispan configuration, as in the following example:

```
<infinispan>
  <jgroups>
    <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the
default TCP stack. -->
    <stack name="encrypt-tcp" extends="tcp">
      <!-- Adds a keystore that nodes use to perform certificate authentication.
-->
      <!-- Uses the stack.combine and stack.position attributes to insert
SSL_KEY_EXCHANGE into the default TCP stack after VERIFY_SUSPECT. -->
      <SSL_KEY_EXCHANGE keystore_name="mykeystore.jks"
                        keystore_password="changeit"
                        stack.combine="INSERT_AFTER"
                        stack.position="VERIFY_SUSPECT"/>
      <!-- Configures ASYM_ENCRYPT -->
      <!-- Uses the stack.combine and stack.position attributes to insert
ASYM_ENCRYPT into the default TCP stack before pbcast.NAKACK2. -->
      <!-- The use_external_key_exchange = "true" attribute configures nodes to use
the `SSL_KEY_EXCHANGE` protocol for certificate authentication. -->
      <ASYM_ENCRYPT asym_keylength="2048"
                    asym_algorithm="RSA"
                    change_key_on_coord_leave = "false"
                    change_key_on_leave = "false"
                    use_external_key_exchange = "true"
                    stack.combine="INSERT_BEFORE"
                    stack.position="pbcast.NAKACK2"/>
    </stack>
  </jgroups>
  <cache-container name="default" statistics="true">
    <!-- Configures the cluster to use the JGroups stack. -->
    <transport cluster="${infinispan.cluster.name}"
               stack="encrypt-tcp"
               node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

Verification

When you start your Infinispan cluster, the following log message indicates that the cluster is using the secure JGroups stack:

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack
<encrypted_stack_name>
```

Infinispan nodes can join the cluster only if they use `ASYM_ENCRYPT` and can obtain the secret key from the coordinator node. Otherwise the following message is written to Infinispan logs:

```
[org.jgroups.protocols.ASYM_ENCRYPT] <hostname>: received message without encrypt
header from <hostname>; dropping it
```

Reference

The example `ASYM_ENCRYPT` configuration in this procedure shows commonly used parameters. Refer to JGroups documentation for the full set of available parameters.

- JGroups 4 Manual
- JGroups 4.2 Schema

5.7.3. Configuring Cluster Transport with Symmetric Encryption

Configure Infinispan clusters to encrypt JGroups messages with secret keys from keystores that you provide.

Procedure

1. Create a keystore that contains a secret key.
2. Place the keystore on the classpath for each node in the cluster.

   For Infinispan Server, you put the keystore in the $ISPN_HOME directory.
3. Add the `SYM_ENCRYPT` protocol to a JGroups stack in your Infinispan configuration.

```
<infinispan>
  <jgroups>
    <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the default
TCP stack. -->
    <stack name="encrypt-tcp" extends="tcp">
      <!-- Adds a keystore from which nodes obtain secret keys. -->
      <!-- Uses the stack.combine and stack.position attributes to insert SYM_ENCRYPT
into the default TCP stack after VERIFY_SUSPECT. -->
      <SYM_ENCRYPT keystore_name="myKeystore.p12"
                   keystore_type="PKCS12"
                   store_password="changeit"
                   key_password="changeit"
                   alias="myKey"
                   stack.combine="INSERT_AFTER"
                   stack.position="VERIFY_SUSPECT"/>
    </stack>
  </jgroups>
  <cache-container name="default" statistics="true">
    <!-- Configures the cluster to use the JGroups stack. -->
    <transport cluster="${infinispan.cluster.name}"
               stack="encrypt-tcp"
               node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

Verification

When you start your Infinispan cluster, the following log message indicates that the cluster is using the secure JGroups stack:

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack
<encrypted_stack_name>
```

Infinispan nodes can join the cluster only if they use `SYM_ENCRYPT` and can obtain the secret key from the shared keystore. Otherwise the following message is written to Infinispan logs:

```
[org.jgroups.protocols.SYM_ENCRYPT] <hostname>: received message without encrypt
header from <hostname>; dropping it
```

Reference

The example `SYM_ENCRYPT` configuration in this procedure shows commonly used parameters. Refer to JGroups documentation for the full set of available parameters.

- JGroups 4 Manual
- JGroups 4.2 Schema

Chapter 6. Remotely Creating Infinispan Caches

Add caches to Infinispan Server so you can store data.

6.1. Cache Configuration with Infinispan Server

Caches configure the data container on Infinispan Server.

You create caches at run-time by adding definitions based on `org.infinispan` templates or Infinispan configuration through the console, the Command Line Interface (CLI), the Hot Rod endpoint, or the REST endpoint.

When you create caches at run-time, Infinispan Server replicates your cache definitions across the cluster.

Configuration that you declare directly in `infinispan.xml` is not automatically synchronized across Infinispan clusters. In this case you should use configuration management tooling, such as Ansible or Chef, to ensure that configuration is propagated to all nodes in your cluster.

6.2. Default Cache Manager

{ProductName} Server provides a default Cache Manager configuration. When you start Infinispan Server, it instantiates the Cache Manager so you can remotely create caches at run-time.

Default Cache Manager

```
<!-- Creates a Cache Manager named "default" and exports metrics. -->
<cache-container name="default"
                 statistics="true">
  <!-- Adds cluster transport that uses the default JGroups TCP stack. -->
  <transport cluster="${infinispan.cluster.name:cluster}"
             stack="${infinispan.cluster.stack:tcp}"
             node-name="${infinispan.node.name:}"/>
</cache-container>
```

Examining the Cache Manager

After you start Infinispan Server and add user credentials, you can access the default Cache Manager through the Command Line Interface (CLI) or REST endpoint as follows:

- CLI: Use the `describe` command in the default container.

```
[//containers/default]> describe
```

- REST: Navigate to `<server_hostname>:11222/rest/v2/cache-managers/default/` in any browser.

6.3. Creating Caches with the Infinispan Console

Dynamically add caches from templates or configuration files through the Infinispan console.

Prerequisites

Create a user and start at least one Infinispan server instance.

Procedure

1. Navigate to `<server_hostname>:11222/console/` in any browser.
2. Log in to the console.
3. Open the **Data Container** view.
4. Select **Create Cache** and then add a cache from a template or with Infinispan configuration in XML or JSON format.
5. Return to the **Data Container** view and verify your Infinispan cache.

6.4. Creating Caches with the Infinispan Command Line Interface (CLI)

Use the Infinispan CLI to add caches from templates or configuration files in XML or JSON format.

Prerequisites

Create a user and start at least one Infinispan server instance.

Procedure

1. Create a CLI connection to Infinispan.
2. Add cache definitions with the `create cache` command.
   - Add a cache definition from an XML or JSON file with the `--file` option.

     ```
     [//containers/default]> create cache --file=configuration.xml mycache
     ```

   - Add a cache definition from a template with the `--template` option.

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

     Press the tab key after the `--template=` argument to list available cache templates.
3. Verify the cache exists with the `ls` command.

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

4. Retrieve the cache configuration with the `describe` command.

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

Reference

- Creating Infinispan CLI Connections
- Performing Cache Operations with the Infinispan CLI

6.5. Creating Caches with Hot Rod Clients

Programmatically create caches on Infinispan Server through the `RemoteCacheManager` API.

The following procedure demonstrates programmatic cache creation with the Hot Rod Java client. However Hot Rod clients are available in different languages such as Javascript or C++.

Prerequisites

- Create a user and start at least one Infinispan server instance.
- Get the Hot Rod Java client.

Procedure

1. Configure your client with the `ConfigurationBuilder` class.

```
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.DefaultTemplate;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.commons.configuration.XMLStringConfiguration;
...

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer()
         .host("127.0.0.1")
         .port(11222)
       .security().authentication()
          .enable()
          .username("username")
          .password("password")
          .realm("default")
          .saslMechanism("DIGEST-MD5");

manager = new RemoteCacheManager(builder.build());
```

2. Use the `XMLStringConfiguration` class to add cache definitions in XML format.
3. Call the `getOrCreateCache()` method to add the cache if it already exists or create it if not.

```
private void createCacheWithXMLConfiguration() {
    String cacheName = "CacheWithXMLConfiguration";
    String xml = String.format("<distributed-cache name=\"%s\" mode=\"SYNC\"
                                  statistics=\"true\">" +
                                 "<locking isolation=\"READ_COMMITTED\"/>" +
                                 "<transaction mode=\"NON_XA\"/>" +
                                 "<expiration lifespan=\"60000\"
interval=\"20000\"/>" +
                               "</distributed-cache>" +
                               , cacheName);
    manager.administration().getOrCreateCache(cacheName, new
XMLStringConfiguration(xml));
    System.out.println("Cache created or already exists.");
}
```

4. Create caches with `org.infinispan` templates as in the following example with the `createCache()` invocation:

```
private void createCacheWithTemplate() {
    manager.administration().createCache("myCache", "org.infinispan.DIST_SYNC");
    System.out.println("Cache created.");
}
```

Next Steps

Try some working code examples that show you how to create remote caches with the Hot Rod Java client. Visit the Infinispan Tutorials.

Reference

- `RemoteCacheManager` Javadoc
- Getting the Hot Rod Java Client

6.6. Creating Infinispan Caches with HTTP Clients

Add cache definitions to Infinispan servers through the REST endpoint with any suitable HTTP client.

Prerequisites

Create a user and start at least one Infinispan server instance.

Procedure

- Create caches with `POST` requests to `/rest/v2/caches/$cacheName`.

Use XML or JSON configuration by including it in the request payload.

```
POST /rest/v2/caches/mycache
```

Use the `?template=` parameter to create caches from `org.infinispan` templates.

```
POST /rest/v2/caches/mycache?template=org.infinispan.DIST_SYNC
```

Reference

- Creating and Managing Caches with the REST API

6.7. Cache Configuration

You can provide cache configuration in XML or JSON format.

XML

```
<infinispan>
    <cache-container>
        <distributed-cache name="myCache" mode="SYNC">
          <encoding media-type="application/x-protostream"/>
          <memory max-count="1000000" when-full="REMOVE"/>
        </distributed-cache>
    </cache-container>
</infinispan>
```

JSON

```
{
  "distributed-cache": {
    "name": "myCache",
    "mode": "SYNC",
    "encoding": {
      "media-type": "application/x-protostream"
      },
    "memory": {
      "max-count": 1000000,
      "when-full": "REMOVE"
    }
  }
}
```

JSON format

Cache configuration in JSON format must follow the structure of an XML configuration. * XML elements become JSON objects. * XML attributes become JSON fields.

Chapter 7. Configuring Infinispan Server Datasources

Create managed datasources to optimize connection pooling and performance for database connections.

You can specify database connection properties as part of a JDBC cache store configuration. However you must do this for each cache definition, which duplicates configuration and wastes resources by creating multiple distinct connection pools.

By using shared, managed datasources, you centralize connection configuration and pooling for more efficient usage.

7.1. Datasource Configuration for JDBC Cache Stores

Infinispan server configuration for datasources is composed of two sections:

- A `connection factory` that defines how to connect to the database.
- A `connection pool` that defines how to pool and reuse connections.

```
<data-sources>
   <!-- Defines a unique name for the datasource, JNDI name, and enables statisitics.
-->
   <data-source name="ds" jndi-name="jdbc/datasource" statistics="true">
      <!-- Specifies the JDBC driver that creates connections. -->
      <connection-factory driver="org.database.Driver"
                          username="db_user"
                          password="secret"
                          url="jdbc:db://database-host:10000/dbname"
                          new-connection-sql="SELECT 1"
                          transaction-isolation="READ_COMMITTED">
         <!-- Sets optional JDBC driver-specific connection properties. -->
         <connection-property name="name">value</connection-property>
      </connection-factory>
      <!-- Defines connection pool properties. -->
      <connection-pool initial-size="1"
                       max-size="10"
                       min-size="3"
                       background-validation="1000"
                       idle-removal="1"
                       blocking-timeout="1000"
                       leak-detection="10000"/>
   </data-source>
</data-sources>
```

7.2. Using Datasources in JDBC Cache Stores

Use a shared, managed datasource in your JDBC cache store configuration instead of specifying individual connection properties for each cache definition.

Prerequisites

Create a managed datasource for JDBC cache stores in your Infinispan server configuration.

Procedure

- Reference the JNDI name of the datasource in the JDBC cache store configuration of your cache configuration, as in the following example:

```
<distributed-cache-configuration name="persistent-cache" xmlns:jdbc=
"urn:infinispan:config:store:jdbc:12.1">
    <persistence>
        <jdbc:string-keyed-jdbc-store>
            <!-- Specifies the JNDI name that you provided for the datasource
connection in the server configuration. -->
            <jdbc:data-source jndi-url="jdbc/postgres"/>
            <jdbc:string-keyed-table drop-on-exit="true"
                        create-on-start="true"
                        prefix="TBL">
                <jdbc:id-column name="ID" type="VARCHAR(255)"/>
                <jdbc:data-column name="DATA" type="BYTEA"/>
                <jdbc:timestamp-column name="TS" type="BIGINT"/>
                <jdbc:segment-column name="S" type="INT"/>
            </jdbc:string-keyed-table>
        </jdbc:string-keyed-jdbc-store>
    </persistence>
</distributed-cache-configuration>
```

Chapter 8. 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.

8.1. Creating Server Tasks

Create custom task implementations and add them to Infinispan servers.

8.1.1. 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`

8.1.2. 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 allow list in your Infinispan configuration. Alternatively set the allow list using system properties.

Reference

- Adding Java Classes to Deserialization White Lists
- Infinispan 12.1 Configuration Schema

8.2. Creating Server Scripts

Create custom scripts and add them to Infinispan servers.

8.2.1. Server Scripts

Infinispan server scripting is based on the `javax.script` API and is compatible with any JVM-based ScriptEngine implementation.

Hello World Script Example

The following is 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.

8.2.2. Adding Scripts to Infinispan Servers

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

Prerequisites

Infinispan Server stores scripts in the `___script_cache` cache. If you enable cache authorization, users require 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. Create a CLI connection to Infinispan.
3. Use the `task` command to upload scripts, as in the following example:

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

4. Verify that your scripts are available.

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

8.2.3. 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

8.3. Running Server-Side Tasks and Scripts

Execute tasks and custom scripts on Infinispan servers.

8.3.1. Running Tasks and Scripts

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

Procedure

1. Create a CLI connection to Infinispan.
2. Use the `task` command to run tasks and scripts, 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"]
     ```

8.3.2. 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

8.3.3. 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 9. Enabling and Customizing Logging

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.

9.1. Server Logs

Infinispan writes server logs to the following files in the `$ISPN_HOME/server/log` directory:

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

`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.

9.1.1. Configuring Server Logs

Infinispan uses Apache Log4j technology to write server log messages. You can configure server logs in the `log4j2.xml` file.

Procedure

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

Additional resources

- Apache Log4j manual

9.1.2. 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.

Log level	Description
WARN	Events that can lead to error or degrade performance.
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.

9.1.3. 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.

9.1.4. 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.

9.1.5. 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 .

9.1.6. Enabling and Configuring the JSON Log Handler

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

Prerequisites

- Stop Infinispan Server if it is running.
  You cannot dynamically enable log handlers.

Procedure

1. Open `$ISPN_HOME/server/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 JSON layout as required.
4. Save and close `log4j2.xml`.

When you start Infinispan, it writes each log message as a JSON map in the following file:

`$ISPN_HOME/server/log/server.log.json`

Additional resources

- RollingFileAppender
- JSONLayout

9.2. Access Logs

Access logs record all inbound client requests for Hot Rod and REST endpoints to files in the `$ISPN_HOME/server/log` directory.

`org.infinispan.HOTROD_ACCESS_LOG`
: Logging category that writes Hot Rod access messages to a `hotrod-access.log` file.

`org.infinispan.REST_ACCESS_LOG`
: Logging category that writes REST access messages to a `rest-access.log` file.

9.2.1. Enabling Access Logs

To record Hot Rod and REST endpoint access messages, you need to enable the logging categories in `log4j2.xml`.

Procedure

1. Open `$ISPN_HOME/server/conf/log4j2.xml` with any text editor.
2. Change the level for the `org.infinispan.HOTROD_ACCESS_LOG` and `org.infinispan.REST_ACCESS_LOG` logging categories to `TRACE`.
3. Save and close `log4j2.xml`.

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

9.2.2. Access Log Properties

The default format for access logs is as follows:

```
%X{address} %X{user} [%d{dd/MMM/yyyy:HH:mm:ss Z}] &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}.

9.3. Audit Logs

Audit logs let you track changes to your Infinispan environment so you know when changes occur and which users make them. Enable and configure audit logging to record server configuration events and administrative operations.

org.infinispan.AUDIT
: Logging category that writes security audit messages to an audit.log file in the $ISPN_HOME/server/log directory.

9.3.1. Enabling Audit Logging

To record security audit messages, you need to enable the logging category in log4j2.xml.

Procedure

1. Open $ISPN_HOME/server/conf/log4j2.xml with any text editor.
2. Change the level for the org.infinispan.AUDIT logging category to INFO.
3. Save and close log4j2.xml.

```
<!-- Set to INFO to enable audit logging -->
<Logger name="org.infinispan.AUDIT" additivity="false" level="INFO">
   <AppenderRef ref="AUDIT-FILE"/>
</Logger>
```

9.3.2. Configuring Audit Logging Appenders

Apache Log4j provides different appenders that you can use to send audit messages to a destination other than the default log file. For instance, if you want to send audit logs to a syslog daemon, JDBC database, or Apache Kafka server, you can configure an appender in `log4j2.xml`.

Procedure

1. Open `$ISPN_HOME/server/conf/log4j2.xml` with any text editor.
2. Comment or remove the default `AUDIT-FILE` rolling file appender.

```
<!--RollingFile name="AUDIT-FILE"
  ...
</RollingFile-->
```

3. Add the desired logging appender for audit messages.

   For example, you could add a logging appender for a Kafka server as follows:

```
<Kafka name="AUDIT-KAFKA" topic="audit">
  <PatternLayout pattern="%date %message"/>
  <Property name="bootstrap.servers">localhost:9092</Property>
</Kafka>
```

4. Save and close `log4j2.xml`.

Additional resources

- Log4j Appenders

9.3.3. Using Custom Audit Logging Implementations

You can create custom implementations of the `org.infinispan.security.AuditLogger` API if configuring Log4j appenders does not meet your needs.

Prerequisites

- Implement `org.infinispan.security.AuditLogger` as required and package it in a JAR file.

Procedure

1. Add your JAR to the `server/lib` directory in your Infinispan Server installation.
2. Specify the fully qualified class name of your custom audit logger as the value for the `audit-logger` attribute on the `authorization` element in your cache container security configuration.

   For example, the following configuration defines `my.package.CustomAuditLogger` as the class for logging audit messages:

```
<infinispan>
   <cache-container>
      <security>
         <authorization audit-logger="my.package.CustomAuditLogger"/>
      </security>
   </cache-container>
</infinispan>
```

Additional resources

- `org.infinispan.security.AuditLogger`

Chapter 10. Monitoring Infinispan Servers

10.1. 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.

10.1.1. Enabling Infinispan Statistics

Infinispan lets you enable statistics for Cache Managers and caches. However, enabling statistics for a Cache Manager does not enable statistics for the caches that it controls. You must explicitly enable statistics for your caches.

Infinispan server enables statistics for Cache Managers by default.

Procedure

- Enable statistics declaratively or programmatically.

Declaratively

```
<!-- Enables statistics for the Cache Manager. -->
<cache-container statistics="true">
  <!-- Enables statistics for the named cache. -->
  <local-cache name="mycache" statistics="true"/>
</cache-container>
```

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  //Enables statistics for the Cache Manager.
  .cacheContainer().statistics(true)
  .build();

Configuration config = new ConfigurationBuilder()
  //Enables statistics for the named cache.
  .statistics().enable()
  .build();
```

10.1.2. Enabling Infinispan Metrics

Configure Infinispan to export gauges and histograms.

Procedure

- Configure metrics declaratively or programmatically.

Declaratively

```
<!-- Computes and collects statistics for the Cache Manager. -->
<cache-container statistics="true">
  <!-- Exports collected statistics as gauge and histogram metrics. -->
  <metrics gauges="true" histograms="true" />
</cache-container>
```

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  //Computes and collects statistics for the Cache Manager.
  .statistics().enable()
  //Exports collected statistics as gauge and histogram metrics.
  .metrics().gauges(true).histograms(true)
  .build();
```

10.1.3. 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

10.1.4. Configuring Infinispan to Register JMX MBeans

Infinispan can register JMX MBeans that you can use to collect statistics and perform administrative operations. However, you must enable statistics separately to JMX otherwise Infinispan provides `0` values for all statistic attributes.

Procedure

- Enable JMX declaratively or programmatically to register Infinispan JMX MBeans.

Declaratively

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

Programmatically

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

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

10.2. 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

10.2.1. 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.

10.2.2. 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.

Chapter 11. Performing Rolling Upgrades for Infinispan Servers

Perform rolling upgrades of your Infinispan clusters to change between versions without downtime or data loss. Rolling upgrades migrate both your Infinispan servers and your data to the target version over Hot Rod.

11.1. Setting Up Target Clusters

Create a cluster that runs the target Infinispan version and uses a remote cache store to load data from the source cluster.

Prerequisites

- Install a Infinispan cluster with the target upgrade version.

Ensure the network properties for the target cluster do not overlap with those for the source cluster. You should specify unique names for the target and source clusters in the JGroups transport configuration. Depending on your environment you can also use different network interfaces and specify port offsets to keep the target and source clusters separate.

Procedure

1. Add a `RemoteCacheStore` on the target cluster for each cache you want to migrate from the source cluster.

   Remote cache stores use the Hot Rod protocol to retrieve data from remote Infinispan clusters. When you add the remote cache store to the target cluster, it can lazily load data from the source cluster to handle client requests.

2. Switch clients over to the target cluster so it starts handling all requests.
   a. Update client configuration with the location of the target cluster.
   b. Restart clients.

11.1.1. Remote Cache Stores for Rolling Upgrades

You must use specific remote cache store configuration to perform rolling upgrades, as follows:

```
<!-- Remote cache stores for rolling upgrades must disable passivation. -->
<persistence passivation="false">
   <!-- The value of the cache attribute matches the name of a cache in the source
cluster. Target clusters load data from this cache using the remote cache store. -->
   <!-- The "protocol-version" attribute matches the Hot Rod protocol version of the
source cluster. 2.5 is the minimum version and is suitable for any upgrade path. -->
   <!-- You should enable segmentation for remote cache stores only if the number of
segments in the target cluster matches the number of segments for the cache in the
source cluster. -->
   <remote-store xmlns="urn:infinispan:config:store:remote:12.1"
                 cache="myDistCache"
                 protocol-version="2.5"
                 hotrod-wrapping="true"
                 raw-values="true"
                 segmented="false">
      <!-- Points to the location of the source cluster. -->
      <remote-server host="127.0.0.1" port="11222"/>
   </remote-store>
</persistence>
```

Reference

- Remote cache store configuration schema
- RemoteStore
- RemoteStoreConfigurationBuilder

11.2. Synchronizing Data to Target Clusters

When your target cluster is running and handling client requests using a remote cache store to load data on demand, you can synchronize data from the source cluster to the target cluster.

This operation reads data from the source cluster and writes it to the target cluster. Data migrates to all nodes in the target cluster in parallel, with each node receiving a subset of the data. You must perform the synchronization for each cache in your Infinispan configuration.

Procedure

1. Start the synchronization operation for each cache in your Infinispan configuration that you want to migrate to the target cluster.

   Use the Infinispan REST API and invoke `POST` requests with the `?action=sync- data` parameter. For example, to synchronize data in a cache named "myCache" from a source cluster to a target cluster, do the following:

   ```
   POST /v2/caches/myCache?action=sync-data
   ```

   When the operation completes, Infinispan responds with the total number of entries copied to the target cluster.

Alternatively, you can use JMX by invoking `synchronizeData(migratorName=hotrod)` on the `RollingUpgradeManager` MBean.

2. Disconnect each node in the target cluster from the source cluster.

   For example, to disconnect the "myCache" cache from the source cluster, invoke the following `POST` request:

   ```
   POST /v2/caches/myCache?action=disconnect-source
   ```

   To use JMX, invoke `disconnectSource(migratorName=hotrod)` on the `RollingUpgradeManager` MBean.

Next steps

After you synchronize all data from the source cluster, the rolling upgrade process is complete. You can now decommission the source cluster.

Chapter 12. Patching Infinispan Server Installations

Install and manage patches for Infinispan server installations.

You can apply patches to multiple Infinispan servers with different versions to upgrade to a desired target version. However, patches do not take effect if Infinispan servers are running. For this reason you install patches while servers are offline. If you want to upgrade Infinispan clusters without downtime, create a new cluster with the target version and perform a rolling upgrade to that version instead of patching.

12.1. Infinispan Server Patches

Infinispan server patches are `.zip` archives that contain artifacts that you can apply to your `$ISPN_HOME` directory to fix issues and add new features.

Patches also provide a set of rules for Infinispan to modify your server installation. When you apply patches, Infinispan overwrites some files and removes others, depending on if they are required for the target version.

However, Infinispan does not make any changes to configuration files that you have created or modified when applying a patch. Server patches do not modify or replace any custom configuration or data.

12.2. Creating Server Patches

You can create patches for Infinispan servers from an existing server installation.

You can create patches for Infinispan servers starting from 10.1.7. You can patch any 10.1 or later server installation. However you cannot patch 9.4.x or earlier servers with 10.1.7 or later.

You can also create patches that either upgrade or downgrade the Infinispan server version. For example, you can create a patch from version 10.1.7 and use it to upgrade version 10.1.5 or downgrade version 11.0.0.

Procedure

1. Navigate to `$ISPN_HOME` for a Infinispan server installation that has the target version for the patch you want to create.
2. Start the CLI.

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

3. Use the `patch create` command to generate a patch archive and include the `-q` option with a meaningful qualifier to describe the patch.

```
[disconnected]> patch create -q "this is my test patch" path/to/mypatch.zip \
path/to/target/server/home path/to/source/server/home
```

The preceding command generates a `.zip` archive in the specified directory. Paths are relative to `$ISPN_HOME` for the target server.

Create single patches for multiple different Infinispan versions, for example:

```
[disconnected]> patch create -q "this is my test patch"
path/to/mypatch.zip \
path/to/target/server/home \
path/to/source/server1/home path/to/source/server2/home
```

Where `server1` and `server2` are different Infinispan versions where you can install "mypatch.zip".

4. Describe the generated patch archive.

```
[disconnected]> patch describe path/to/mypatch.zip

Infinispan patch target=$target_version(my test patch)  source=$source_version
created=$timestamp
```

   - `$target_version` is the Infinispan server version from which the patch was created.
   - `$source_version` is one or more Infinispan server versions to which you can apply the patch.

     You can apply patches to Infinispan servers that match the `$source_version` only. Attempting to apply patches to other versions results in the following exception:

     ```
     java.lang.IllegalStateException: The supplied patch cannot be applied to
     `$source_version`
     ```

12.3. Installing Server Patches

Apply patches to Infinispan servers to upgrade or downgrade an existing version.

Prerequisites

- Create a server patch for the target version.

Procedure

1. Navigate to `$ISPN_HOME` for the Infinispan server you want to patch.
2. Stop the server if it is running.

If you patch a server while it is running, the version changes take effect after restart. If you do not want to stop the server, create a new cluster with the target version and perform a rolling upgrade to that version instead of patching.

3. Start the CLI.

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

4. Install the patch.

```
[disconnected]> patch install path/to/patch.zip

Infinispan patch target=$target_version source=$source_version \
created=$timestamp installed=$timestamp
```

   - `$target_version` displays the Infinispan version that the patch installed.
   - `$source_version` displays the Infinispan version before you installed the patch.

5. Start the server to verify the patch is installed.

```
$ bin/server.sh
...
ISPN080001: Infinispan Server $version
```

   If the patch is installed successfully `$version` matches `$target_version`.

Use the `--server` option to install patches in a different `$ISPN_HOME` directory, for example:

```
[disconnected]> patch install path/to/patch.zip
--server=path/to/server/home
```

12.4. Rolling Back Server Patches

Remove patches from Infinispan servers by rolling them back and restoring the previous Infinispan version.

If a server has multiple patches installed, you can roll back the last installed patch only.

Rolling back patches does not revert configuration changes you make to Infinispan server. Before you roll back patches, you should ensure that your configuration is compatible with the version to which you are rolling back.

Procedure

1. Navigate to `$ISPN_HOME` for the Infinispan server installation you want to roll back.
2. Stop the server if it is running.
3. Start the CLI.

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

4. List the installed patches.

```
[disconnected]> patch ls

Infinispan patch target=$target_version source=$source_version
created=$timestamp installed=$timestamp
```

   - `$target_version` is the Infinispan server version after the patch was applied.
   - `$source_version` is the version for Infinispan server before the patch was applied. Rolling back the patch restores the server to this version.

5. Roll back the last installed patch.

```
[disconnected]> patch rollback
```

6. Quit the CLI.

```
[disconnected]> quit
```

7. Start the server to verify the patch is rolled back to the previous version.

```
$ bin/server.sh
...
ISPN080001: Infinispan Server $version
```

   If the patch is rolled back successfully `$version` matches `$source_version`.

Use the `--server` option to rollback patches in a different `$ISPN_HOME` directory, for example:

```
[disconnected]> patch rollback --server=path/to/server/home
```

Chapter 13. Troubleshooting Infinispan Servers

Gather diagnostic information about Infinispan server deployments and perform troubleshooting steps to resolve issues.

13.1. Getting Diagnostic Reports for Infinispan Servers

Infinispan servers provide aggregated reports in `tar.gz` archives that contain diagnostic information about both the Infinispan server and the host. The report provides details about CPU, memory, open files, network sockets and routing, threads, in addition to configuration and log files.

Procedure

1. Create a CLI connection to Infinispan.
2. Use the `server report` command to download a `tar.gz` archive:

```
[//containers/default]> server report
Downloaded report 'infinispan-<hostname>-<timestamp>-report.tar.gz'
```

3. Move the `tar.gz` file to a suitable location on your filesystem.
4. Extract the `tar.gz` file with any archiving tool.

13.2. Changing Infinispan Server Logging Configuration at Runtime

Modify the logging configuration for Infinispan servers at runtime to temporarily adjust logging to troubleshoot issues and perform root cause analysis.

Modifying the logging configuration through the CLI is a runtime-only operation, which means that changes:

- Are not saved to the `log4j2.xml` file. Restarting server nodes or the entire cluster resets the logging configuration to the default properties in the `log4j2.xml` file.
- Apply only to the nodes in the cluster when you invoke the CLI. Nodes that join the cluster after you change the logging configuration use the default properties.

Procedure

1. Create a CLI connection to Infinispan.
2. Use the `logging` to make the required adjustments.
    - List all appenders defined on the server:

```
[//containers/default]> logging list-appenders
```

The preceding command returns:

```
{
  "STDOUT" : {
    "name" : "STDOUT"
  },
  "JSON-FILE" : {
    "name" : "JSON-FILE"
  },
  "HR-ACCESS-FILE" : {
    "name" : "HR-ACCESS-FILE"
  },
  "FILE" : {
    "name" : "FILE"
  },
  "REST-ACCESS-FILE" : {
    "name" : "REST-ACCESS-FILE"
  }
}
```

- List all logger configurations defined on the server:

```
[//containers/default]> logging list-loggers
```

The preceding command returns:

```
[ {
  "name" : "",
  "level" : "INFO",
  "appenders" : [ "STDOUT", "FILE" ]
}, {
  "name" : "org.infinispan.HOTROD_ACCESS_LOG",
  "level" : "INFO",
  "appenders" : [ "HR-ACCESS-FILE" ]
}, {
  "name" : "com.arjuna",
  "level" : "WARN",
  "appenders" : [ ]
}, {
  "name" : "org.infinispan.REST_ACCESS_LOG",
  "level" : "INFO",
  "appenders" : [ "REST-ACCESS-FILE" ]
} ]
```

- Add and modify logger configurations with the `set` subcommand

For example, the following command sets the logging level for the `org.infinispan` package to `DEBUG`:

```
[//containers/default]> logging set --level=DEBUG org.infinispan
```

- Remove existing logger configurations with the `remove` subcommand.

For example, the following command removes the `org.infinispan` logger configuration, which means the root configuration is used instead:

```
[//containers/default]> logging remove org.infinispan
```

13.3. Resource Statistics

You can inspect server-collected statistics for some of the resources within a Infinispan server using the `stats` command.

Use the `stats` command either from the context of a resource which collects statistics (containers, caches) or with a path to such a resource:

```
[//containers/default]> stats
{
  "statistics_enabled" : true,
  "number_of_entries" : 0,
  "hit_ratio" : 0.0,
  "read_write_ratio" : 0.0,
  "time_since_start" : 0,
  "time_since_reset" : 49,
  "current_number_of_entries" : 0,
  "current_number_of_entries_in_memory" : 0,
  "total_number_of_entries" : 0,
  "off_heap_memory_used" : 0,
  "data_memory_used" : 0,
  "stores" : 0,
  "retrievals" : 0,
  "hits" : 0,
  "misses" : 0,
  "remove_hits" : 0,
  "remove_misses" : 0,
  "evictions" : 0,
  "average_read_time" : 0,
  "average_read_time_nanos" : 0,
  "average_write_time" : 0,
  "average_write_time_nanos" : 0,
  "average_remove_time" : 0,
  "average_remove_time_nanos" : 0,
  "required_minimum_number_of_nodes" : -1
}
```

```
[//containers/default]> stats /containers/default/caches/mycache
{
  "time_since_start" : -1,
  "time_since_reset" : -1,
  "current_number_of_entries" : -1,
  "current_number_of_entries_in_memory" : -1,
  "total_number_of_entries" : -1,
  "off_heap_memory_used" : -1,
  "data_memory_used" : -1,
  "stores" : -1,
  "retrievals" : -1,
  "hits" : -1,
  "misses" : -1,
  "remove_hits" : -1,
  "remove_misses" : -1,
  "evictions" : -1,
  "average_read_time" : -1,
  "average_read_time_nanos" : -1,
  "average_write_time" : -1,
  "average_write_time_nanos" : -1,
  "average_remove_time" : -1,
  "average_remove_time_nanos" : -1,
  "required_minimum_number_of_nodes" : -1
}
```