Deploying and Configuring Infinispan 11.0 Servers

Table of Contents

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

Chapter 1. Getting Started with Infinispan Server

Quickly set up Infinispan server and learn the basics.

1.1. Infinispan Server Requirements

Check host system requirements for the Infinispan server.

Infinispan server requires a Java Virtual Machine and supports:

- Java 8
- Java 11

1.2. Downloading Server Distributions

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

Procedure

Download the Infinispan 11.0 server from Infinispan downloads.

Verification

Use the checksum to verify the integrity of your download.

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

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

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

Reference

- Infinispan Server README describes the contents of the server distribution.

1.3. Installing Infinispan Server

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

Procedure

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

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

The resulting directory is your `$ISPN_HOME`.

1.4. Running Infinispan Servers

Quickly create a locally running Infinispan cluster with two server instances. Infinispan server nodes on the same network automatically discover each other and form clusters.

1.4.1. Adding Infinispan Credentials

Infinispan Server provides a default property realm that restricts access to authenticated users only. Use the Infinispan CLI to add credentials.

Procedure

1. Open a terminal in `$ISPN_HOME`.
2. Define credentials with the `user` command as in the following examples:
   - Create a new user named "myuser" and specify a password:

     Linux

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

     Microsoft Windows

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

   - Create a new user that belongs to the "supervisor", "reader", and "writer" groups if you use security authorization:

     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
     ```

1.4.2. Starting Infinispan Servers

Launch Infinispan server with the startup script.

Prerequisites

- Add Infinispan credentials.

Procedure

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

Linux

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

Microsoft Windows

```
bin\server.bat
```

Server logs display 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
```

Hello Infinispan!

- Open `127.0.0.1:11222` in any browser, enter your credentials, and then access the Infinispan console.

1.4.3. Verifying Infinispan Cluster Discovery

Infinispan servers running on the same network discover each other with the `MPING` protocol and automatically form clusters. This procedure demonstrates that capability with two locally running Infinispan server instances.

Prerequisites

Start a locally running Infinispan server instance.

Procedure

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

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

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

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

Verification

Check server logs for the following messages:

```
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
```

1.5. Shutting Down Infinispan Servers

Gracefully shut down running Infinispan servers to passivate all entries to disk and persist state.

Procedure

1. Create a CLI connection to Infinispan.
2. Do one of the following:
   - Stop individual servers with the `shutdown server` command:

     ```
     [//containers/default]> shutdown server $hostname
     ```

   - Stop all nodes in the cluster with the `shutdown cluster` command:

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

Verification

Check the server logs for the following messages:

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

Chapter 2. Remotely Creating Infinispan Caches

Add caches to Infinispan Server so you can store data.

2.1. 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.

2.2. 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

```
<cache-container name="default" ①
                 statistics="true"> ②
  <transport cluster="${infinispan.cluster.name}" ③
             stack="${infinispan.cluster.stack:tcp}" ④
             node-name="${infinispan.node.name:}"/>
</cache-container>
```

① Creates a Cache Manager named "default".

② Exports Cache Manager statistics through the `metrics` endpoint.

③ Adds a JGroups cluster transport that allows {ProductName} servers to automatically discover each other and form clusters.

④ Uses the default TCP stack for cluster traffic.

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.

2.3. Creating Caches with the Infinispan Console

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

Prerequisites

Add Infinispan credentials 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.

2.4. 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

Add Infinispan credentials 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

2.5. 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

- Add Infinispan credentials 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("<infinispan>" +
                                  "<cache-container>" +
                                  "<distributed-cache name=\"%s\" mode=\"SYNC\"
                                  statistics=\"true\">" +
                                    "<locking isolation=\"READ_COMMITTED\"/>" +
                                    "<transaction mode=\"NON_XA\"/>" +
                                    "<expiration lifespan=\"60000\"
interval=\"20000\"/>" +
                                  "</distributed-cache>" +
                                  "</cache-container>" +
                                "</infinispan>"
                                , 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

2.6. Creating Infinispan Caches with HTTP Clients

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

Prerequisites

Add Infinispan credentials 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

2.7. Infinispan Configuration

Infinispan configuration in XML and JSON format.

2.7.1. XML Configuration

Infinispan configuration in XML format must conform to the schema and include:

- `<infinispan>` root element.

- `<cache-container>` definition.

Example XML Configuration

```
<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>
```

2.7.2. JSON Configuration

Infinispan configuration in JSON format:

- Requires the cache definition only.
- Must follow the structure of an XML configuration.
  - XML elements become JSON objects.
  - XML attributes become JSON fields.

Example JSON Configuration

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

Chapter 3. 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.

3.1. Server Interfaces

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

3.1.1. 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.

3.1.2. 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>
```

3.1.3. Non-Loopback Strategy

Selects a non-loopback address.

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

3.1.4. Network Address Strategy

Selects networks based on IP address.

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

3.1.5. 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>
```

3.1.6. 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>
```

3.1.7. 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>
```

3.1.8. 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>
```

3.1.9. 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>
```

3.1.10. 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>
```

3.1.11. 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>
```

3.1.12. 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
```

3.2. 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>
```

3.2.1. 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
```

3.3. 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.

3.3.1. 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 4. Configuring Infinispan Server Endpoints

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

4.1. 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.

4.1.1. 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

4.1.2. 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

4.1.3. 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

4.1.4. 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**

4.2. Endpoint Connectors

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

The default endpoint connector configuration is as follows:

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

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

Reference

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

4.2.1. Hot Rod Connectors

Hot Rod connector declarations enable Hot Rod servers.

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

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

Reference

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

4.2.2. REST Connectors

REST connector declarations enable REST servers.

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

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

Reference

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

4.2.3. Memcached Connectors

Memcached connector declarations enable Memcached servers.

Infinispan servers do not enable Memcached connectors by default.

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

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

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

Reference

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

Chapter 5. Monitoring Infinispan Servers

5.1. Working with Infinispan Server Logs

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

5.1.1. Infinispan Log Files

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

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

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

5.1.2. Configuring Infinispan Log Properties

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

Procedure

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

Log Levels

Log levels indicate the nature and severity of messages.

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

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

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

Infinispan Log Categories

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

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

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

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

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

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

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

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

Log Appenders

Log appenders define how Infinispan records log messages.

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

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

JSON-FILE

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

Log Patterns

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

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

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

Patterns are fully described in the `PatternLayout` documentation .

Enabling and Configuring the JSON Log Handler

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

Prerequisites

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

Procedure

1. Open `$ISPN_HOME/${infinispan.server.root}/conf/log4j2.xml` with any text editor.
2. Uncomment the `JSON-FILE` appender and comment out the `FILE` appender:

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

3. Optionally configure the JSON appender and layout.
4. Save and close `logging.properties`.

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

5.1.3. Access Logs

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

categories:

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

Enabling Access Logs

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

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

Access Log Properties

The default format for access logs is as follows:

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

The preceding format creates log entries such as the following:

```
127.0.0.1 - [DD/MM/YYYY:HH:MM:SS +0000] "PUT /rest/v2/caches/default/key HTTP/1.1" 404 5 77 10
```

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

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

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

5.2. Configuring Statistics, Metrics, and JMX

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

5.2.1. 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

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

① Enables statistics for the Cache Manager.

② Enables statistics for the named cache.

Programmatically

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

 ...

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

① Enables statistics for the Cache Manager.

② Enables statistics for the named cache.

5.2.2. Enabling Infinispan Metrics

Configure Infinispan to export gauges and histograms.

Procedure

- Configure metrics declaratively or programmatically.

Declaratively

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

① Computes and collects statistics about the Cache Manager.

② Exports collected statistics as gauge and histogram metrics.

Programmatically

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

① Computes and collects statistics about the Cache Manager.

② Exports collected statistics as gauge and histogram metrics.

5.2.3. Collecting Infinispan Metrics

Collect Infinispan metrics with monitoring tools such as Prometheus.

Prerequisites

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

Procedure

- Get metrics in Prometheus (OpenMetrics) format:

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

- Get metrics in MicroProfile JSON format:

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

Next steps

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

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

Reference

- Prometheus Configuration
- Enabling Infinispan Statistics

5.2.4. 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.

Declaratively

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

① Registers Infinispan JMX MBeans.

Programmatically

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

① Registers Infinispan JMX MBeans.

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

5.3. Retrieving Server Health Statistics

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

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

5.3.1. Accessing the Health API via JMX

Retrieve Infinispan cluster health statistics via JMX.

Procedure

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

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

2. Select available MBeans to retrieve cluster health statistics.

5.3.2. Accessing the Health API via REST

Get Infinispan cluster health via the REST API.

Procedure

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

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

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

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

}
```


Get cache manager status as follows:

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

Reference

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

Chapter 6. Securing Access to Infinispan Servers

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

6.1. Defining Infinispan Server Security Realms

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

6.1.1. 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.

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

Property realm configuration

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

① Defines groups as roles for Infinispan server authorization.

② Specifies the `users.properties` file.

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

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

⑤ Specifies the `groups.properties` file.

Supported authentication mechanisms

Property realms support the following authentication mechanisms:

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

Adding Infinispan Credentials

Infinispan Server provides a default property realm that restricts access to authenticated users only. Use the Infinispan CLI to add credentials.

Procedure

1. Open a terminal in `$ISPN_HOME`.
2. Define credentials with the `user` command as in the following examples:
   - Create a new user named "myuser" and specify a password:

     Linux

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

     Microsoft Windows

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

   - Create a new user that belongs to the "supervisor", "reader", and "writer" groups if you use security authorization:

     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
     ```

6.1.2. 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. For this reason, LDAP realm configuration is complex. It is beyond the scope of this document to provide examples for all possibile configurations.

LDAP realm configuration

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

① Names the LDAP realm.

② Specifies the LDAP server connection URL.

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

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

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

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

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

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

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

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

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

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

Supported authentication mechanisms

LDAP realms support the following authentication mechanisms directly:

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

LDAP Realm Principal Rewriting

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

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

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

6.1.3. Trust Store Realms

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

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

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

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

③ Specifies a keystore password.

④ Specifies a keystore alias.

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

Supported authentication mechanisms

Trust store realms work with client-certificate authentication mechanisms:

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

6.1.4. 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:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
         <token-realm name="token"
                      auth-server-url="https://oauth-server/auth/"> ①
            <oauth2-introspection
                     introspection-url="https://oauth-
server/auth/realms/infinispan/protocol/openid-connect/token/introspect" ②
                     client-id="infinispan-server" ③
                     client-secret="1fdca4ec-c416-47e0-867a-3d471af7050f"/> ④
         </token-realm>
      </security-realm>
   </security-realms>
</security>
```

① Specifies the URL of the authentication server.

② Specifies the URL of the token introspection endpoint.

③ Names the client identifier for Infinispan server.

④ Specifies the client secret for Infinispan server.

Supported authentication mechanisms

Token realms support the following authentication mechanisms:

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

6.2. Creating Infinispan Server Identities

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

6.2.1. Setting Up SSL Identities

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

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

Procedure

1. Create a keystore for Infinispan server.

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

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

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

SSL Identity Configuration

The following example configures an SSL identity for Infinispan server:

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

① Defines identities for Infinispan server.

② Configures an SSL identity for Infinispan server.

③ Names a keystore that contains Infinispan server SSL certificates.

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

⑤ Specifies a keystore password.

⑥ Specifies a keystore alias.

Automatically Generating Keystores

Configure Infinispan servers to automatically generate keystores at startup.

Automatically generated keystores:

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

Procedure

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

SSL server identity with a generated keystore

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

① generates a keystore using `localhost`

Tuning SSL Protocols and Cipher Suites

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

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

Procedure

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

SSL engine configuration

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

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

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

6.2.2. 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:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
          xmlns="urn:infinispan:server:11.0">
   <security-realms>
      <security-realm name="default">
         <server-identities> ①
            <kerberos keytab-path="hotrod.keytab" ②
                      principal="hotrod/datagrid@INFINISPAN.ORG" ③
                      required="true"/> ④
            <kerberos keytab-path="http.keytab" ⑤
                      principal="HTTP/localhost@INFINISPAN.ORG" ⑥
                      required="true"/>
         </server-identities>
      </security-realm>
   </security-realms>
</security>
```

① Defines identities for Infinispan server.

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

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

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

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

⑥ Names the Kerberos service principal for the REST connector.

6.3. 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.

6.3.1. 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
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.

6.3.2. 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:11.0
           https://infinispan.org/schemas/infinispan-server-11.0.xsd"
           xmlns="urn:infinispan:server:11.0"
           socket-binding="default" security-realm="default"> ①
   <hotrod-connector name="hotrod">
      <authentication>
         <sasl mechanisms="SCRAM-SHA-512 SCRAM-SHA-384 SCRAM-SHA-256 ②
                           SCRAM-SHA-1 DIGEST-SHA-512 DIGEST-SHA-384
                           DIGEST-SHA-256 DIGEST-SHA DIGEST-MD5 PLAIN"
               server-name="infinispan" ③
               qop="auth"/> ④
      </authentication>
   </hotrod-connector>
</endpoints>
```

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

② Specifies SASL mechanisms to use for authentication.

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

④ Enables **auth** QoP.

Hot Rod connector with Kerberos authentication

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

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

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

③ Specifies the Kerberos identity for the server.

Hot Rod Endpoint Authentication Mechanisms

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

Authentication mechanism	Description	Related details
PLAIN	Uses credentials in plain-text format. You should use PLAIN authentication with encrypted connections only.	Similar to the Basic HTTP mechanism.
DIGEST-*	Uses hashing algorithms and nonce values. Hot Rod connectors support DIGEST-MD5, DIGEST-SHA, DIGEST-SHA-256, DIGEST-SHA-384, and DIGEST-SHA-512 hashing algorithms, in order of strength.	Similar to the Digest HTTP mechanism.
SCRAM-*	Uses *salt* values in addition to hashing algorithms and nonce values. Hot Rod connectors support SCRAM-SHA, SCRAM-SHA-256, SCRAM-SHA-384, and SCRAM-SHA-512 hashing algorithms, in order of strength.	Similar to the Digest HTTP mechanism.
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
`pass-credentials`	Use only SASL mechanisms that require client credentials.	false
`no-plain-text`	Do not use SASL mechanisms that are susceptible to simple plain passive attacks.	false
`no-active`	Do not use SASL mechanisms that are susceptible to active, non-dictionary, attacks.	false
`no-dictionary`	Do not use SASL mechanisms that are susceptible to passive dictionary attacks.	false
`no-anonymous`	Do not use SASL mechanisms that accept anonymous logins.	true

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

Hot Rod connector with SASL policy configuration

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

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

② Defines policies for SASL mechanisms.

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

6.3.3. 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:11.0
https://infinispan.org/schemas/infinispan-server-11.0.xsd"
           xmlns="urn:infinispan:server:11.0"
           socket-binding="default" security-realm="default"> ①
   <rest-connector name="rest">
      <authentication mechanisms="DIGEST BASIC"/> ②
   </rest-connector>
</endpoints>
```

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

② Specifies SASL mechanisms to use for authentication

REST connector with Kerberos authentication

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

① Enables the `SPENGO` mechanism for Kerberos authentication.

② Specifies the Kerberos identity for the server.

REST Endpoint Authentication Mechanisms

Infinispan supports the following authentications mechanisms with the REST connector:

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

6.4. 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>
```

6.5. Configuring Infinispan Authorization

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

6.5.1. Infinispan Authorization

Infinispan lets you configure authorization to secure Cache Managers and cache instances. When user applications or clients attempt to perform an operation on secured Cached Managers and caches, they must provide an identity with a role that has sufficient permissions to perform that operation.

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.

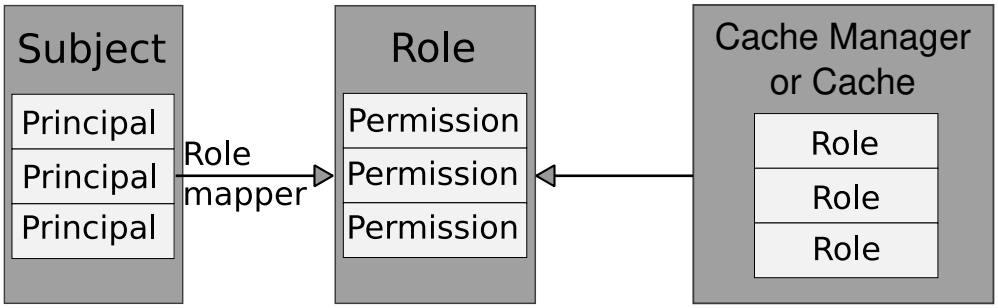
In this scenario, if a user application or client with the `reader` role attempts to write an entry, Infinispan denies the request and throws a security exception. If a user application or client with the `writer` role sends a write request, Infinispan validates authorization and issues a token for subsequent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

Identity to Role Mapping

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.

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

The following image illustrates how security principals map to roles:



Permissions

Permissions control access to Cache Managers and caches by restricting the actions that you can perform. Permissions can also apply to specific entities such as named caches.

Table 1. 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.
ALL	-	Includes all Cache Manager permissions.

Table 2. 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.

Permission	Function	Description
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.

Combining permissions

You might need to combine permissions so that they are useful. For example, to allow "supervisors" to run stream operations but restrict "standard" users to puts and gets only, you can define the following mappings:

```
<role name="standard" permission="READ WRITE" />
<role name="supervisors" permission="READ WRITE EXEC BULK"/>
```

Reference

- Infinispan Security API

Role Mappers

Infinispan includes a `PrincipalRoleMapper` API that maps security Principals in a Subject to authorization roles. There are two role mappers available by default:

IdentityRoleMapper

Uses the Principal name as the role name.

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

CommonNameRoleMapper

Uses the Common Name (CN) as the role name if the Principal name is a Distinguished Name

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

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

You can also use custom role mappers that implement the `org.infinispan.security.PrincipalRoleMapper` interface. To configure custom role mappers declaratively, use: `<custom-role-mapper class="my.custom.RoleMapper" />`

Reference

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

6.5.2. Declaratively Configuring Authorization

Configure authorization in your `infinispan.xml` file.

Procedure

1. Configure the global authorization settings in the `cache-container` that specify a role mapper, and define a set of roles and permissions.
2. Configure authorization for caches to restrict access based on user roles.

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

① Enables Infinispan authorization for the Cache Manager.

② Specifies an implementation of `PrincipalRoleMapper` that maps Principals to roles.

③ Defines roles and their associated permissions.

④ Implicitly adds all roles from the global configuration.

If you do not want to apply all roles to a cache, explicitly define the roles that are authorized for

caches as follows:

```
<infinispan>
   <cache-container default-cache="secured" name="secured">
      <security>
         <authorization>
            <identity-role-mapper />
            <role name="admin" permissions="ALL" />
            <role name="reader" permissions="READ" />
            <role name="writer" permissions="WRITE" />
            <role name="supervisor" permissions="READ WRITE EXEC"/>
         </authorization>
      </security>
      <local-cache name="secured">
         <security>
            <authorization roles="admin supervisor reader"/> ①
         </security>
      </local-cache>
   </cache-container>

</infinispan>
```

① Defines authorized roles for the cache. In this example, users who have the `writer` role only are not authorized for the "secured" cache. Infinispan denies any access requests from those users.

Reference

- Infinispan Configuration Schema Reference

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>
   <data-source name="ds" jndi-name="jdbc/datasource" statistics="true"> ①
      <connection-factory driver="org.database.Driver" ②
                          username="db_user" ③
                          password="secret" ④
                          url="jdbc:db://database-host:10000/dbname" ⑤
                          new-connection-sql="SELECT 1" ⑥
                          transaction-isolation="READ_COMMITTED"> ⑦
         <connection-property name="name">value</connection-property> ⑧
      </connection-factory>
      <connection-pool
         initial-size="1" ⑨
         max-size="10" ⑩
         min-size="3" ⑪
         background-validation="1000" ⑫
         idle-removal="1" ⑬
         blocking-timeout="1000" ⑭
         leak-detection="10000"/> ⑮
   </data-source>
</data-sources>
```

① Defines a datasource name, JNDI name, and whether to enable statistics collection.

② Specifies the JDBC driver that creates connections. Place driver JARs in the `server/lib` directory.

③ Specifies a username for the connection.

④ Specifies a corresponding password for the connection.

⑤ Specifies the JDBC URL specific to the driver in use.

⑥ Adds a query that verifies new connections.

⑦ Configures one of the transaction isolation levels for the connection: `NONE`, `READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, `SERIALIZABLE`.

⑧ Sets optional JDBC driver-specific connection properties.

⑨ Defines the initial number of connections the pool contains.

⑩ Sets the maximum number of connections in the pool.

⑪ Sets the minimum number of connections the pool should contain.

⑫ Specifies the time, in milliseconds, between background validation runs.

⑬ Specifies the time, in minutes, a connections can remain idle before it is removed.

⑭ Specifies the amount of time, in milliseconds, to block while waiting for a connection, after which an exception is thrown.

⑮ Specifies the time, in milliseconds, a connection can be held before a leak warning occurs.

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:11.0">
    <persistence>
        <jdbc:string-keyed-jdbc-store>
            <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>
```

① Specifies the JNDI name that you provided for the datasource connection in your Infinispan

server 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 whitelist in your Infinispan configuration. Alternatively set the whitelist using system properties.

Reference

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

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 3. 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. 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.

9.1. 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.

9.1.1. Remote Cache Stores for Rolling Upgrades

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

```
<persistence passivation="false"> ①
   <remote-store xmlns="urn:infinispan:config:store:remote:11.0"
                 cache="myDistCache" ②
                 protocol-version="2.5" ③
                 hotrod-wrapping="true" ④
                 raw-values="true" ⑤
                 segmented="false"> ⑥
      <remote-server host="127.0.0.1" port="11222"/> ⑦
   </remote-store>
</persistence>
```

① Disables passivation. Remote cache stores for rolling upgrades must disable passivation.

② Matches the name of a cache in the source cluster. Target clusters load data from this cache using the remote cache store.

③ Matches the Hot Rod protocol version of the source cluster. `2.5` is the minimum version and is suitable for any upgrade paths. You do not need to set another Hot Rod version.

④ Ensures that entries are wrapped in a suitable format for the Hot Rod protocol.

⑤ Stores data in the remote cache store in raw format. This ensures that clients can use data directly from the remote cache store.

⑥ Disables segmentation for the remote cache store. You should enable segmentation for remote cache stores only if the number of segments in the target cluster matches the number of segements for the cache in the source cluster.

⑦ Points to the location of the source cluster.

Reference

- Remote cache store configuration schema
- RemoteStore
- RemoteStoreConfigurationBuilder

9.2. 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 10. 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.

10.1. 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.

10.2. 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`
     ```

10.3. 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
```

10.4. 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 11. Troubleshooting Infinispan Servers

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

11.1. 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.

11.2. 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
```

11.3. 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
}
```