Guide to Infinispan Server

Table of Contents

Infinispan Server runs in a dedicated Java Virtual Machine (JVM) and provides client access to remote caches through Hot Rod or REST endpoints. Infinispan Server speeds time to deployment by separating caches from application logic and offers built-in capabilities for monitoring and administration.

Chapter 1. Getting started with Infinispan Server

Quickly set up Infinispan Server and learn the basics.

[Get started icon] Start running Infinispan in 4 easy steps

Ansible collection

Automate installation of Infinispan clusters with our Ansible collection that optionally includes Keycloak caches and cross-site replication configuration. The Ansible collection also lets you inject Infinispan caches into the static configuration for each server instance during installation.

The Ansible collection for Infinispan is available from Ansible Galaxy.

1.1. Infinispan Server requirements

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

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

1.2. Downloading Infinispan Server distributions

The Infinispan Server distribution is an archive of Java libraries (`JAR` files) and configuration files.

Procedure

1. Download Infinispan 13.0 Server from Infinispan downloads.
2. Run the `sha1sum` command with the server download archive as the argument, for example:

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

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

Reference

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

1.3. Installing Infinispan Server

Install the Infinispan Server distribution on a host system.

Prerequisites

- Download a Infinispan Server distribution archive.

Procedure

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

```
$ unzip infinispan-server-13.0.9.Final.zip
```

The resulting directory is your `$ISPN_HOME`.

1.4. Starting Infinispan Server

Run Infinispan Server instances in a Java Virtual Machine (JVM).

Prerequisites

- Download and install the server distribution.

Procedure

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

   Linux

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

   Microsoft Windows

   ```
   bin\server.bat
   ```

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

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

Verification

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

1.5. Passing Infinispan Server configuration at startup

Specify custom configuration when you start Infinispan Server.

Infinispan Server can parse multiple configuration files that you overlay on startup with the `--server-config` argument. You can use as many configuration overlay files as required, in any order. Configuration overlay files:

- Must be valid Infinispan configuration and contain the root `server` element or field.
- Do not need to be full configuration as long as your combination of overlay files results in a full configuration.

> Infinispan Server does not detect conflicting configuration between overlay files. Each overlay file overwrites any conflicting configuration in the preceding configuration.

> If you pass cache configuration to Infinispan Server on startup it does not dynamically create those cache across the cluster. You must manually propagate caches to each node.
>
> Additionally, cache configuration that you pass to Infinispan Server on startup must include the `infinispan` and `cache-container` elements.

Prerequisites

- Download and install the server distribution.
- Add custom server configuration to the `server/conf` directory of your Infinispan Server installation.

Procedure

1. Open a terminal in `$ISPN_HOME`.
2. Specify one or more configuration files with the `--server-config=` or `-c` argument, for example:

```
$ bin/server.sh -c infinispan.xml -c datasources.yaml -c security-realms.json
```

1.6. Creating and modifying Infinispan users

Add Infinispan user credentials and assign permissions to control access to data.

Infinispan server installations use a property realm to authenticate users for the Hot Rod and REST endpoints. This means you need to create at least one user before you can access Infinispan.

By default, users also need roles with permissions to access caches and interact with Infinispan resources. You can assign roles to users individually or add users to groups that have role permissions.

You create users and assign roles with the `user` command in the Infinispan command line interface (CLI).

> Run `help user` from a CLI session to get complete command details.

1.6.1. Adding credentials

You need an `admin` user for the Infinispan Console and full control over your Infinispan environment. For this reason you should create a user with `admin` permissions the first time you add credentials.

Procedure

1. Open a terminal in `$ISPN_HOME`.
2. Create an `admin` user with the `user create` command.
   - Add a user assigned to the `admin` group.

     ```
     $ bin/cli.sh user create myuser -p changeme -g admin
     ```

   - Use implicit authorization to gain `admin` permissions.

     ```
     $ bin/cli.sh user create admin -p changeme
     ```

3. Open `user.properties` and `groups.properties` with any text editor to verify users and groups.

   ```
   $ cat server/conf/users.properties

   #$REALM_NAME=default$
   #$ALGORITHM=encrypted$
   myuser=scram-sha-1\:BYGcIAwvf6b...

   $ cat server/conf/groups.properties

   myuser=admin
   ```

1.6.2. Assigning roles to users

Assign roles to users so they have the correct permissions to access data and modify Infinispan resources.

Procedure

1. Start a CLI session with an `admin` user.

   ```
   $ bin/cli.sh
   ```

2. Assign the `deployer` role to "katie".

   ```
   [//containers/default]> user roles grant --roles=deployer katie
   ```

3. List roles for "katie".

   ```
   [//containers/default]> user roles ls katie
   ["deployer"]
   ```

1.6.3. Adding users to groups

Groups let you change permissions for multiple users. You assign a role to a group and then add users to that group. Users inherit permissions from the group role.

Procedure

1. Start a CLI session with an `admin` user.
2. Use the `user create` command to create a group.
   a. Specify "developers" as the group name with the `--groups` argument.
   b. Set a username and password for the group.

      In a property realm, a group is a special type of user that also requires a username and password.

      ```
      [//containers/default]> user create --groups=developers developers -p changeme
      ```

3. List groups.

   ```
   [//containers/default]> user ls --groups
   ["developers"]
   ```

4. Assign the `application` role to the "developers" group.

   ```
   [//containers/default]> user roles grant --roles=application developers
   ```

5. List roles for the "developers" group.

   ```
   [//containers/default]> user roles ls developers
   ["application"]
   ```

6. Add existing users, one at a time, to the group as required.

   ```
   [//containers/default]> user groups john --groups=developers
   ```

1.6.4. User roles and permissions

Infinispan includes a default set of roles that grant users with permissions to access data and interact with Infinispan resources.

`ClusterRoleMapper` is the default mechanism that Infinispan uses to associate security principals to authorization roles.

> `ClusterRoleMapper` matches principal names to role names. A user named `admin` gets `admin` permissions automatically, a user named `deployer` gets `deployer` permissions, and so on.

Role	Permissions	Description
`admin`	ALL	Superuser with all permissions including control of the Cache Manager lifecycle.
`deployer`	ALL_READ, ALL_WRITE, LISTEN, EXEC, MONITOR, CREATE	Can create and delete Infinispan resources in addition to `application` permissions.
`application`	ALL_READ, ALL_WRITE, LISTEN, EXEC, MONITOR	Has read and write access to Infinispan resources in addition to `observer` permissions. Can also listen to events and execute server tasks and scripts.
`observer`	ALL_READ, MONITOR	Has read access to Infinispan resources in addition to `monitor` permissions.
`monitor`	MONITOR	Can view statistics via JMX and the `metrics` endpoint.

Reference

- org.infinispan.security.AuthorizationPermission Enumeration
- Infinispan configuration schema reference

1.7. Verifying cluster views

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

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

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

Prerequisites

Have one instance of Infinispan Server running.

Procedure

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

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

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

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

Verification

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

Infinispan also logs the following messages when nodes join clusters:

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

INFO  [org.infinispan.CLUSTER] (jgroups-11,<server_hostname>)
ISPN100000: Node <server2_hostname> joined the cluster
```

1.8. Shutting down Infinispan Server

Stop individually running servers or bring down clusters gracefully.

Procedure

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

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

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

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

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

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

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

Verification

Infinispan logs the following messages when you shut down servers:

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

1.8.1. Infinispan cluster restarts

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

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

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

1.9. Infinispan Server installation directory structure

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

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

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

1.9.1. Server root directory

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

You can create multiple nodes under the same `$ISPN_HOME` directory or in different directories, but each Infinispan Server instance must have its own server root directory. For example, a cluster of 5 nodes could have the following server root directories on the filesystem:

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

Each server root directory should contain the following folders:

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

`server/conf`

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

Infinispan separates configuration into two layers:

Dynamic

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

Static

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

`server/data`

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

> Never directly delete or modify content in `server/data`.
>
> Modifying files such as `caches.xml` while the server is running can cause corruption. Deleting content can result in an incorrect state, which means clusters cannot restart after shutdown.

`server/lib`

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

`server/log`

Holds Infinispan Server log files.

Chapter 2. Network interfaces and socket bindings

Expose Infinispan Server through a network interface by binding it to an IP address. You can then configure endpoints to use the interface so Infinispan Server can handle requests from remote client applications.

2.1. Network interfaces

Infinispan Server multiplexes endpoints to a single TCP/IP port and automatically detects protocols of inbound client requests. You can configure how Infinispan Server binds to network interfaces to listen for client requests.

Internet Protocol (IP) address

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <!-- Selects a specific IPv4 address, which can be public, private, or loopback.
This is the default network interface for Infinispan Server. -->
  <interfaces>
    <interface name="public">
      <inet-address value="${infinispan.bind.address:127.0.0.1}"/>
    </interface>
  </interfaces>
</server>
```

JSON

```json
{
  "server": {
    "interfaces": [{
      "name": "public",
      "inet-address": {
        "value": "127.0.0.1"
      }
    }]
  }
}
```

YAML

```yaml
server:
  interfaces:
    - name: "public"
      inetAddress:
        value: "127.0.0.1"
```

Loopback address

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <!-- Selects an IP address in an IPv4 or IPv6 loopback address block. -->
  <interfaces>
    <interface name="public">
      <loopback/>
    </interface>
  </interfaces>
</server>
```

JSON

```json
{
  "server": {
    "interfaces": [{
      "name": "public",
      "loopback": null
    }]
  }
}
```

YAML

```yaml
server:
  interfaces:
    - name: "public"
      loopback: ~
```

Non-loopback address

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <!-- Selects an IP address in an IPv4 or IPv6 non-loopback address block. -->
  <interfaces>
    <interface name="public">
      <non-loopback/>
    </interface>
  </interfaces>
</server>
```

JSON

```json
{
  "server": {
    "interfaces": [{
      "name": "public",
      "non_loopback": null
    }]
  }
}
```

YAML

```yaml
server:
  interfaces:
    - name: "public"
      nonLoopback: ~
```

Any address

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <!-- Uses the `INADDR_ANY` wildcard address which means Infinispan Server listens
for inbound client requests on all interfaces. -->
  <interfaces>
    <interface name="public">
      <any-address/>
    </interface>
  </interfaces>
</server>
```

JSON

```
{
  "server": {
    "interfaces": [{
      "name": "public",
      "any_address": null
    }]
  }
}
```

YAML

```
server:
  interfaces:
    - name: "public"
      anyAddress: ~
```

Link local

XML

```
<server xmlns="urn:infinispan:server:13.0">
  <!-- Selects a link-local IP address in an IPv4 or IPv6 address block. -->
  <interfaces>
    <interface name="public">
      <link-local/>
    </interface>
  </interfaces>
</server>
```

JSON

```
{
  "server": {
    "interfaces": [{
      "name": "public",
      "link_local": null
    }]
  }
}
```

YAML

```yaml
server:
  interfaces:
    - name: "public"
      linkLocal: ~
```

Site local

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <!-- Selects a site-local (private) IP address in an IPv4 or IPv6 address block. -->
  <interfaces>
    <interface name="public">
      <site-local/>
    </interface>
  </interfaces>
</server>
```

JSON

```json
{
  "server": {
    "interfaces": [{
      "name": "public",
      "site_local": null
    }]
  }
}
```

YAML

```yaml
server:
  interfaces:
    - name: "public"
      siteLocal: ~
```

2.1.1. Match and fallback strategies

Infinispan Server can enumerate all network interfaces on the host system and bind to an interface, host, or IP address that matches a value, which can include regular expressions for additional flexibility.

Match host

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <!-- Selects an IP address that is assigned to a matching host name. -->
  <interfaces>
    <interface name="public">
      <match-host value="my_host_name"/>
    </interface>
  </interfaces>
</server>
```

JSON

```json
{
  "server": {
    "interfaces": [{
      "name": "public",
      "match-host": {
        "value": "my_host_name"
      }
    }]
  }
}
```

YAML

```yaml
server:
  interfaces:
    - name: "public"
      matchHost:
        value: "my_host_name"
```

Match interface

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <!--Selects an IP address assigned to a matching network interface. -->
  <interfaces>
    <interface name="public">
      <match-interface value="eth0"/>
    </interface>
  </interfaces>
</server>
```

JSON

```json
{
  "server": {
    "interfaces": [{
      "name": "public",
      "match-interface": {
        "value": "eth0"
      }
    }]
  }
}
```

YAML

```yaml
server:
  interfaces:
    - name: "public"
      matchInterface:
        value: "eth0"
```

Match address

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <!-- Selects an IP address that matches a regular expression. -->
  <interfaces>
    <interface name="public">
      <match-address value="132\..*"/>
    </interface>
  </interfaces>
</server>
```

JSON

```json
{
  "server": {
    "interfaces": [{
      "name": "public",
      "match-address": {
        "value": "132\\..*"
      }
    }]
  }
}
```

YAML

```yaml
server:
  interfaces:
    - name: "public"
      matchAddress:
        value: "127\\..*"
```

Fallback

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <!-- Includes multiple strategies that Infinispan Server tries in the declared order
until it finds a match. -->
  <interfaces>
    <interface name="public">
      <match-host value="my_host_name"/>
      <match-address value="132\..*"/>
      <any-address/>
    </interface>
  </interfaces>
</server>
```

JSON

```json
{
  "server": {
    "interfaces": [{
      "name": "public",
      "match-host": {
        "value": "my_host_name"
      },
      "match-address": {
        "value": "132\\..*"
      },
      "any_address": null
    }]
  }
}
```

YAML

```yaml
server:
  interfaces:
    - name: "public"
      matchHost:
        value: "my_host_name"
      matchAddress:
        value: "132\\..*"
      anyAddress: ~
```

2.2. Socket bindings

Socket bindings map endpoint connectors to network interfaces and ports. By default, Infinispan Server includes a socket binding configuration that listens on the localhost interface, `127.0.0.1`, at port `11222` for the REST and Hot Rod endpoints. If you enable the Memcached endpoint, the default socket bindings configure Infinispan Server to bind to port `11221`.

Default socket bindings

```xml
<server xmlns="urn:infinispan:server:13.0">
  <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>
</server>
```

Configuration element or attribute	Description
`socket-bindings`	Root element that contains all network interfaces and ports to which Infinispan Server endpoints can bind and listen for client connections.
`default-interface`	Declare the network interface that Infinispan Server listens on by default.
`port-offset`	Specifies the offset that Infinispan Server applies to port declarations for socket bindings.
`socket-binding`	Configures Infinispan Server to bind to a port on a network interface.

Custom socket binding declarations

The following example configuration adds an `interface` declaration named "private" and a `socket-binding` declaration that binds Infinispan Server to the private IP address:

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <interfaces>
    <interface name="public">
      <inet-address value="${infinispan.bind.address:127.0.0.1}"/>
    </interface>
    <interface name="private">
      <inet-address value="10.1.2.3"/>
    </interface>
  </interfaces>

  <socket-bindings default-interface="public"
                   port-offset="${infinispan.socket.binding.port-offset:0}">
    <socket-binding name="private_binding"
                    interface="private"
                    port="49152"/>
  </socket-bindings>

  <endpoints socket-binding="private_binding"
             security-realm="default"/>
</server>
```

JSON

```json
{
    "server": {
        "interfaces": [{
            "name": "private",
            "inet-address": {
                "value": "10.1.2.3"
            }
        }, {
            "name": "public",
            "inet-address": {
                "value": "127.0.0.1"
            }
        }],
        "socket-bindings": {
            "port-offset": "0",
            "default-interface": "public",
            "socket-binding": [{
                "name": "private_binding",
                "port": "1234",
                "interface": "private"
            }]
        },
        "endpoints": {
            "endpoint": {
                "socket-binding": "private_binding",
                "security-realm": "default"
            }
        }
    }
}
```

YAML

```
server:
  interfaces:
    - name: "private"
      inetAddress:
        value: "10.1.2.3"
    - name: "public"
      inetAddress:
        value: "127.0.0.1"
  socketBindings:
    portOffset: "0"
    defaultInterface: "public"
    socketBinding:
      - name: "private_binding"
        port: "49152"
        interface: "private"
  endpoints:
    endpoint:
      socketBinding: "private_binding"
      securityRealm: "default"
```

2.3. Changing the bind address for Infinispan Server

Infinispan Server binds to a network IP address to listen for inbound client connections on the Hot Rod and REST endpoints. You can specify that IP address directly in your Infinispan Server configuration or when starting server instances.

Prerequisites

- Have at least one Infinispan Server installation.

Procedure

Specify the IP address to which Infinispan Server binds in one of the following ways:

- Open your Infinispan Server configuration and set the value for the `inet-address` element, for example:

  ```
  <server xmlns="urn:infinispan:server:13.0">
    <interfaces>
      <interface name="custom">
        <inet-address value="${infinispan.bind.address:192.0.2.0}"/>
      </interface>
    </interfaces>
  </server>
  ```

- Use the `-b` option or the `infinispan.bind.address` system property.

Linux

```
$ bin/server.sh -b 192.0.2.0
```

Windows

```
bin\server.bat -b 192.0.2.0
```

2.3.1. Listening on all addresses

If you specify the `0.0.0.0` meta-address, or `INADDR_ANY`, as the bind address in your Infinispan Server configuration, it listens for incoming client connections on all available network interfaces.

Client intelligence

Configuring Infinispan to listen on all addresses affects how it provides Hot Rod clients with cluster topology. If there are multiple interfaces to which Infinispan Server binds, then it sends a list of IP addresses for each interface.

For example, a cluster where each server node binds to:

- `10.0.0.0/8` subnet
- `192.168.0.0/16` subnet
- `127.0.0.1` loopback

Hot Rod clients receive IP addresses for server nodes that belong to the interface through which the clients connect. If a client connects to `192.168.0.0`, for example, it does not receive any cluster topology details for nodes that listen on `10.0.0.0`.

Netmask override

Kubernetes, and some other environments, divide the IP address space into subnets and use those different subnets as a single network. For example, `10.129.2.100/23` and `10.129.4.100/23` are in different subnets but belong to the `10.0.0.0/8` network.

For this reason, Infinispan Server overrides netmasks that the host system provides with netmasks that follow IANA conventions for private networks:

- IPv4: `10.0.0.0/8`, `192.168.0.0/16`, `172.16.0.0/16`, and `169.254.0.0/16`
- IPv6: `fc00::/7` and `fe80::/10`

See `RFC 1918` for IPv4 or `RFC 4193` and `RFC 3513` for IPv6.

You can optionally configure the Hot Rod connector to use the netmask that the host system provides for interfaces with the `network-prefix-override` attribute in your Infinispan Server configuration.

Additional resources

- Infinispan Server schema reference

- RFC 1918
- RFC 4193
- RFC 3513

2.4. Infinispan Server ports and protocols

Infinispan Server provides network endpoints that allow client access with different protocols.

Port	Protocol	Description
`11222`	TCP	Hot Rod and REST
`11221`	TCP	Memcached (disabled by default)

Single port

Infinispan Server exposes multiple protocols through a single TCP port, `11222`. Handling multiple protocols with a single port simplifies configuration and reduces management complexity when deploying Infinispan clusters. Using a single port also enhances security by minimizing the attack surface on the network.

Infinispan Server handles HTTP/1.1, HTTP/2, and Hot Rod protocol requests from clients via the single port in different ways.

HTTP/1.1 upgrade headers

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

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

Client requests include Server Name Indication (SNI) mappings for Infinispan Server endpoints to negotiate protocols over a TLS connection.

Applications must use a TLS library that supports the ALPN extension. Infinispan uses WildFly OpenSSL bindings for Java.

Automatic Hot Rod detection

Client requests that include Hot Rod headers automatically route to Hot Rod endpoints.

2.4.1. Configuring network firewalls for Infinispan traffic

Adjust firewall rules to allow traffic between Infinispan Server and client applications.

Procedure

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

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

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

2.5. Specifying port offsets

Configure port offsets for multiple Infinispan Server instances on the same host. The default port offset is `0`.

Procedure

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
```

Chapter 3. Infinispan Server endpoints

Infinispan Server endpoints provide client access to the cache manager over Hot Rod and REST protocols.

3.1. Infinispan Server endpoints

3.1.1. Hot Rod

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

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

Topology state transfer

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

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

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

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

3.1.2. REST

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

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

3.1.3. Memcached

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

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

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

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

3.1.4. Comparison of endpoint protocols

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

3.1.5. Hot Rod client compatibility with Infinispan Server

Infinispan Server allows you to connect Hot Rod clients with different versions. For instance during a migration or upgrade to your Infinispan cluster, the Hot Rod client version might be a lower Infinispan version than Infinispan Server.

Infinispan recommends using the latest Hot Rod client version to benefit from the most recent capabilities and security enhancements.

Infinispan 10 and later

Hot Rod protocol version 3.x automatically negotiates the highest version possible for clients with Infinispan Server.

Infinispan 9.4 and earlier

Clients that use a Hot Rod protocol version that is higher than the Infinispan Server version must set the `infinispan.client.hotrod.protocol_version` property.

Additional resources

- Hot Rod protocol reference

3.2. Configuring Infinispan Server endpoints

Control how Hot Rod and REST endpoints bind to sockets and use security realm configuration. You can also configure multiple endpoints and disable administrative capabilities.

Each unique endpoint configuration must include both a Hot Rod connector and a REST connector. Infinispan Server implicitly includes the `hotrod-connector` and `rest-connector` elements, or fields, in an `endpoint` configuration. You should only add these elements to custom configuration to specify authentication mechanisms for endpoints.

Prerequisites

- Add socket bindings and security realms to your Infinispan Server configuration.

Procedure

1. Open your Infinispan Server configuration for editing.
2. Wrap multiple `endpoint` configurations with the `endpoints` element.
3. Specify the socket binding that the endpoint uses with the `socket-binding` attribute.
4. Specify the security realm that the endpoint uses with the `security-realm` attribute.
5. Disable administrator access with the `admin="false"` attribute, if required.

   With this configuration users cannot access Infinispan Console or the Command Line Interface (CLI) from the endpoint.

6. Save the changes to your configuration.

Multiple endpoint configuration

The following Infinispan Server configuration creates endpoints on separate socket bindings with dedicated security realms:

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <endpoints>
    <endpoint socket-binding="public"
              security-realm="application-realm"
              admin="false">
    </endpoint>
    <endpoint socket-binding="private"
              security-realm="management-realm">
    </endpoint>
  </endpoints>
</server>
```

JSON

```
{
  "server": {
    "endpoints": [{
      "socket-binding": "private",
      "security-realm": "private-realm"
    }, {
      "socket-binding": "public",
      "security-realm": "default",
      "admin": "false"
    }]
  }
}
```

YAML

```
server:
  endpoints:
   - socketBinding: public
     securityRealm: application-realm
     admin: false
   - socketBinding: private
     securityRealm: management-realm
```

Additional resources

- Network interfaces and socket bindings

3.3. Endpoint connectors

Connectors configure Hot Rod and REST endpoints to use socket bindings and security realms.

Default endpoint configuration

```
<endpoints socket-binding="default" security-realm="default"/>
```

Configuration element or attribute	Description
`endpoints`	Wraps endpoint connector configuration.
`endpoint`	Declares a Infinispan Server endpoint that configures Hot Rod and REST connectors to use a socket binding and security realm.
`hotrod-connector`	Includes the Hot Rod endpoint in the `endpoint` configuration.
`rest-connector`	Includes the Hot Rod endpoint in the `endpoint` configuration.
`memcached-connector`	Configures the Memcached endpoint and is disabled by default.

Additional resources

- Infinispan schema reference

3.4. Endpoint IP address filtering rules

Infinispan Server endpoints can use filtering rules that control whether clients can connect based on their IP addresses. Infinispan Server applies filtering rules in order until it finds a match for the client IP address.

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

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

For example:

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

IP address filter configuration

In the following configuration, Infinispan Server accepts connections only from addresses in the `192.168.0.0/16` and `10.0.0.0/8` CIDR blocks. Infinispan Server rejects all other connections.

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <endpoints>
    <endpoint socket-binding="default" security-realm="default">
      <ip-filter>
        <accept from="192.168.0.0/16"/>
        <accept from="10.0.0.0/8"/>
        <reject from="/0"/>
      </ip-filter>
    </endpoint>
  </endpoints>
</server>
```

JSON

```json
{
  "server": {
    "endpoints": {
      "endpoint": {
        "socket-binding": "default",
        "security-realm": "default",
        "ip-filter": {
          "accept-from": ["192.168.0.0/16", "10.0.0.0/8"],
          "reject-from": "/0"
        }
      }
    }
  }
}
```

YAML

```yaml
server:
  endpoints:
    endpoint:
      socketBinding: "default"
      securityRealm: "default"
      ipFilter:
        acceptFrom: ["192.168.0.0/16","10.0.0.0/8"]
        rejectFrom: "/0"
```

3.5. Inspecting and modifying rules for filtering IP addresses

Configure IP address filtering rules on Infinispan Server endpoints to accept or reject connections based on client address.

Prerequisites

- Install Infinispan Command Line Interface (CLI).

Procedure

1. Create a CLI connection to Infinispan Server.
2. Inspect and modify the IP filter rules `server connector ipfilter` command as required.
    a. List all IP filtering rules active on a connector across the cluster:

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

    b. Set IP filtering rules across the cluster.

    This command replaces any existing rules.

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

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

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

Chapter 4. Endpoint authentication mechanisms

Infinispan Server can use custom SASL and HTTP authentication mechanisms for Hot Rod and REST endpoints.

4.1. Infinispan Server authentication

Authentication restricts user access to endpoints as well as the Infinispan Console and Command Line Interface (CLI).

Infinispan Server includes a "default" security realm that enforces user authentication. Default authentication uses a property realm with user credentials stored in the `server/conf/users.properties` file. Infinispan Server also enables security authorization by default so you must assign users with permissions stored in the `server/conf/groups.properties` file.

> Use the `user create` command with the Command Line Interface (CLI) to add users and assign permissions. Run `user create --help` for examples and more information.

4.2. Configuring Infinispan Server authentication mechanisms

You can explicitly configure Hot Rod and REST endpoints to use specific authentication mechanisms. Configuring authentication mechanisms is required only if you need to explicitly override the default mechanisms for a security realm.

> Each `endpoint` section in your configuration must include `hotrod-connector` and `rest-connector` elements or fields. For example, if you explicitly declare a `hotrod-connector` you must also declare a `rest-connector` even if it does not configure an authentication mechanism.

Prerequisites

- Add security realms to your Infinispan Server configuration as required.

Procedure

1. Open your Infinispan Server configuration for editing.
2. Add an `endpoint` element or field and specify the security realm that it uses with the `security-realm` attribute.
3. Add a `hotrod-connector` element or field to configure the Hot Rod endpoint.
    a. Add an `authentication` element or field.
    b. Specify SASL authentication mechanisms for the Hot Rod endpoint to use with the `sasl mechanisms` attribute.

c. If applicable, specify SASL quality of protection settings with the `qop` attribute.

d. Specify the Infinispan Server identity with the `server-name` attribute if necessary.

4. Add a `rest-connector` element or field to configure the REST endpoint.

   a. Add an `authentication` element or field.

   b. Specify HTTP authentication mechanisms for the REST endpoint to use with the `mechanisms` attribute.

5. Save the changes to your configuration.

Authentication mechanism configuration

The following configuration specifies SASL mechanisms for the Hot Rod endpoint to use for authentication:

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <endpoints>
    <endpoint socket-binding="default"
              security-realm="my-realm">
      <hotrod-connector>
        <authentication>
          <sasl mechanisms="SCRAM-SHA-512 SCRAM-SHA-384 SCRAM-SHA-256
                            SCRAM-SHA-1 DIGEST-SHA-512 DIGEST-SHA-384
                            DIGEST-SHA-256 DIGEST-SHA DIGEST-MD5 PLAIN"
                server-name="infinispan"
                qop="auth"/>
        </authentication>
      </hotrod-connector>
      <rest-connector>
        <authentication mechanisms="DIGEST BASIC"/>
      </rest-connector>
    </endpoint>
  </endpoints>
</server>
```

JSON

```
{
  "server": {
    "endpoints": {
      "endpoint": {
        "socket-binding": "default",
        "security-realm": "my-realm",
        "hotrod-connector": {
          "authentication": {
            "security-realm": "default",
            "sasl": {
              "server-name": "infinispan",
              "mechanisms": ["SCRAM-SHA-512", "SCRAM-SHA-384", "SCRAM-SHA-256",
"SCRAM-SHA-1", "DIGEST-SHA-512", "DIGEST-SHA-384", "DIGEST-SHA-256", "DIGEST-SHA",
"DIGEST-MD5", "PLAIN"],
              "qop": ["auth"]
            }
          }
        },
        "rest-connector": {
          "authentication": {
            "mechanisms": ["DIGEST", "BASIC"],
            "security-realm": "default"
          }
        }
      }
    }
  }
}
```

YAML

```yaml
server:
  endpoints:
    endpoint:
      socketBinding: "default"
      securityRealm: "my-realm"
      hotrodConnector:
        authentication:
          securityRealm: "default"
          sasl:
            serverName: "infinispan"
            mechanisms:
              - "SCRAM-SHA-512"
              - "SCRAM-SHA-384"
              - "SCRAM-SHA-256"
              - "SCRAM-SHA-1"
              - "DIGEST-SHA-512"
              - "DIGEST-SHA-384"
              - "DIGEST-SHA-256"
              - "DIGEST-SHA"
              - "DIGEST-MD5"
              - "PLAIN"
            qop:
              - "auth"
      restConnector:
        authentication:
          mechanisms:
            - "DIGEST"
            - "BASIC"
          securityRealm: "default"
```

4.2.1. Disabling authentication

In local development environments or on isolated networks you can configure Infinispan to allow unauthenticated client requests. When you disable user authentication you should also disable authorization in your Infinispan security configuration.

Procedure

1. Open your Infinispan Server configuration for editing.
2. Remove the `security-realm` attribute from the `endpoints` element or field.
3. Remove any `authorization` elements from the `security` configuration for the `cache-container` and each cache configuration.
4. Save the changes to your configuration.

XML

```
<server xmlns="urn:infinispan:server:13.0">
  <endpoints socket-binding="default"/>
</server>
```

JSON

```
{
  "server": {
    "endpoints": {
      "endpoint": {
        "socket-binding": "default"
      }
    }
  }
}
```

YAML

```
server:
  endpoints:
    endpoint:
      socketBinding: "default"
```

4.3. Infinispan Server authentication mechanisms

Infinispan Server automatically configures endpoints with authentication mechanisms that match your security realm configuration. For example, if you add a Kerberos security realm then Infinispan Server enables the `GSSAPI` and `GS2-KRB5` authentication mechanisms for the Hot Rod endpoint.

Hot Rod endpoints

Infinispan Server enables the following SASL authentication mechanisms for Hot Rod endpoints when your configuration includes the corresponding security realm:

Security realm	**SASL authentication mechanism**
Property realms and LDAP realms	SCRAM-*, DIGEST-*, `SCRAM-*`
Token realms	OAUTHBEARER
Trust realms	EXTERNAL
Kerberos identities	GSSAPI, GS2-KRB5
SSL/TLS identities	PLAIN

REST endpoints

Infinispan Server enables the following HTTP authentication mechanisms for REST endpoints when your configuration includes the corresponding security realm:

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

4.3.1. SASL authentication mechanisms

Infinispan Server supports the following SASL authentications mechanisms with Hot Rod endpoints:

Authentication mechanism	Description	Security realm type	Related details
`PLAIN`	Uses credentials in plain-text format. You should use `PLAIN` authentication with encrypted connections only.	Property realms and LDAP realms	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.	Property realms and LDAP realms	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.	Property realms and LDAP realms	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.	Kerberos realms	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.	Kerberos realms	Similar to the `SPNEGO` HTTP mechanism.

Authentication mechanism	Description	Security realm type	Related details
`EXTERNAL`	Uses client certificates.	Trust store realms	Similar to the `CLIENT_CERT` HTTP mechanism.
`OAUTHBEARER`	Uses OAuth tokens and requires a `token-realm` configuration.	Token realms	Similar to the `BEARER_TOKEN` HTTP mechanism.

4.3.2. SASL quality of protection (QoP)

If SASL mechanisms support integrity and privacy protection (QoP) settings, you can add them to your Hot Rod endpoint 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.

4.3.3. SASL policies

SASL policies provide fine-grain control over Hot Rod authentication mechanisms.

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

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

Policy	Description	Default value
`no-anonymous`	Do not use SASL mechanisms that accept anonymous logins.	true

SASL policy configuration

In the following configuration the Hot Rod endpoint uses the `GSSAPI` mechanism for authentication because it is the only mechanism that complies with all SASL policies:

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <endpoints>
    <endpoint socket-binding="default"
              security-realm="default">
      <hotrod-connector>
        <authentication>
          <sasl mechanisms="PLAIN DIGEST-MD5 GSSAPI EXTERNAL"
                server-name="infinispan"
                qop="auth"
                policy="no-active no-plain-text"/>
        </authentication>
      </hotrod-connector>
      <rest-connector/>
    </endpoint>
  </endpoints>
</server>
```

JSON

```json
{
  "server": {
    "endpoints" : {
      "endpoint" : {
        "socket-binding" : "default",
        "security-realm" : "default",
        "hotrod-connector" : {
          "authentication" : {
            "sasl" : {
              "server-name" : "infinispan",
              "mechanisms" : [ "PLAIN","DIGEST-MD5","GSSAPI","EXTERNAL" ],
              "qop" : [ "auth" ],
              "policy" : [ "no-active","no-plain-text" ]
            }
          }
        },
        "rest-connector" : ""
      }
    }
  }
}
```

YAML

```yaml
server:
  endpoints:
    endpoint:
      socketBinding: "default"
      securityRealm: "default"
      hotrodConnector:
        authentication:
          sasl:
            serverName: "infinispan"
            mechanisms:
              - "PLAIN"
              - "DIGEST-MD5"
              - "GSSAPI"
              - "EXTERNAL"
            qop:
              - "auth"
            policy:
              - "no-active"
              - "no-plain-text"
      restConnector: ~
```

4.3.4. HTTP authentication mechanisms

Infinispan Server supports the following HTTP authentication mechanisms with REST endpoints:

Authentication mechanism	Description	Security realm type	Related details
`BASIC`	Uses credentials in plain-text format. You should use `BASIC` authentication with encrypted connections only.	Property realms and LDAP realms	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.	Property realms and LDAP realms	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.	Kerberos realms	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.	Token realms	Corresponds to the `Bearer` HTTP authentication scheme and is similar to `OAUTHBEARER` SASL mechanism.
`CLIENT_CERT`	Uses client certificates.	Trust store realms	Similar to the `EXTERNAL` SASL mechanism.

Chapter 5. Security realms

Security realms integrate Infinispan Server deployments with the network protocols and infrastructure in your environment that control access and verify user identities.

5.1. Creating security realms

Add security realms to Infinispan Server configuration to control access to deployments. You can add one or more security realm to your configuration.

When you add security realms to your configuration, Infinispan Server automatically enables the matching authentication mechanisms for the Hot Rod and REST endpoints.

Prerequisites

- Add socket bindings to your Infinispan Server configuration as required.
- Create keystores, or have a PEM file, to configure the security realm with TLS/SSL encryption.

  Infinispan Server can also generate keystores at startup.
- Provision the resources or services that the security realm configuration relies on.
  For example, if you add a token realm, you need to provision OAuth services.

This procedure demonstrates how to configure multiple property realms. Before you begin, you need to create properties files that add users and assign permissions with the Command Line Interface (CLI). Use the `user create` commands as follows:

```
user create <username> -p <changeme> -g <role> \
     --users-file=application-users.properties \
     --groups-file=application-groups.properties

user create <username> -p <changeme> -g <role> \
     --users-file=management-users.properties \
     --groups-file=management-groups.properties
```

Run `user create --help` for examples and more information.

Adding credentials to a properties realm with the CLI creates the user only on the server instance to which you are connected. You must manually synchronize credentials in a properties realm to each node in the cluster.

Procedure

1. Open your Infinispan Server configuration for editing.
2. Use the `security-realms` element in the `security` configuration to contain create multiple security realms.

3. Add a security realm with the `security-realm` element and give it a unique name with the `name` attribute.

   To follow the example, create one security realm named `application-realm` and another named `management-realm`.

4. Provide the TLS/SSL identify for Infinispan Server with the `server-identities` element and configure a keystore as required.
5. Specify the type of security realm by adding one the following elements or fields:
   - `properties-realm`
   - `ldap-realm`
   - `token-realm`
   - `truststore-realm`
6. Specify properties for the type of security realm you are configuring as appropriate.

   To follow the example, specify the `*.properties` files you created with the CLI using the `path` attribute on the `user-properties` and `group-properties` elements or fields.

7. If you add multiple different types of security realm to your configuration, include the `distributed-realm` element or field so that Infinispan Server uses the realms in combination with each other.
8. Configure Infinispan Server endpoints to use the security realm with the with the `security-realm` attribute.
9. Save the changes to your configuration.

Multiple property realms

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="application-realm">
        <properties-realm groups-attribute="Roles">
          <user-properties path="application-users.properties"/>
          <group-properties path="application-groups.properties"/>
        </properties-realm>
      </security-realm>
      <security-realm name="management-realm">
        <properties-realm groups-attribute="Roles">
          <user-properties path="management-users.properties"/>
          <group-properties path="management-groups.properties"/>
        </properties-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "management-realm",
        "properties-realm": {
          "groups-attribute": "Roles",
          "user-properties": {
            "digest-realm-name": "management-realm",
            "path": "management-users.properties"
          },
          "group-properties": {
            "path": "management-groups.properties"
          }
        }
      }, {
        "name": "application-realm",
        "properties-realm": {
          "groups-attribute": "Roles",
          "user-properties": {
            "digest-realm-name": "application-realm",
            "path": "application-users.properties"
          },
          "group-properties": {
            "path": "application-groups.properties"
          }
        }
      }]
    }
  }
}
```

YAML

```yaml
server:
  security:
    securityRealms:
      - name: "management-realm"
        propertiesRealm:
          groupsAttribute: "Roles"
          userProperties:
            digestRealmName: "management-realm"
            path: "management-users.properties"
          groupProperties:
            path: "management-groups.properties"
      - name: "application-realm"
        propertiesRealm:
          groupsAttribute: "Roles"
          userProperties:
            digestRealmName: "application-realm"
            path: "application-users.properties"
          groupProperties:
            path: "application-groups.properties"
```

5.2. Setting up Kerberos identities

Add Kerberos identities to a security realm in your Infinispan Server configuration to use *keytab* files that contain service principal names and encrypted keys, derived from Kerberos passwords.

Prerequisites

- Have Kerberos service account principals.

> *keytab* files can contain both user and service account principals. However, Infinispan Server uses service account principals only which means it 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 endpoints. For example, if you have a "datagrid" server in the "INFINISPAN.ORG" domain 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 `server/conf` directory of your Infinispan Server installation.
3. Open your Infinispan Server configuration for editing.
4. Add a `server-identities` definition to the Infinispan server security realm.
5. Specify the location of keytab files that provide service principals to Hot Rod and REST connectors.
6. Name the Kerberos service principals.
7. Save the changes to your configuration.

Kerberos identity configuration

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="kerberos-realm">
        <server-identities>
          <!-- Specifies a keytab file that provides a Kerberos identity. -->
          <!-- Names the Kerberos service principal for the Hot Rod endpoint. -->
          <!-- The required="true" attribute specifies that the keytab file must be
present when the server starts. -->
          <kerberos keytab-path="hotrod.keytab"
                    principal="hotrod/datagrid@INFINISPAN.ORG"
                    required="true"/>
          <!-- Specifies a keytab file and names the Kerberos service principal for
the REST endpoint. -->
          <kerberos keytab-path="http.keytab"
                    principal="HTTP/localhost@INFINISPAN.ORG"
                    required="true"/>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
  <endpoints>
    <endpoint socket-binding="default"
              security-realm="kerberos-realm">
     <hotrod-connector>
        <authentication>
           <sasl server-name="datagrid"
                 server-principal="hotrod/datagrid@INFINISPAN.ORG"/>
        </authentication>
     </hotrod-connector>
     <rest-connector>
       <authentication server-principal="HTTP/localhost@INFINISPAN.ORG"/>
     </rest-connector>
   </endpoint>
  </endpoints>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "kerberos-realm",
        "server-identities": [{
          "kerberos": {
            "principal": "hotrod/datagrid@INFINISPAN.ORG",
            "keytab-path": "hotrod.keytab",
            "required": true
          },
          "kerberos": {
            "principal": "HTTP/localhost@INFINISPAN.ORG",
            "keytab-path": "http.keytab",
            "required": true
          }
        }]
      }]
    },
    "endpoints": {
      "endpoint": {
        "socket-binding": "default",
        "security-realm": "kerberos-realm",
        "hotrod-connector": {
          "authentication": {
            "security-realm": "kerberos-realm",
            "sasl": {
              "server-name": "datagrid",
              "server-principal": "hotrod/datagrid@INFINISPAN.ORG"
            }
          }
        },
        "rest-connector": {
          "authentication": {
            "server-principal": "HTTP/localhost@INFINISPAN.ORG"
          }
        }
      }
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "kerberos-realm"
        serverIdentities:
          - kerberos:
              principal: "hotrod/datagrid@INFINISPAN.ORG"
              keytabPath: "hotrod.keytab"
              required: "true"
          - kerberos:
              principal: "HTTP/localhost@INFINISPAN.ORG"
              keytabPath: "http.keytab"
              required: "true"
  endpoints:
    endpoint:
      socketBinding: "default"
      securityRealm: "kerberos-realm"
      hotrodConnector:
        authentication:
          sasl:
            serverName: "datagrid"
            serverPrincipal: "hotrod/datagrid@INFINISPAN.ORG"
      restConnector:
        authentication:
          securityRealm: "kerberos-realm"
          serverPrincipal" : "HTTP/localhost@INFINISPAN.ORG"
```

5.3. Property realms

Property realms use property files to define users and groups.

- `users.properties` contains Infinispan user credentials. Passwords can be pre-digested with the `DIGEST-MD5` and `DIGEST` authentication mechanisms.
- `groups.properties` associates users with roles and permissions.

Properties files contain headers that associate them with security realms in Infinispan Server configuration.

users.properties

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

groups.properties

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

Property realm configuration

XML

```
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="default">
        <!-- groups-attribute configures the "groups.properties" file to contain
security authorization roles. -->
        <properties-realm groups-attribute="Roles">
          <user-properties path="users.properties"
                           relative-to="infinispan.server.config.path"
                           plain-text="true"/>
          <group-properties path="groups.properties"
                            relative-to="infinispan.server.config.path"/>
        </properties-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "default",
        "properties-realm": {
          "groups-attribute": "Roles",
          "user-properties": {
            "digest-realm-name": "default",
            "path": "users.properties",
            "relative-to": "infinispan.server.config.path",
            "plain-text": true
          },
          "group-properties": {
            "path": "groups.properties",
            "relative-to": "infinispan.server.config.path"
          }
        }
      }]
    }
  }
}
```

YAML

```yaml
server:
  security:
    securityRealms:
      - name: "default"
        propertiesRealm:
          # groupsAttribute configures the "groups.properties" file
          # to contain security authorization roles.
          groupsAttribute: "Roles"
          userProperties:
            digestRealmName: "default"
            path: "users.properties"
            relative-to: 'infinispan.server.config.path'
            plainText: "true"
          groupProperties:
            path: "groups.properties"
            relative-to: 'infinispan.server.config.path'
```

5.4. LDAP realms

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

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

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

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

You cannot use endpoint authentication mechanisms that perform hashing with the `direct-verification` attribute.

Because Active Directory does not expose the `password` attribute you can use the `direct-verification` attribute only and not the `user-password-mapper` element. As a result you must use the `BASIC` authentication mechanism with the REST endpoint and `PLAIN` with the Hot Rod endpoint to integrate with Active Directory Server. A more secure alternative is to use Kerberos, which allows the `SPNEGO`, `GSSAPI`, and `GS2-KRB5` authentication mechanisms.

The `rdn-identifier` attribute specifies an LDAP attribute that finds the user entry based on a provided identifier, which is typically a username; for example, the `uid` or `sAMAccountName` attribute. Add `search-recursive="true"` to the configuration to search the directory recursively. By default, the search for the user entry uses the `(rdn_identifier={0})` filter. Specify a different filter with the `filter-name` attribute.

The `attribute-mapping` element retrieves all the groups of which the user is a member. There are typically two ways in which membership information is stored:

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

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

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

LDAP realm configuration

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="ldap-realm">
        <!-- Specifies connection properties. -->
        <ldap-realm url="ldap://my-ldap-server:10389"
                    principal="uid=admin,ou=People,dc=infinispan,dc=org"
                    credential="strongPassword"
                    connection-timeout="3000"
                    read-timeout="30000"
                    connection-pooling="true"
                    referral-mode="ignore"
                    page-size="30"
                    direct-verification="true">
          <!-- Defines how principals are mapped to LDAP entries. -->
          <identity-mapping rdn-identifier="uid"
                            search-dn="ou=People,dc=infinispan,dc=org"
                            search-recursive="false">
            <!-- Retrieves all the groups of which the user is a member. -->
            <attribute-mapping>
              <attribute from="cn" to="Roles"
                         filter="(&amp;(objectClass=groupOfNames)(member={1}))"
                         filter-dn="ou=Roles,dc=infinispan,dc=org"/>
            </attribute-mapping>
          </identity-mapping>
        </ldap-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "ldap-realm",
        "ldap-realm": {
          "url": "ldap://my-ldap-server:10389",
          "principal": "uid=admin,ou=People,dc=infinispan,dc=org",
          "credential": "strongPassword",
          "connection-timeout": "3000",
          "read-timeout": "30000",
          "connection-pooling": "true",
          "referral-mode": "ignore",
          "page-size": "30",
          "direct-verification": "true",
          "identity-mapping": {
            "rdn-identifier": "uid",
            "search-dn": "ou=People,dc=infinispan,dc=org",
            "search-recursive": "false",
            "attribute-mapping": [{
              "from": "cn",
              "to": "Roles",
              "filter": "(&(objectClass=groupOfNames)(member={1}))",
              "filter-dn": "ou=Roles,dc=infinispan,dc=org"
            }]
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: ldap-realm
        ldapRealm:
          url: 'ldap://my-ldap-server:10389'
          principal: 'uid=admin,ou=People,dc=infinispan,dc=org'
          credential: strongPassword
          connectionTimeout': '3000'
          readTimeout': '30000'
          connectionPooling': true
          referralMode': ignore
          pageSize': '30'
          directVerification': true
          identityMapping:
            rdnIdentifier: uid
            searchDn: 'ou=People,dc=infinispan,dc=org'
            searchRecursive: false
            attributeMapping:
              - filter: '(&(objectClass=groupOfNames)(member={1}))'
                filterDn: 'ou=Roles,dc=infinispan,dc=org'
                from: cn
                to: Roles
```

5.4.1. LDAP realm principal re-writing

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

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="ldap-realm">
        <ldap-realm url="ldap://${org.infinispan.test.host.address}:10389"
                    principal="uid=admin,ou=People,dc=infinispan,dc=org"
                    credential="strongPassword">
          <name-rewriter>
            <!-- Defines a rewriter that extracts the username from the principal
using a regular expression. -->
            <regex-principal-transformer name="domain-remover"
                                         pattern="(.*)@INFINISPAN\.ORG"
                                         replacement="$1"/>
          </name-rewriter>
          <identity-mapping rdn-identifier="uid"
                            search-dn="ou=People,dc=infinispan,dc=org">
            <attribute-mapping>
              <attribute from="cn" to="Roles"
                         filter="(&amp;(objectClass=groupOfNames)(member={1}))"
                         filter-dn="ou=Roles,dc=infinispan,dc=org"/>
            </attribute-mapping>
            <user-password-mapper from="userPassword"/>
          </identity-mapping>
        </ldap-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "ldap-realm",
        "ldap-realm": {
          "principal": "uid=admin,ou=People,dc=infinispan,dc=org",
          "url": "ldap://${org.infinispan.test.host.address}:10389",
          "credential": "strongPassword",
          "name-rewriter": {
            "regex-principal-transformer": {
              "pattern": "(.*)@INFINISPAN\\.ORG",
              "replacement": "$1"
            }
          },
          "identity-mapping": {
            "rdn-identifier": "uid",
            "search-dn": "ou=People,dc=infinispan,dc=org",
            "attribute-mapping": {
              "attribute": {
                "filter": "(&(objectClass=groupOfNames)(member={1}))",
                "filter-dn": "ou=Roles,dc=infinispan,dc=org",
                "from": "cn",
                "to": "Roles"
              }
            },
            "user-password-mapper": {
              "from": "userPassword"
            }
          }
        }
      }]
    }
  }
}
```

YAML

```yaml
server:
  security:
    securityRealms:
      - name: "ldap-realm"
        ldapRealm:
          principal: "uid=admin,ou=People,dc=infinispan,dc=org"
          url: "ldap://${org.infinispan.test.host.address}:10389"
          credential: "strongPassword"
          nameRewriter:
            regexPrincipalTransformer:
              pattern: (.*)@INFINISPAN\.ORG
              replacement: "$1"
          identityMapping:
            rdnIdentifier: "uid"
            searchDn: "ou=People,dc=infinispan,dc=org"
            attributeMapping:
              attribute:
                filter: "(&(objectClass=groupOfNames)(member={1}))"
                filterDn: "ou=Roles,dc=infinispan,dc=org"
                from: "cn"
                to: "Roles"
            userPasswordMapper:
              from: "userPassword"
```

5.5. 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.

Token realm configuration

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="token-realm">
        <!-- Specifies the URL of the authentication server. -->
        <token-realm name="token"
                     auth-server-url="https://oauth-server/auth/">
          <!-- Specifies the URL of the token introspection endpoint. -->
          <oauth2-introspection introspection-url="https://oauth-
server/auth/realms/infinispan/protocol/openid-connect/token/introspect"
                                client-id="infinispan-server"
                                client-secret="1fdca4ec-c416-47e0-867a-3d471af7050f"/>
        </token-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "token-realm",
        "token-realm": {
          "auth-server-url": "https://oauth-server/auth/",
          "oauth2-introspection": {
            "client-id": "infinispan-server",
            "client-secret": "1fdca4ec-c416-47e0-867a-3d471af7050f",
            "introspection-url": "https://oauth-
server/auth/realms/infinispan/protocol/openid-connect/token/introspect"
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: token-realm
        tokenRealm:
          authServerUrl: 'https://oauth-server/auth/'
          oauth2Introspection:
            clientId: infinispan-server
            clientSecret: '1fdca4ec-c416-47e0-867a-3d471af7050f'
            introspectionUrl: 'https://oauth-
server/auth/realms/infinispan/protocol/openid-connect/token/introspect'
```

5.6. Trust store realms

Trust store realms use certificates, or certificates chains, that verify Infinispan Server and client identities when they negotiate connections.

Keystores

Contain server certificates that provide a Infinispan Server identity to clients. If you configure a keystore with server certificates, Infinispan Server encrypts traffic using industry standard SSL/TLS protocols.

Trust stores

Contain client certificates, or certificate chains, that clients present to Infinispan Server. Client trust stores are optional and allow Infinispan Server to perform client certificate authentication.

Client certificate authentication

You must add the `require-ssl-client-auth="true"` attribute to the endpoint configuration if you want Infinispan Server to validate or authenticate client certificates.

Trust store realm configuration

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="trust-store-realm">
        <server-identities>
          <ssl>
            <!-- Provides an SSL/TLS identity with a keystore that contains server
certificates. -->
            <keystore path="server.p12"
                      relative-to="infinispan.server.config.path"
                      keystore-password="secret"
                      alias="server"/>
            <!-- Configures a trust store that contains client certificates or part of
a certificate chain. -->
            <truststore path="trust.p12"
                        relative-to="infinispan.server.config.path"
                        password="secret"/>
          </ssl>
        </server-identities>
        <!-- Authenticates client certificates against the trust store. If you
configure this, the trust store must contain the public certificates for all clients.
-->
        <truststore-realm/>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "trust-store-realm",
        "server-identities": {
          "ssl": {
            "keystore": {
              "path": "server.p12",
              "relative-to": "infinispan.server.config.path",
              "keystore-password": "secret",
              "alias": "server"
            },
            "truststore": {
              "path": "trust.p12",
              "relative-to": "infinispan.server.config.path",
              "password": "secret"
            }
          }
        },
        "truststore-realm": {}
      }]
    }
  }
}
```

YAML

```yaml
server:
  security:
    securityRealms:
      - name: "trust-store-realm"
        serverIdentities:
          ssl:
            keystore:
              path: "server.p12"
              relative-to: "infinispan.server.config.path"
              keystore-password: "secret"
              alias: "server"
            truststore:
              path: "trust.p12"
              relative-to: "infinispan.server.config.path"
              password: "secret"
        truststoreRealm: ~
```

5.7. Distributed security realms

Distributed realms combine multiple different types of security realms. When users attempt to access the Hot Rod or REST endpoints, Infinispan Server uses each security realm in turn until it finds one that can perform the authentication.

Distributed realm configuration

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="distributed-realm">
        <ldap-realm url="ldap://my-ldap-server:10389"
                    principal="uid=admin,ou=People,dc=infinispan,dc=org"
                    credential="strongPassword">
          <identity-mapping rdn-identifier="uid"
                            search-dn="ou=People,dc=infinispan,dc=org"
                            search-recursive="false">
            <attribute-mapping>
              <attribute from="cn" to="Roles"
                         filter="(&amp;(objectClass=groupOfNames)(member={1}))"
                         filter-dn="ou=Roles,dc=infinispan,dc=org"/>
            </attribute-mapping>
          </identity-mapping>
        </ldap-realm>
        <properties-realm groups-attribute="Roles">
          <user-properties path="users.properties"
                           relative-to="infinispan.server.config.path"/>
          <group-properties path="groups.properties"
                            relative-to="infinispan.server.config.path"/>
        </properties-realm>
        <distributed-realm/>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "distributed-realm",
        "ldap-realm": {
          "principal": "uid=admin,ou=People,dc=infinispan,dc=org",
          "url": "ldap://my-ldap-server:10389",
          "credential": "strongPassword",
          "identity-mapping": {
            "rdn-identifier": "uid",
            "search-dn": "ou=People,dc=infinispan,dc=org",
            "search-recursive": false,
            "attribute-mapping": {
              "attribute": {
                "filter": "(&(objectClass=groupOfNames)(member={1}))",
                "filter-dn": "ou=Roles,dc=infinispan,dc=org",
                "from": "cn",
                "to": "Roles"
              }
            }
          }
        },
        "properties-realm": {
          "groups-attribute": "Roles",
          "user-properties": {
            "digest-realm-name": "distributed-realm",
            "path": "users.properties"
          },
          "group-properties": {
            "path": "groups.properties"
          }
        },
        "distributed-realm": {}
      }]
    }
  }
}
```

YAML

```yaml
server:
  security:
    securityRealms:
      - name: "distributed-realm"
        ldapRealm:
          principal: "uid=admin,ou=People,dc=infinispan,dc=org"
          url: "ldap://my-ldap-server:10389"
          credential: "strongPassword"
          identityMapping:
            rdnIdentifier: "uid"
            searchDn: "ou=People,dc=infinispan,dc=org"
            searchRecursive: "false"
            attributeMapping:
              attribute:
                filter: "(&(objectClass=groupOfNames)(member={1}))"
                filterDn: "ou=Roles,dc=infinispan,dc=org"
                from: "cn"
                to: "Roles"
        propertiesRealm:
          groupsAttribute: "Roles"
          userProperties:
            digestRealmName: "distributed-realm"
            path: "users.properties"
          groupProperties:
            path: "groups.properties"
        distributedRealm: ~
```

Chapter 6. Configuring TLS/SSL encryption

You can secure Infinispan Server connections using SSL/TLS encryption by configuring a keystore that contains public and private keys for Infinispan. You can also configure client certificate authentication if you require mutual TLS.

6.1. Configuring Infinispan Server keystores

Add keystores to Infinispan Server and configure it to present SSL/TLS certificates that verify its identity to clients. If a security realm contains TLS/SSL identities, it encrypts any connections to Infinispan Server endpoints that use that security realm.

Prerequisites

- Create a keystore that contains certificates, or certificate chains, for Infinispan Server.

Infinispan Server supports the following keystore formats: JKS, JCEKS, PKCS12/PFX and PEM. BKS, BCFKS, and UBER are also supported if the Bouncy Castle library is present.

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

You can use PEM files as keystores if they contain both of the following:

- A private key in PKCS#1 or PKCS#8 format.
- One or more certificates.

You should also configure PEM file keystores with an empty password (`password=""`).

Procedure

1. Open your Infinispan Server configuration for editing.
2. Add the keystore that contains SSL/TLS identities for Infinispan Server to the `$ISPN_HOME/server/conf` directory.
3. Add a `server-identities` definition to the Infinispan Server security realm.
4. Specify the keystore file name with the `path` attribute.
5. Provide the keystore password and certificate alias with the `keystore-password` and `alias` attributes.
6. Save the changes to your configuration.

Next steps

Configure clients with a trust store so they can verify SSL/TLS identities for Infinispan Server.

Keystore configuration

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="default">
        <server-identities>
          <ssl>
            <!-- Adds a keystore that contains server certificates that provide
SSL/TLS identities to clients. -->
            <keystore path="server.p12"
                      relative-to="infinispan.server.config.path"
                      password="secret"
                      alias="my-server"/>
          </ssl>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "default",
        "server-identities": {
          "ssl": {
            "keystore": {
              "alias": "my-server",
              "path": "server.p12",
              "password": "secret"
            }
          }
        }
      }]
    }
  }
}
```

YAML

```yaml
server:
  security:
    securityRealms:
      - name: "default"
        serverIdentities:
          ssl:
            keystore:
              alias: "my-server"
              path: "server.p12"
              password: "secret"
```

Additional resources

- Configuring Hot Rod client encryption

6.1.1. Generating Infinispan Server keystores

Configure Infinispan Server 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. Open your Infinispan Server configuration for editing.
2. Include the `generate-self-signed-certificate-host` attribute for the `keystore` element in the server configuration.
3. Specify a hostname for the server certificate as the value.
4. Save the changes to your configuration.

Generated keystore configuration

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="generated-keystore">
        <server-identities>
          <ssl>
            <!-- Generates a keystore that includes a self-signed certificate with the
specified hostname. -->
            <keystore path="server.p12"
                      relative-to="infinispan.server.config.path"
                      password="secret"
                      alias="server"
                      generate-self-signed-certificate-host="localhost"/>
          </ssl>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "generated-keystore",
        "server-identities": {
          "ssl": {
            "keystore": {
              "alias": "server",
              "generate-self-signed-certificate-host": "localhost",
              "path": "server.p12",
              "password": "secret"
            }
          }
        }
      }]
    }
  }
}
```

YAML

```yaml
server:
  security:
    securityRealms:
      - name: "generated-keystore"
        serverIdentities:
          ssl:
            keystore:
              alias: "server"
              generateSelfSignedCertificateHost: "localhost"
              path: "server.p12"
              password: "secret"
```

6.1.2. Configuring TLS versions and cipher suites

When using SSL/TLS encryption to secure your deployment, you can configure Infinispan Server to use specific versions of the TLS protocol as well as specific cipher suites within the protocol.

Procedure

1. Open your Infinispan Server configuration for editing.
2. Add the `engine` element to the SSL configuration for Infinispan Server.
3. Configure Infinispan to use one or more TLS versions with the `enabled-protocols` attribute.

   Infinispan Server supports TLS version 1.2 and 1.3 by default. If appropriate you can set `TLSv1.3` only to restrict the security protocol for client connections. Infinispan does not recommend enabling `TLSv1.1` because it is an older protocol with limited support and provides weak security. You should never enable any version of TLS older than 1.1.

   If you modify the SSL `engine` configuration for Infinispan Server you must explicitly configure TLS versions with the `enabled-protocols` attribute. Omitting the `enabled-protocols` attribute allows any TLS version.

   ```xml
   <engine enabled-protocols="TLSv1.3 TLSv1.2" />
   ```

4. Configure Infinispan to use one or more cipher suites with the `enabled-ciphersuites` attribute.

   You must ensure that you set a cipher suite that supports any protocol features you plan to use; for example `HTTP/2 ALPN`.

5. Save the changes to your configuration.

SSL engine configuration

XML

```
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="default">
        <server-identities>
          <ssl>
            <keystore path="server.p12"
                      relative-to="infinispan.server.config.path"
                      password="secret"
                      alias="server"/>
            <!-- Configures Infinispan Server to use specific TLS versions and cipher
suites. -->
            <engine enabled-protocols="TLSv1.3"
                    enabled-ciphersuites=
"TLS_AES_256_GCM_SHA384,TLS_AES_128_GCM_SHA256"/>
          </ssl>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "default",
        "server-identities": {
          "ssl": {
            "keystore": {
              "alias": "server",
              "path": "server.p12",
              "password": "secret"
            },
            "engine": {
              "enabled-protocols": ["TLSv1.3"],
              "enabled-ciphersuites": "TLS_AES_256_GCM_SHA384,TLS_AES_128_GCM_SHA256"
            }
          }
        }
      }]
    }
  }
}
```

YAML

```yaml
server:
  security:
    securityRealms:
      - name: "default"
        serverIdentities:
          ssl:
            keystore:
              alias: "server"
              path: "server.p12"
              password: "secret"
            engine:
              enabledProtocols:
                - "TLSv1.3"
              enabledCiphersuites: "TLS_AES_256_GCM_SHA384,TLS_AES_128_GCM_SHA256"
```

6.2. Configuring Infinispan Server on a system with FIPS 140-2 compliant cryptography

It is possible to configure Infinispan Server to present SSL/TLS certificates that verify its identity to clients. If a security realm contains TLS/SSL identities, it encrypts any connections to Infinispan Server endpoints that use that security realm.

Prerequisites

- A system in FIPS mode. Execute `fips-mode-setup --check` to ensure it is enabled.
- A correctly initialized NSS database.
- A JDK installation with the `java.security` file configured to enable the `SunPKCS11` provider pointing to the NSS database and the SSL provider initialized with the `SunPKCS11-nss-fips` argument.
- A certificate installed in the NSS database.

The OpenSSL provider requires a private key, but it is not possible to retrieve a private key from the PKCS11 store. FIPS does not allow the export of unencrypted keys from FIPS compliant cryptographic module. Therefore it is not possible to use the OpenSSL provider for TLS when in FIPS mode. Start the server with the `-Dorg.infinispan.openssl=false` argument to disable OpenSSL.

Procedure

1. Open your Infinispan Server configuration for editing.
2. Add a `server-identities` definition to the Infinispan Server security realm.
3. Specify the PKCS11 keystore with the `SunPKCS11-NSS-FIPS` provider.
4. Save the changes to your configuration.

Keystore configuration

XML

```
<server xmlns="urn:infinispan:server:13.0">
   <security>
      <security-realms>
         <security-realm name="default">
            <server-identities>
               <ssl>
                  <!-- Adds a keystore that reads certificates from the NSS database.
-->
                  <keystore provider="SunPKCS11-NSS-FIPS" type="PKCS11"/>
               </ssl>
            </server-identities>
         </security-realm>
      </security-realms>
   </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "default",
        "server-identities": {
          "ssl": {
            "keystore": {
              "provider": "SunPKCS11-NSS-FIPS",
              "type": "PKCS11"
            }
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    securityRealms:
      - name: "default"
        serverIdentities:
          ssl:
            keystore:
              provider: "SunPKCS11-NSS-FIPS"
              type: "PKCS11"
```

6.3. Configuring client certificate authentication

Configure Infinispan Server to use mutual TLS to secure client connections.

You can configure Infinispan to verify client identities from certificates in a trust store in two ways:

- Require a trust store that contains only the signing certificate, which is typically a Certificate Authority (CA). Any client that presents a certificate signed by the CA can connect to Infinispan.
- Require a trust store that contains all client certificates in addition to the signing certificate. Only clients that present a signed certificate that is present in the trust store can connect to Infinispan.

Alternatively to providing trust stores you can use shared system certificates.

Prerequisites

- Create a client trust store that contains either the CA certificate or all public certificates.
- Create a keystore for Infinispan Server and configure an SSL/TLS identity.

PEM files can be used as trust stores provided they contain one or more certificates. These trust stores should be configured with an empty password: `password=""`.

Procedure

1. Open your Infinispan Server configuration for editing.
2. Add the `require-ssl-client-auth="true"` parameter to your `endpoints` configuration.
3. Add the client trust store to the `$ISPN_HOME/server/conf` directory.
4. Specify the `path` and `password` attributes for the `truststore` element in the Infinispan Server security realm configuration.
5. Add the `<truststore-realm/>` element to the security realm if you want Infinispan Server to authenticate each client certificate.
6. Save the changes to your configuration.

Next steps

- Set up authorization with client certificates in the Infinispan Server configuration if you control access with security roles and permissions.
- Configure clients to negotiate SSL/TLS connections with Infinispan Server.

Client certificate authentication configuration

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="trust-store-realm">
        <server-identities>
          <ssl>
            <!-- Provides an SSL/TLS identity with a keystore that
                 contains server certificates. -->
            <keystore path="server.p12"
                      relative-to="infinispan.server.config.path"
                      keystore-password="secret"
                      alias="server"/>
            <!-- Configures a trust store that contains client certificates
                 or part of a certificate chain. -->
            <truststore path="trust.p12"
                        relative-to="infinispan.server.config.path"
                        password="secret"/>
          </ssl>
        </server-identities>
        <!-- Authenticates client certificates against the trust store. If you
configure this, the trust store must contain the public certificates for all clients.
-->
        <truststore-realm/>
      </security-realm>
    </security-realms>
  </security>
  <endpoints>
    <endpoint socket-binding="default"
              security-realm="trust-store-realm"
              require-ssl-client-auth="true">
      <hotrod-connector>
        <authentication>
          <sasl mechanisms="EXTERNAL"
                server-name="infinispan"
                qop="auth"/>
        </authentication>
      </hotrod-connector>
      <rest-connector>
        <authentication mechanisms="CLIENT_CERT"/>
      </rest-connector>
    </endpoint>
  </endpoints>
</server>
```

JSON

```json
{
  "server": {
    "security": {
      "security-realms": [{
        "name": "trust-store-realm",
        "server-identities": {
          "ssl": {
            "keystore": {
              "path": "server.p12",
              "relative-to": "infinispan.server.config.path",
              "keystore-password": "secret",
              "alias": "server"
            },
            "truststore": {
              "path": "trust.p12",
              "relative-to": "infinispan.server.config.path",
              "password": "secret"
            }
          }
        },
        "truststore-realm": {}
      }]
    },
    "endpoints": [{
      "socket-binding": "default",
      "security-realm": "trust-store-realm",
      "require-ssl-client-auth": "true",
      "connectors": {
        "hotrod": {
          "hotrod-connector": {
            "authentication": {
              "sasl": {
                "mechanisms": "EXTERNAL",
                "server-name": "infinispan",
                "qop": "auth"
              }
            }
          },
          "rest": {
            "rest-connector": {
              "authentication": {
                "mechanisms": "CLIENT_CERT"
              }
            }
          }
        }
      }
    }]
  }
}
```

YAML

```yaml
server:
  security:
    securityRealms:
      - name: "trust-store-realm"
        serverIdentities:
          ssl:
            keystore:
              path: "server.p12"
              relative-to: "infinispan.server.config.path"
              keystore-password: "secret"
              alias: "server"
            truststore:
              path: "trust.p12"
              relative-to: "infinispan.server.config.path"
              password: "secret"
        truststoreRealm: ~
  endpoints:
    socketBinding: "default"
    securityRealm: "trust-store-realm"
    requireSslClientAuth: "true"
    connectors:
      - hotrod:
          hotrodConnector:
            authentication:
              sasl:
                mechanisms: "EXTERNAL"
                serverName: "infinispan"
                qop: "auth"
      - rest:
          restConnector:
            authentication:
              mechanisms: "CLIENT_CERT"
```

Additional resources

- Configuring Hot Rod client encryption
- Using Shared System Certificates (Red Hat Enterprise Linux 7 Security Guide)

6.4. Configuring authorization with client certificates

Enabling client certificate authentication means you do not need to specify Infinispan user credentials in client configuration, which means you must associate roles with the Common Name (CN) field in the client certificate(s).

Prerequisites

- Provide clients with a Java keystore that contains either their public certificates or part of the certificate chain, typically a public CA certificate.

- Configure Infinispan Server to perform client certificate authentication.

Procedure

1. Open your Infinispan Server configuration for editing.
2. Enable the `common-name-role-mapper` in the security authorization configuration.
3. Assign the Common Name (`CN`) from the client certificate a role with the appropriate permissions.
4. Save the changes to your configuration.

Client certificate authorization configuration

XML

```xml
<infinispan>
  <cache-container name="certificate-authentication" statistics="true">
    <security>
      <authorization>
        <!-- Declare a role mapper that associates the common name (CN) field in
client certificate trust stores with authorization roles. -->
        <common-name-role-mapper/>
        <!-- In this example, if a client certificate contains `CN=Client1` then
clients with matching certificates get ALL permissions. -->
        <role name="Client1" permissions="ALL"/>
      </authorization>
    </security>
  </cache-container>
</infinispan>
```

JSON

```json
{
  "infinispan": {
    "cache-container": {
      "name": "certificate-authentication",
      "security": {
        "authorization": {
          "common-name-role-mapper": null,
          "roles": {
            "Client1": {
              "role": {
                "permissions": "ALL"
              }
            }
          }
        }
      }
    }
  }
}
```

YAML

```yaml
infinispan:
  cacheContainer:
    name: "certificate-authentication"
    security:
      authorization:
        commonNameRoleMapper: ~
        roles:
          Client1:
            role:
              permissions: "ALL"
```

Chapter 7. Storing Infinispan Server credentials in keystores

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

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

> Plain-text passwords in `$ISPN_HOME/server/conf` are unencrypted. Any user account with read access to the host filesystem can view plain-text passwords.
>
> While credential keystores are password-protected store encrypted passwords, any user account with write access to the host filesystem can tamper with the keystore itself.
>
> To completely secure Infinispan Server credentials, you should grant read-write access only to user accounts that can configure and run Infinispan Server.

7.1. Setting up credential keystores

Create keystores that encrypt credential for Infinispan Server access.

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

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

Procedure

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

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

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

   Linux

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

Microsoft Windows

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

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

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

4. Configure Infinispan to use the credential keystore.

   a. Specify the name and location of the credential keystore in the `credential-stores` configuration.

   b. Provide the credential keystore and alias in the `credential-reference` configuration.

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

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

7.2. Credential keystore configuration

This topic provides examples of credential keystores in Infinispan Server configuration.

Credential keystores

XML

```
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <!-- Uses a keystore to manage server credentials. -->
    <credential-stores>
      <!-- Specifies the name and filesystem location of a keystore. -->
      <credential-store name="credentials" path="credentials.pfx">
        <!-- Specifies the password for the credential keystore. -->
        <clear-text-credential clear-text="secret1234!"/>
      </credential-store>
    </credential-stores>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "credential-stores": [{
        "name": "credentials",
        "path": "credentials.pfx",
        "clear-text-credential": {
          "clear-text": "secret1234!"
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    credentialStores:
      - name: credentials
        path: credentials.pfx
        clearTextCredential:
          clearText: "secret1234!"
```

Datasource connections

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <data-sources>
    <data-source name="postgres"
                 jndi-name="jdbc/postgres">
      <!-- Specifies the database username in the connection factory. -->
      <connection-factory driver="org.postgresql.Driver"
                          username="dbuser"
                          url="${org.infinispan.server.test.postgres.jdbcUrl}">
        <!-- Specifies the credential keystore that contains an encrypted password and
the alias for it. -->
        <credential-reference store="credentials"
                              alias="dbpassword"/>
      </connection-factory>
      <connection-pool max-size="10"
                       min-size="1"
                       background-validation="1000"
                       idle-removal="1"
                       initial-size="1"
                       leak-detection="10000"/>
    </data-source>
  </data-sources>
</server>
```

JSON

```json
{
  "server": {
    "data-sources": [{
      "name": "postgres",
      "jndi-name": "jdbc/postgres",
      "connection-factory": {
        "driver": "org.postgresql.Driver",
        "username": "dbuser",
        "url": "${org.infinispan.server.test.postgres.jdbcUrl}",
        "credential-reference": {
          "store": "credentials",
          "alias": "dbpassword"
        }
      }
    }]
  }
}
```

YAML

```yaml
server:
  dataSources:
    - name: postgres
      jndiName: jdbc/postgres
      connectionFactory:
        driver: org.postgresql.Driver
        username: dbuser
        url: '${org.infinispan.server.test.postgres.jdbcUrl}'
        credentialReference:
          store: credentials
          alias: dbpassword
```

LDAP connections

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <credential-stores>
      <credential-store name="credentials"
                        path="credentials.pfx">
        <clear-text-credential clear-text="secret1234!"/>
      </credential-store>
    </credential-stores>
    <security-realms>
      <security-realm name="default">
        <!-- Specifies the LDAP principal in the connection factory. -->
        <ldap-realm name="ldap"
                    url="ldap://my-ldap-server:10389"
                    principal="uid=admin,ou=People,dc=infinispan,dc=org">
          <!-- Specifies the credential keystore that contains an encrypted password
and the alias for it. -->
          <credential-reference store="credentials"
                                alias="ldappassword"/>
        </ldap-realm>
      </security-realm>
    </security-realms>
  </security>
</server>
```

JSON

```
{
  "server": {
    "security": {
      "credential-stores": [{
        "name": "credentials",
        "path": "credentials.pfx",
        "clear-text-credential": {
          "clear-text": "secret1234!"
        }
      }],
      "security-realms": [{
        "name": "default",
        "ldap-realm": {
          "name": "ldap",
          "url": "ldap://my-ldap-server:10389",
          "principal": "uid=admin,ou=People,dc=infinispan,dc=org",
          "credential-reference": {
            "store": "credentials",
            "alias": "ldappassword"
          }
        }
      }]
    }
  }
}
```

YAML

```
server:
  security:
    credentialStores:
      - name: credentials
        path: credentials.pfx
        clearTextCredential:
          clearText: "secret1234!"
    securityRealms:
      - name: "default"
        ldapRealm:
          name: ldap
          url: 'ldap://my-ldap-server:10389'
          principal: 'uid=admin,ou=People,dc=infinispan,dc=org'
          credentialReference:
            store: credentials
            alias: ldappassword
```

Chapter 8. Configuring user roles and permissions

Authorization is a security feature that requires users to have certain permissions before they can access caches or interact with Infinispan resources. You assign roles to users that provide different levels of permissions, from read-only access to full, super user privileges.

8.1. Security authorization

Infinispan authorization secures your deployment by restricting user access.

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

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

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

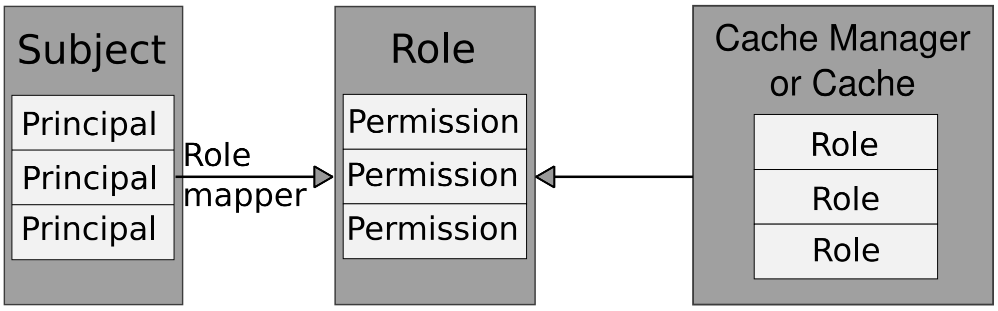
Identities

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

Identities to roles

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

The following image illustrates how security principals correspond to roles:



8.1.1. User roles and permissions

Infinispan includes a default set of roles that grant users with permissions to access data and interact with Infinispan resources.

ClusterRoleMapper is the default mechanism that Infinispan uses to associate security principals to authorization roles.

> ClusterRoleMapper matches principal names to role names. A user named admin gets admin permissions automatically, a user named deployer gets deployer permissions, and so on.

Role	Permissions	Description
admin	ALL	Superuser with all permissions including control of the Cache Manager lifecycle.
deployer	ALL_READ, ALL_WRITE, LISTEN, EXEC, MONITOR, CREATE	Can create and delete Infinispan resources in addition to application permissions.
application	ALL_READ, ALL_WRITE, LISTEN, EXEC, MONITOR	Has read and write access to Infinispan resources in addition to observer permissions. Can also listen to events and execute server tasks and scripts.
observer	ALL_READ, MONITOR	Has read access to Infinispan resources in addition to monitor permissions.
monitor	MONITOR	Can view statistics via JMX and the metrics endpoint.

Reference

- org.infinispan.security.AuthorizationPermission Enumeration
- Infinispan configuration schema reference

8.1.2. Permissions

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

Cache Manager permissions

Permission	Function	Description
CONFIGURATION	defineConfiguration	Defines new cache configurations.
LISTEN	addListener	Registers listeners against a Cache Manager.
LIFECYCLE	stop	Stops the Cache Manager.

Permission	Function	Description
CREATE	`createCache`, `removeCache`	Create and remove container resources such as caches, counters, schemas, and scripts.
MONITOR	`getStats`	Allows access to JMX statistics and the `metrics` endpoint.
ALL	-	Includes all Cache Manager permissions.

Cache permissions

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

Permission	Function	Description
ALL_WRITE	-	Combines the WRITE and BULK_WRITE permissions.

Additional resources

- Infinispan Security API

8.1.3. Role mappers

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

Cluster role mappers

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

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

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

Identity role mappers

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

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

CommonName role mappers

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

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

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

Custom role mappers

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

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

Additional resources

- Infinispan Security API

- org.infinispan.security.PrincipalRoleMapper

8.2. Access control list (ACL) cache

Infinispan caches roles that you grant to users internally for optimal performance. Whenever you grant or deny roles to users, Infinispan flushes the ACL cache to ensure user permissions are applied correctly.

If necessary, you can disable the ACL cache or configure it with the `cache-size` and `cache-timeout` attributes.

XML

```
<infinispan>
  <cache-container name="acl-cache-configuration">
    <security cache-size="1000"
              cache-timeout="300000">
      <authorization/>
    </security>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "name" : "acl-cache-configuration",
      "security" : {
        "cache-size" : "1000",
        "cache-timeout" : "300000",
        "authorization" : {}
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    name: "acl-cache-configuration"
    security:
      cache-size: "1000"
      cache-timeout: "300000"
      authorization: ~
```

Additional resources

* Infinispan configuration schema reference

8.3. Customizing roles and permissions

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

Procedure

1. Declare a role mapper and a set of custom roles and permissions in the Cache Manager configuration.
2. Configure authorization for caches to restrict access based on user roles.

Custom roles and permissions configuration

XML

```xml
<infinispan>
  <cache-container name="custom-authorization">
    <security>
      <authorization>
        <!-- Declare a role mapper that associates a security principal
             to each role. -->
        <identity-role-mapper />
        <!-- Specify user roles and corresponding permissions. -->
        <role name="admin" permissions="ALL" />
        <role name="reader" permissions="READ" />
        <role name="writer" permissions="WRITE" />
        <role name="supervisor" permissions="READ WRITE EXEC"/>
      </authorization>
    </security>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "name" : "custom-authorization",
      "security" : {
        "authorization" : {
          "identity-role-mapper" : null,
          "roles" : {
            "reader" : {
              "role" : {
                "permissions" : "READ"
              }
            },
            "admin" : {
              "role" : {
                "permissions" : "ALL"
              }
            },
            "writer" : {
              "role" : {
                "permissions" : "WRITE"
              }
            },
            "supervisor" : {
              "role" : {
                "permissions" : "READ WRITE EXEC"
              }
            }
          }
        }
      }
    }
  }
}
```

YAML

```yaml
infinispan:
  cacheContainer:
    name: "custom-authorization"
    security:
      authorization:
        identityRoleMapper: "null"
        roles:
          reader:
            role:
              permissions: "READ"
          admin:
            role:
              permissions: "ALL"
          writer:
            role:
              permissions: "WRITE"
          supervisor:
            role:
              permissions: "READ WRITE EXEC"
```

8.4. Configuring caches with security authorization

Use authorization in your cache configuration to restrict user access. Before they can read or write cache entries, or create and delete caches, users must have a role with a sufficient level of permission.

Prerequisites

- Ensure the `authorization` element is included in the `security` section of the `cache-container` configuration.

  Infinispan enables security authorization in the Cache Manager by default and provides a global set of roles and permissions for caches.

- If necessary, declare custom roles and permissions in the Cache Manager configuration.

Procedure

1. Open your cache configuration for editing.
2. Add the `authorization` element to caches to restrict user access based on their roles and permissions.
3. Save the changes to your configuration.

Authorization configuration

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

XML

```xml
<distributed-cache>
  <security>
    <!-- Inherit authorization settings from the cache-container. --> <authorization/>
  </security>
</distributed-cache>
```

JSON

```json
{
  "distributed-cache": {
    "security": {
      "authorization": {
        "enabled": true
      }
    }
  }
}
```

YAML

```yaml
distributedCache:
  security:
    authorization:
      enabled: true
```

Custom roles and permissions

XML

```xml
<distributed-cache>
  <security>
    <authorization roles="admin supervisor"/>
  </security>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "security": {
      "authorization": {
        "enabled": true,
        "roles": ["admin","supervisor"]
      }
    }
  }
}
```

YAML

```
distributedCache:
  security:
    authorization:
      enabled: true
      roles: ["admin","supervisor"]
```

8.5. Disabling security authorization

In local development environments you can disable authorization so that users do not need roles and permissions. Disabling security authorization means that any user can access data and interact with Infinispan resources.

Procedure

1. Open your Infinispan configuration for editing.
2. Remove any `authorization` elements from the `security` configuration for the Cache Manager.
3. Remove any `authorization` configuration from your caches.
4. Save the changes to your configuration.

Chapter 9. Enabling and configuring Infinispan statistics and JMX monitoring

Infinispan can provide Cache Manager and cache statistics as well as export JMX MBeans.

9.1. Enabling statistics in remote caches

Infinispan Server automatically enables statistics for the default cache manager. However, you must explicitly enable statistics for your caches.

Procedure

1. Open your Infinispan configuration for editing.
2. Add the `statistics` attribute or field and specify `true` as the value.
3. Save and close your Infinispan configuration.

Remote cache statistics

XML

```xml
<distributed-cache statistics="true" />
```

JSON

```json
{
  "distributed-cache": {
    "statistics": "true"
  }
}
```

YAML

```yaml
distributedCache:
  statistics: true
```

9.2. Enabling Hot Rod client statistics

Hot Rod Java clients can provide statistics that include remote cache and near-cache hits and misses as well as connection pool usage.

Procedure

1. Open your Hot Rod Java client configuration for editing.
2. Set `true` as the value for the `statistics` property or invoke the `statistics().enable()` methods.
3. Export JMX MBeans for your Hot Rod client with the `jmx` and `jmx_domain` properties or invoke the

`jmxEnable()` and `jmxDomain()` methods.

4. Save and close your client configuration.

Hot Rod Java client statistics

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.statistics().enable()
         .jmxEnable()
         .jmxDomain("my.domain.org")
       .addServer()
         .host("127.0.0.1")
         .port(11222);
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(builder.build());
```

hotrod-client.properties

```
infinispan.client.hotrod.statistics = true
infinispan.client.hotrod.jmx = true
infinispan.client.hotrod.jmx_domain = my.domain.org
```

9.3. Configuring Infinispan metrics

Infinispan generates metrics that are compatible with the MicroProfile Metrics API.

- Gauges provide values such as the average number of nanoseconds for write operations or JVM uptime.
- Histograms provide details about operation execution times such as read, write, and remove times.

By default, Infinispan generates gauges when you enable statistics but you can also configure it to generate histograms.

Procedure

1. Open your Infinispan configuration for editing.
2. Add the `metrics` element or object to the cache container.
3. Enable or disable gauges with the `gauges` attribute or field.
4. Enable or disable histograms with the `histograms` attribute or field.
5. Save and close your client configuration.

Metrics configuration

XML

```xml
<infinispan>
  <cache-container statistics="true">
    <metrics gauges="true"
             histograms="true" />
  </cache-container>
</infinispan>
```

JSON

```json
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
      "metrics" : {
        "gauges" : "true",
        "histograms" : "true"
      }
    }
  }
}
```

YAML

```yaml
infinispan:
  cacheContainer:
    statistics: "true"
    metrics:
      gauges: "true"
      histograms: "true"
```

Verification

Infinispan Server exposes statistics through the `metrics` endpoint. You can collect metrics with any monitoring tool that supports the OpenMetrics format, such as Prometheus.

Infinispan metrics are provided at the `vendor` scope. Metrics related to the JVM are provided in the `base` scope.

You can retrieve metrics from Infinispan Server as follows:

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

To retrieve metrics in MicroProfile JSON format, do the following:

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

Additional resources

- Eclipse MicroProfile Metrics

9.4. Registering JMX MBeans

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

Procedure

1. Open your Infinispan configuration for editing.
2. Add the `jmx` element or object to the cache container and specify `true` as the value for the `enabled` attribute or field.
3. Add the `domain` attribute or field and specify the domain where JMX MBeans are exposed, if required.
4. Save and close your client configuration.

JMX configuration

XML

```xml
<infinispan>
  <cache-container statistics="true">
    <jmx enabled="true"
         domain="example.com"/>
  </cache-container>
</infinispan>
```

JSON

```json
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
      "jmx" : {
        "enabled" : "true",
        "domain" : "example.com"
      }
    }
  }
}
```

YAML

```yaml
infinispan:
  cacheContainer:
    statistics: "true"
    jmx:
      enabled: "true"
      domain: "example.com"
```

9.4.1. Enabling JMX remote ports

Provide unique remote JMX ports to expose Infinispan MBeans through connections in JMXServiceURL format.

> Infinispan Server does not expose JMX remotely via the single port endpoint. If you want to remotely access Infinispan Server via JMX you must enable a remote port.

Procedure

- Pass the following system properties to Infinispan at startup:

  ```
  -Dcom.sun.management.jmxremote
  -Dcom.sun.management.jmxremote.port=9999
  -Dcom.sun.management.jmxremote.authenticate=false
  -Dcom.sun.management.jmxremote.ssl=false
  ```

9.4.2. 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.

Additional resources

- Infinispan JMX Components

9.4.3. Registering MBeans in custom MBean servers

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

Prerequisites

- Create an implementation of `MBeanServerLookup` so that the `getMBeanServer()` method returns the custom MBeanServer instance.
- Configure Infinispan to register JMX MBeans.

Procedure

1. Open your Infinispan configuration for editing.
2. Add the `mbean-server-lookup` attribute or field to the JMX configuration for the cache manager.
3. Specify fully qualified name (FQN) of your `MBeanServerLookup` implementation.
4. Save and close your client configuration.

JMX MBean server lookup configuration

XML

```
<infinispan>
  <cache-container statistics="true">
    <jmx enabled="true"
         domain="example.com"
         mbean-server-lookup="com.example.MyMBeanServerLookup"/>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
      "jmx" : {
        "enabled" : "true",
        "domain" : "example.com",
        "mbean-server-lookup" : "com.example.MyMBeanServerLookup"
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    statistics: "true"
    jmx:
      enabled: "true"
      domain: "example.com"
      mbeanServerLookup: "com.example.MyMBeanServerLookup"
```

Chapter 10. Adding managed datasources to Infinispan Server

Optimize connection pooling and performance for JDBC database connections by adding managed datasources to your Infinispan Server configuration.

10.1. Configuring managed datasources

Create managed datasources as part of your Infinispan Server configuration to optimize connection pooling and performance for JDBC database connections. You can then specify the JDNI name of the managed datasources in your caches, which centralizes JDBC connection configuration for your deployment.

Prerequisites

- Copy database drivers to the `server/lib` directory in your Infinispan Server installation.

Use the `install` command with the Infinispan Command Line Interface (CLI) to download the required drivers to the `server/lib` directory, for example:

```
install org.postgresql:postgresql:42.1.3
```

Procedure

1. Open your Infinispan Server configuration for editing.
2. Add a new `data-source` to the `data-sources` section.
3. Uniquely identify the datasource with the `name` attribute or field.
4. Specify a JNDI name for the datasource with the `jndi-name` attribute or field.

   You use the JNDI name to specify the datasource in your JDBC cache store configuration.

5. Set `true` as the value of the `statistics` attribute or field to enable statistics for the datasource through the `/metrics` endpoint.
6. Provide JDBC driver details that define how to connect to the datasource in the `connection-factory` section.
   a. Specify the name of the database driver with the `driver` attribute or field.
   b. Specify the JDBC connection url with the `url` attribute or field.
   c. Specify credentials with the `username` and `password` attributes or fields.
   d. Provide any other configuration as appropriate.
7. Define how Infinispan Server nodes pool and reuse connections with connection pool tuning properties in the `connection-pool` section.

8. Save the changes to your configuration.

Verification

Use the Infinispan Command Line Interface (CLI) to test the datasource connection, as follows:

1. Start a CLI session.

   ```
   $ bin/cli.sh
   ```

2. List all datasources and confirm the one you created is available.

   ```
   [//containers/default]> server datasource ls
   ```

3. Test a datasource connection.

   ```
   [//containers/default]> server datasource test my-datasource
   ```

Managed datasource configuration

XML

```xml
<server xmlns="urn:infinispan:server:13.0">
  <data-sources>
     <!-- Defines a unique name for the datasource and JNDI name that you
          reference in JDBC cache store configuration.
          Enables statistics for the datasource, if required. -->
     <data-source name="ds"
                  jndi-name="jdbc/postgres"
                  statistics="true">
        <!-- Specifies the JDBC driver that creates connections. -->
        <connection-factory driver="org.postgresql.Driver"
                            url="jdbc:postgresql://localhost:5432/postgres"
                            username="postgres"
                            password="changeme">
           <!-- Sets optional JDBC driver-specific connection properties. -->
           <connection-property name="name">value</connection-property>
        </connection-factory>
        <!-- Defines connection pool tuning properties. -->
        <connection-pool initial-size="1"
                         max-size="10"
                         min-size="3"
                         background-validation="1000"
                         idle-removal="1"
                         blocking-timeout="1000"
                         leak-detection="10000"/>
     </data-source>
  </data-sources>
</server>
```

JSON

```json
{
  "server": {
    "data-sources": [{
      "name": "ds",
      "jndi-name": "jdbc/postgres",
      "statistics": true,
      "connection-factory": {
        "driver": "org.postgresql.Driver",
        "url": "jdbc:postgresql://localhost:5432/postgres",
        "username": "postgres",
        "password": "changeme",
        "connection-properties": {
          "name": "value"
        }
      },
      "connection-pool": {
        "initial-size": 1,
        "max-size": 10,
        "min-size": 3,
        "background-validation": 1000,
        "idle-removal": 1,
        "blocking-timeout": 1000,
        "leak-detection": 10000
      }
    }]
  }
}
```

YAML

```
server:
  dataSources:
    - name: ds
      jndiName: 'jdbc/postgres'
      statistics: true
      connectionFactory:
        driver: "org.postgresql.Driver"
        url: "jdbc:postgresql://localhost:5432/postgres"
        username: "postgres"
        password: "changeme"
        connectionProperties:
          name: value
      connectionPool:
        initialSize: 1
        maxSize: 10
        minSize: 3
        backgroundValidation: 1000
        idleRemoval: 1
        blockingTimeout: 1000
        leakDetection: 10000
```

10.2. Configuring caches with JNDI names

When you add a managed datasource to Infinispan Server you can add the JNDI name to a JDBC-based cache store configuration.

Prerequisites

- Configure Infinispan Server with a managed datasource.

Procedure

1. Open your cache configuration for editing.
2. Add the `data-source` element or field to the JDBC-based cache store configuration.
3. Specify the JNDI name of the managed datasource as the value of the `jndi-url` attribute.
4. Configure the JDBC-based cache stores as appropriate.
5. Save the changes to your configuration.

JNDI name in cache configuration

XML

```
<distributed-cache>
  <persistence>
    <jdbc:string-keyed-jdbc-store>
      <!-- Specifies the JNDI name of a managed datasource on Infinispan Server. -->
      <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>
```

JSON

```json
{
  "distributed-cache": {
    "persistence": {
      "string-keyed-jdbc-store": {
        "data-source": {
          "jndi-url": "jdbc/postgres"
          },
        "string-keyed-table": {
          "prefix": "TBL",
          "drop-on-exit": true,
          "create-on-start": true,
          "id-column": {
            "name": "ID",
            "type": "VARCHAR(255)"
          },
          "data-column": {
            "name": "DATA",
            "type": "BYTEA"
          },
          "timestamp-column": {
            "name": "TS",
            "type": "BIGINT"
          },
          "segment-column": {
            "name": "S",
            "type": "INT"
          }
        }
      }
    }
  }
}
```

YAML

```yaml
distributedCache:
  persistence:
    stringKeyedJdbcStore:
      dataSource:
        jndi-url: "jdbc/postgres"
      stringKeyedTable:
        prefix: "TBL"
        dropOnExit: true
        createOnStart: true
        idColumn:
          name: "ID"
          type: "VARCHAR(255)"
        dataColumn:
          name: "DATA"
          type: "BYTEA"
        timestampColumn:
          name: "TS"
          type: "BIGINT"
        segmentColumn:
          name: "S"
          type: "INT"
```

10.3. Connection pool tuning properties

You can tune JDBC connection pools for managed datasources in your Infinispan Server configuration.

Property	Description
`initial-size`	Initial number of connections the pool should hold.
`max-size`	Maximum number of connections in the pool.
`min-size`	Minimum number of connections the pool should hold.
`blocking-timeout`	Maximum time in milliseconds to block while waiting for a connection before throwing an exception. This will never throw an exception if creating a new connection takes an inordinately long period of time. Default is `0` meaning that a call will wait indefinitely.
`background-validation`	Time in milliseconds between background validation runs. A duration of `0` means that this feature is disabled.
`validate-on-acquisition`	Connections idle for longer than this time, specified in milliseconds, are validated before being acquired (foreground validation). A duration of `0` means that this feature is disabled.
`idle-removal`	Time in minutes a connection has to be idle before it can be removed.
`leak-detection`	Time in milliseconds a connection has to be held before a leak warning.

Chapter 11. Setting up Infinispan cluster transport

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

11.1. Default JGroups stacks

Infinispan provides default JGroups stack files, `default-jgroups-*.xml`, in the `default-configs` directory inside the `infinispan-core-13.0.9.Final.jar` file.

You can find this JAR file in the `$ISPN_HOME/lib` directory.

File name	Stack name	Description
`default-jgroups-udp.xml`	`udp`	Uses UDP for transport and UDP multicast for discovery. Suitable for larger clusters (over 100 nodes) or if you are using replicated caches or invalidation mode. Minimizes the number of open sockets.
`default-jgroups-tcp.xml`	`tcp`	Uses TCP for transport and the `MPING` protocol for discovery, which uses `UDP` multicast. Suitable for smaller clusters (under 100 nodes) *only if* you are using distributed caches because TCP is more efficient than UDP as a point-to-point protocol.
`default-jgroups-kubernetes.xml`	`kubernetes`	Uses TCP for transport and `DNS_PING` for discovery. Suitable for Kubernetes and Red Hat OpenShift nodes where UDP multicast is not always available.
`default-jgroups-ec2.xml`	`ec2`	Uses TCP for transport and `NATIVE_S3_PING` for discovery. Suitable for Amazon EC2 nodes where UDP multicast is not available. Requires additional dependencies.
`default-jgroups-google.xml`	`google`	Uses TCP for transport and `GOOGLE_PING2` for discovery. Suitable for Google Cloud Platform nodes where UDP multicast is not available. Requires additional dependencies.
`default-jgroups-azure.xml`	`azure`	Uses TCP for transport and `AZURE_PING` for discovery. Suitable for Microsoft Azure nodes where UDP multicast is not available. Requires additional dependencies.

Additional resources

* JGroups Protocols

11.2. Cluster discovery protocols

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

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

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

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

Additional resources

* JGroups Discovery Protocols

11.2.1. PING

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

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

PING configuration example

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

Additional resources

* JGroups PING

11.2.2. TCPPING

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

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

TCPPING configuration example

```
<TCP bind_port="7800" />
<TCPPING timeout="3000"
         initial_hosts="${jgroups.tcpping.initial_hosts:hostname1[port1],hostname2[por
t2]}"
          port_range="0"
          num_initial_members="3"/>
```

Additional resources

- JGroups TCPPING

11.2.3. MPING

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

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

MPING configuration example

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

Additional resources

- JGroups MPING

11.2.4. TCPGOSSIP

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

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

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

Gossip router configuration example

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

Additional resources

- JGroups Gossip Router

11.2.5. JDBC_PING

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

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

JDBC_PING configuration example

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

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

Additional resources

- JDBC_PING
- JDBC_PING Wiki

11.2.6. DNS_PING

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

DNS_PING configuration example

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

Additional resources

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

11.2.7. Cloud discovery protocols

Infinispan includes default JGroups stacks that use discovery protocol implementations that are specific to cloud providers.

Discovery protocol	Default stack file	Artifact	Version
NATIVE_S3_PING	default-jgroups-ec2.xml	org.jgroups.aws.s3:native-s3-ping	1.0.0.Final

Discovery protocol	Default stack file	Artifact	Version
`GOOGLE_PING2`	`default-jgroups-google.xml`	`org.jgroups.google:jgroups-google`	`1.0.0.Final`
`AZURE_PING`	`default-jgroups-azure.xml`	`org.jgroups.azure:jgroups-azure`	`1.3.0.Final`

Providing dependencies for cloud discovery protocols

To use `NATIVE_S3_PING`, `GOOGLE_PING2`, or `AZURE_PING` cloud discovery protocols, you need to provide dependent libraries to Infinispan.

Procedure

1. Download the artifact JAR file and all dependencies.
2. Add the artifact JAR file and all dependencies to the `$ISPN_HOME/server/lib` directory of your Infinispan Server installation.

   For more details see the Downloading artifacts for JGroups cloud discover protocols for Data Grid Server ({RedHat} knowledgebase article)

You can then configure the cloud discovery protocol as part of a JGroups stack file or with system properties.

Additional resources

- JGroups NATIVE_S3_PING
- JGroups GOOGLE_PING2
- JGroups AZURE_PING

11.3. Using the default JGroups stacks

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

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

Procedure

Do one of the following to use one of the default JGroups stacks:

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

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

- Use the `cluster-stack` argument to set the JGroups stack file when Infinispan Server starts:

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

Verification

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

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

11.4. Customizing JGroups stacks

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

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

Procedure

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

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

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

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

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

11.4.1. Inheritance attributes

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

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

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

11.5. Using JGroups system properties

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

Procedure

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

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

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

11.5.1. Cluster transport properties

Use the following properties to customize JGroups cluster transport.

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

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

Additional resources

- JGroups system properties
- JGroups protocol list

11.5.2. System properties for cloud discovery protocols

Use the following properties to configure JGroups discovery protocols for hosted platforms.

Amazon EC2

System properties for configuring `NATIVE_S3_PING`.

System Property	Description	Default Value	Required/Optional
`jgroups.s3.region_name`	Name of the Amazon S3 region.	No default value.	Optional
`jgroups.s3.bucket_name`	Name of the Amazon S3 bucket. The name must exist and be unique.	No default value.	Optional

Google Cloud Platform

System properties for configuring `GOOGLE_PING2`.

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

Azure

System properties for `AZURE_PING`.

System Property	Description	Default Value	Required/Optional
`jboss.jgroups.azure_ping.storage_account_name`	Name of the Azure storage account. The name must exist and be unique.	No default value.	Required
`jboss.jgroups.azure_ping.storage_access_key`	Name of the Azure storage access key.	No default value.	Required
`jboss.jgroups.azure_ping.container`	Valid DNS name of the container that stores ping information.	No default value.	Required

Kubernetes

System properties for `DNS_PING`.

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

11.6. Using inline JGroups stacks

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

Procedure

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

```xml
<infinispan>
  <!-- Contains one or more JGroups stack definitions. -->
  <jgroups>
    <!-- Defines a custom JGroups stack named "prod". -->
    <stack name="prod">
      <TCP bind_port="7800" port_range="30" recv_buf_size="20000000" send_buf_size
="640000"/>
      <MPING break_on_coord_rsp="true"
             mcast_addr="${jgroups.mping.mcast_addr:228.2.4.6}"
             mcast_port="${jgroups.mping.mcast_port:43366}"
             num_discovery_runs="3"
             ip_ttl="${jgroups.udp.ip_ttl:2}"/>
      <MERGE3 />
      <FD_SOCK />
      <FD_ALL timeout="3000" interval="1000" timeout_check_interval="1000" />
      <VERIFY_SUSPECT timeout="1000" />
      <pbcast.NAKACK2 use_mcast_xmit="false" xmit_interval="200"
xmit_table_num_rows="50"
                      xmit_table_msgs_per_row="1024"
xmit_table_max_compaction_time="30000" />
      <UNICAST3 conn_close_timeout="5000" xmit_interval="200" xmit_table_num_rows=
"50"
                xmit_table_msgs_per_row="1024" xmit_table_max_compaction_time=
"30000" />
      <pbcast.STABLE desired_avg_gossip="2000" max_bytes="1M" />
      <pbcast.GMS print_local_addr="false" join_timeout=
"${jgroups.join_timeout:2000}" />
      <UFC max_credits="4m" min_threshold="0.40" />
      <MFC max_credits="4m" min_threshold="0.40" />
      <FRAG3 />
    </stack>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Uses "prod" for cluster transport. -->
    <transport cluster="${infinispan.cluster.name}"
          stack="prod"
          node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

11.7. Using external JGroups stacks

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

Procedure

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

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

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

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

11.8. Encrypting cluster transport

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

11.8.1. Securing cluster transport with TLS identities

Add SSL/TLS identities to a Infinispan Server security realm and use them to secure cluster transport. Nodes in the Infinispan Server cluster then exchange SSL/TLS certificates to encrypt JGroups messages, including RELAY messages if you configure cross-site replication.

Prerequisites

- Install a Infinispan Server cluster.

Procedure

1. Create a TLS keystore that contains a single certificate to identify Infinispan Server.

   You can also use a PEM file if it contains a private key in PKCS#1 or PKCS#8 format, a certificate, and has an empty password: `password=""`.

   If the certificate in the keystore is not signed by a public certificate authority (CA) then you must also create a trust store that contains either the signing certificate or the public key.

2. Add the keystore to the `$ISPN_HOME/server/conf` directory.
3. Add the keystore to a new security realm in your Infinispan Server configuration.

   You should create dedicated keystores and security realms so that Infinispan Server endpoints do not use the same security realm as cluster transport.

```
<server xmlns="urn:infinispan:server:13.0">
  <security>
    <security-realms>
      <security-realm name="cluster-transport">
        <server-identities>
          <ssl>
            <!-- Adds a keystore that contains a certificate that provides SSL/TLS
identity to encrypt cluster transport. -->
            <keystore path="server.pfx"
                      relative-to="infinispan.server.config.path"
                      password="secret"
                      alias="server"/>
          </ssl>
        </server-identities>
      </security-realm>
    </security-realms>
  </security>
</server>
```

4. Configure cluster transport to use the security realm by specifying the name of the security realm with the `server:security-realm` attribute.

```
<infinispan>
  <cache-container>
    <transport server:security-realm="cluster-transport"/>
  </cache-container>
</infinispan>
```

Verification

When you start Infinispan Server, the following log message indicates that the cluster is using the security realm for cluster transport:

```
[org.infinispan.SERVER] ISPN080060: SSL Transport using realm <security_realm_name>
```

11.8.2. JGroups encryption protocols

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

Infinispan nodes can obtain secret keys from either:

- The coordinator node (asymmetric encryption).
- A shared keystore (symmetric encryption).

Retrieving secret keys from coordinator nodes

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

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

Asymmetric encryption secures cluster traffic as follows:

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

Retrieving secret keys from shared keystores

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

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

Comparison of asymmetric and symmetric encryption

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

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

11.8.3. Securing cluster transport with asymmetric encryption

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

Procedure

1. Create a keystore with certificate chains that enables Infinispan to verify node identity.

2. Place the keystore on the classpath for each node in the cluster.

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

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

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

Verification

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

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

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

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

Additional resources

- JGroups 4 Manual
- JGroups 4.2 Schema

11.8.4. Securing cluster transport with symmetric encryption

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

Procedure

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

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

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

Verification

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

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

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

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

Additional resources

- JGroups 4 Manual
- JGroups 4.2 Schema

11.9. TCP and UDP ports for cluster traffic

Infinispan uses the following ports for cluster transport messages:

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

Cross-site replication

Infinispan uses the following ports for the JGroups RELAY2 protocol:

`7900`
For Infinispan clusters running on Kubernetes.

`7800`
If using UDP for traffic between nodes and TCP for traffic between clusters.

`7801`
If using TCP for traffic between nodes and TCP for traffic between clusters.

Chapter 12. Creating remote caches

When you create remote caches at runtime, Infinispan Server synchronizes your configuration across the cluster so that all nodes have a copy. For this reason you should always create remote caches dynamically with the following mechanisms:

- Infinispan Console
- Infinispan Command Line Interface (CLI)
- Hot Rod or HTTP clients

12.1. Default Cache Manager

Infinispan Server provides a default Cache Manager that controls the lifecycle of remote caches. Starting Infinispan Server automatically instantiates the Cache Manager so you can create and delete remote caches and other resources like Protobuf schema.

After you start Infinispan Server and add user credentials, you can view details about the Cache Manager and get cluster information from Infinispan Console.

- Open `127.0.0.1:11222` in any browser.

You can also get information about the Cache Manager through the Command Line Interface (CLI) or REST API:

CLI

Run the `describe` command in the default container.

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

REST

Open `127.0.0.1:11222/rest/v2/cache-managers/default/` in any browser.

Default Cache Manager configuration

XML

```
<infinispan>
  <!-- Creates a Cache Manager named "default" and enables metrics. -->
  <cache-container name="default"
                   statistics="true">
     <!-- Adds cluster transport that uses the default JGroups TCP stack. -->
     <transport cluster="${infinispan.cluster.name:cluster}"
                stack="${infinispan.cluster.stack:tcp}"
                node-name="${infinispan.node.name:}"/>
     <!-- Requires user permission to access caches and perform operations. -->
     <security>
        <authorization/>
     </security>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "jgroups" : {
      "transport" : "org.infinispan.remoting.transport.jgroups.JGroupsTransport"
    },
    "cache-container" : {
      "name" : "default",
      "statistics" : "true",
      "transport" : {
        "cluster" : "cluster",
        "node-name" : "",
        "stack" : "tcp"
      },
      "security" : {
        "authorization" : {}
      }
    }
  }
}
```

YAML

```yaml
infinispan:
  jgroups:
    transport: "org.infinispan.remoting.transport.jgroups.JGroupsTransport"
  cacheContainer:
    name: "default"
    statistics: "true"
    transport:
      cluster: "cluster"
      nodeName: ""
      stack: "tcp"
    security:
      authorization: ~
```

12.2. Creating caches with Infinispan Console

Use Infinispan Console to create remote caches in an intuitive visual interface from any web browser.

Prerequisites

- Create a Infinispan user with `admin` permissions.
- Start at least one Infinispan Server instance.
- Have a Infinispan cache configuration.

Procedure

1. Open `127.0.0.1:11222/console/` in any browser.
2. Select **Create Cache** and follow the steps as Infinispan Console guides you through the process.

12.3. Creating remote caches with the Infinispan CLI

Use the Infinispan Command Line Interface (CLI) to add remote caches on Infinispan Server.

Prerequisites

- Create a Infinispan user with `admin` permissions.
- Start at least one Infinispan Server instance.
- Have a Infinispan cache configuration.

Procedure

1. Start the CLI and enter your credentials when prompted.

   ```
   $ bin/cli.sh
   ```

2. Use the `create cache` command to create remote caches.

For example, create a cache named "mycache" from a file named `mycache.xml` as follows:

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

Verification

1. List all remote caches with the `ls` command.

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

2. View cache configuration with the `describe` command.

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

12.4. Creating remote caches from Hot Rod clients

Use the Infinispan Hot Rod API to create remote caches on Infinispan Server from Java, C++, .NET/C#, JS clients and more.

This procedure shows you how to use Hot Rod Java clients that create remote caches on first access. You can find code examples for other Hot Rod clients in the Infinispan Tutorials.

Prerequisites

- Create a Infinispan user with `admin` permissions.
- Start at least one Infinispan Server instance.
- Have a Infinispan cache configuration.

Procedure

- Invoke the `remoteCache()` method as part of your the `ConfigurationBuilder`.
- Set the `configuration` or `configuration_uri` properties in the `hotrod-client.properties` file on your classpath.

ConfigurationBuilder

```
File file = new File("path/to/infinispan.xml")
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.remoteCache("another-cache")
       .configuration("<distributed-cache name=\"another-cache\"/>");
builder.remoteCache("my.other.cache")
       .configurationURI(file.toURI());
```

hotrod-client.properties

```
infinispan.client.hotrod.cache.another-cache.configuration=<distributed-cache
name=\"another-cache\"/>
infinispan.client.hotrod.cache.[my.other.cache].configuration_uri=file:///path/to/infi
nispan.xml
```

If the name of your remote cache contains the `.` character, you must enclose it in square brackets when using `hotrod-client.properties` files.

Additional resources

- Hot Rod Client Configuration
- org.infinispan.client.hotrod.configuration.RemoteCacheConfigurationBuilder

12.5. Creating remote caches with the REST API

Use the Infinispan REST API to create remote caches on Infinispan Server from any suitable HTTP client.

Prerequisites

- Create a Infinispan user with `admin` permissions.
- Start at least one Infinispan Server instance.
- Have a Infinispan cache configuration.

Procedure

- Invoke `POST` requests to `/rest/v2/caches/<cache_name>` with cache configuration in the payload.

Additional resources

- Creating and Managing Caches with the REST API

Chapter 13. Running scripts and tasks on Infinispan Server

Add tasks and scripts to Infinispan Server deployments for remote execution from the Command Line Interface (CLI) and Hot Rod or REST clients. You can implement tasks as custom Java classes or define scripts in languages such as JavaScript.

13.1. Adding tasks to Infinispan Server deployments

Add your custom server task classes to Infinispan Server.

Prerequisites

- Stop Infinispan Server if it is running.

  Infinispan Server does not support runtime deployment of custom classes.

Procedure

1. Add a `META-INF/services/org.infinispan.tasks.ServerTask` file that contains the fully qualified names of server tasks, for example:

   ```
   example.HelloTask
   ```

2. Package your server task implementation in a JAR file.
3. Copy the JAR file to the `$ISPN_HOME/server/lib` directory of your Infinispan Server installation.
4. Add your classes to the deserialization allow list in your Infinispan configuration. Alternatively set the allow list using system properties.

Reference

- Adding Java Classes to Deserialization Allow Lists
- Infinispan 13.0 Configuration Schema

13.1.1. Infinispan 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:

```java
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

13.2. Adding scripts to Infinispan Server deployments

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

Prerequisites

Infinispan Server stores scripts in the `___script_cache` cache. If you enable cache authorization, users must have `CREATE` permissions to add to `___script_cache`.

Assign users the `deployer` role at minimum if you use default authorization settings.

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
   ```

13.2.1. Infinispan Server scripts

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

Hello world

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 Server uses when running scripts.

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

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

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

Table 1. Metadata Properties

Property	Description
mode	Defines the execution 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.

Property	Description
`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.

13.2.2. 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

13.3. Running scripts and tasks

Use the command line interface to run tasks and scripts on Infinispan Server deployments. Alternatively you can execute scripts and tasks from Hot Rod clients.

Prerequisites

- Add scripts or tasks to Infinispan Server.

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 `multiplier.js` and specify two parameters:

     ```
     [//containers/default]> task exec multiplier.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"]
     ```

Programmatic execution

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

Script execution

```
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);
```

Task execution

```
// 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);
```

Additional resources

- org.infinispan.client.hotrod.RemoteCache

Chapter 14. Configuring Infinispan Server logging

Infinispan Server 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.

14.1. Infinispan Server log files

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

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

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

14.1.1. Configuring Infinispan Server logs

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

Procedure

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

Additional resources

- Apache Log4j manual

14.1.2. Log levels

Log levels indicate the nature and severity of messages.

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

Log level	Description
`WARN`	Events that can lead to error or degrade performance.
`ERROR`	Error conditions that might prevent operations or activities 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.

14.1.3. Infinispan logging 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.

14.1.4. Log appenders

Log appenders define how Infinispan Server 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.

14.1.5. Log pattern formatters

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 .

14.1.6. Enabling the JSON log handler

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

Prerequisites

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

Procedure

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

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

3. Optionally configure the JSON appender and JSON layout as required.
4. Save and close `log4j2.xml`.

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

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

Additional resources

- RollingFileAppender
- JSONLayout

14.2. Access logs

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

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

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

14.2.1. Enabling access logs

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

Procedure

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

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

14.2.2. Access log properties

The default format for access logs is as follows:

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

The preceding format creates log entries such as the following:

`127.0.0.1 - [DD/MM/YYYY:HH:MM:SS +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}.

14.3. Audit logs

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

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

14.3.1. Enabling audit logging

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

Procedure

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

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

14.3.2. Configuring audit logging appenders

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

Procedure

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

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

3. Add the desired logging appender for audit messages.

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

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

4. Save and close `log4j2.xml`.

Additional resources

- Log4j Appenders

14.3.3. Using custom audit logging implementations

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

Prerequisites

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

Procedure

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

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

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

Additional resources

- `org.infinispan.security.AuditLogger`

Chapter 15. Performing rolling upgrades for Infinispan Server clusters

Perform rolling upgrades of your Infinispan clusters to change between versions without downtime or data loss and migrate data over the Hot Rod protocol.

15.1. Setting up target Infinispan clusters

Create a cluster that uses the Infinispan version to which you plan to upgrade and then connect the source cluster to the target cluster using a remote cache store.

Prerequisites

- Install Infinispan Server nodes with the desired version for your target cluster.

> 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 port offsets to separate the target and source clusters.

Procedure

1. Create a remote cache store configuration, in JSON format, that allows the target cluster to connect to the source cluster.

   Remote cache stores on the target cluster use the Hot Rod protocol to retrieve data from the source cluster.

```
{
  "remote-store": {
    "cache": "myCache",
    "shared": true,
    "raw-values": true,
    "security": {
      "authentication": {
        "digest": {
          "username": "username",
          "password": "changeme",
          "realm": "default"
        }
      }
    },
    "remote-server": [
      {
        "host": "127.0.0.1",
        "port": 12222
      }
    ]
  }
}
```

2. Use the Infinispan Command Line Interface (CLI) or REST API to add the remote cache store configuration to the target cluster so it can connect to the source cluster.
   - CLI: Use the `migrate cluster connect` command on the target cluster.

     ```
     [//containers/default]> migrate cluster connect -c myCache --file=remote
     -store.json
     ```

   - REST API: Invoke a POST request that includes the remote store configuration in the payload with the `rolling-upgrade/source-connection` method.

     ```
     POST /v2/caches/myCache/rolling-upgrade/source-connection
     ```

3. Repeat the preceding step for each cache that you want to migrate.
4. 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.

Additional resources

- Remote cache store configuration schema

15.2. Synchronizing data to target clusters

When you set up a target Infinispan cluster and connect it to a source cluster, the target cluster can handle client requests using a remote cache store and load data on demand. To completely migrate data to the target cluster, so you can decommission the source cluster, you can synchronize data. 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 that you want to migrate to the target cluster.

Prerequisites

- Set up a target cluster with the appropriate Infinispan version.

Procedure

1. Start synchronizing each cache that you want to migrate to the target cluster with the Infinispan Command Line Interface (CLI) or REST API.
   - CLI: Use the `migrate cluster synchronize` command.

     ```
     [//containers/default]> migrate cluster synchronize -c myCache
     ```

   - REST API: Use the `?action=sync-data` parameter with a POST request.

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

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

2. Disconnect each node in the target cluster from the source cluster.
   - CLI: Use the `migrate cluster disconnect` command.

     ```
     [//containers/default]> migrate cluster disconnect -c myCache
     ```

   - REST API: Invoke a DELETE request.

     ```
     DELETE /v2/caches/myCache/rolling-upgrade/source-connection
     ```

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 16. Patching Infinispan Server installations

Install and manage patches for Infinispan Server installations.

You can apply patches to multiple Infinispan Server installations with different versions to upgrade to a desired target version. However, patches do not take effect if Infinispan Server is running. 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.

16.1. 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 Server to modify your 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.

16.2. Creating Infinispan Server patches

You can create patches for Infinispan Server from an existing installation.

You can create patches for Infinispan Server starting from version 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
```

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

     You can apply patches to Infinispan Server installations that match the `$source_version` only.

16.3. Applying patches to Infinispan Server installations

Upgrade or downgrade Infinispan Server installation to a Infinispan version by installing patches.

Prerequisites

- Create a server patch for the target version.

Procedure

1. Navigate to `$ISPN_HOME` for the Infinispan Server installation you want to patch.
2. Stop the Infinispan 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
```

16.4. Rolling back patches

Remove patches from Infinispan Server 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 17. Troubleshooting Infinispan Server deployments

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

17.1. Getting diagnostic reports from Infinispan Server

Infinispan Server provides aggregated reports in `tar.gz` archives that contain diagnostic information about server instances and host systems. 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 Server.
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.

17.2. Changing Infinispan Server logging configuration at runtime

Modify the logging configuration for Infinispan Server 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 Server.
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
```

17.3. Gathering resource statistics from the CLI

You can inspect server-collected statistics for some Infinispan Server resources with 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
}
```

17.4. Accessing cluster health 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.

17.5. Accessing cluster health 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.