Red Hat Data Grid 8.3

Embedding Data Grid in Java Applications

Create embedded caches with Data Grid

Last Updated: 2022-06-07

Red Hat Data Grid 8.3 Embedding Data Grid in Java Applications

Create embedded caches with Data Grid

Abstract

Add Data Grid to Java projects and use embedded caches with your applications.

Table of Contents

RED HAT DATA GRID

Data Grid is a high-performance, distributed in-memory data store.

Schemaless data structure
: Flexibility to store different objects as key-value pairs.

Grid-based data storage
: Designed to distribute and replicate data across clusters.

Elastic scaling
: Dynamically adjust the number of nodes to meet demand without service disruption.

Data interoperability
: Store, retrieve, and query data in the grid from different endpoints.

DATA GRID DOCUMENTATION

Documentation for Data Grid is available on the Red Hat customer portal.

- Data Grid 8.3 Documentation
- Data Grid 8.3 Component Details
- Supported Configurations for Data Grid 8.3
- Data Grid 8 Feature Support
- Data Grid Deprecated Features and Functionality

DATA GRID DOWNLOADS

Access the Data Grid Software Downloads on the Red Hat customer portal.

NOTE

You must have a Red Hat account to access and download Data Grid software.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our technical content and encourage you to tell us what you think. If you'd like to add comments, provide insights, correct a typo, or even ask a question, you can do so directly in the documentation.

> **NOTE**
>
> You must have a Red Hat account and be logged in to the customer portal.

To submit documentation feedback from the customer portal, do the following:

1. Select the **Multi-page HTML** format.
2. Click the **Feedback** button at the top-right of the document.
3. Highlight the section of text where you want to provide feedback.
4. Click the **Add Feedback** dialog next to your highlighted text.
5. Enter your feedback in the text box on the right of the page and then click **Submit**.

We automatically create a tracking issue each time you submit feedback. Open the link that is displayed after you click **Submit** and start watching the issue or add more comments.

Thank you for the valuable feedback.

CHAPTER 1. CONFIGURING THE DATA GRID MAVEN REPOSITORY

Data Grid Java distributions are available from Maven.

You can download the Data Grid Maven repository from the customer portal or pull Data Grid dependencies from the public Red Hat Enterprise Maven repository.

1.1. DOWNLOADING THE DATA GRID MAVEN REPOSITORY

Download and install the Data Grid Maven repository to a local file system, Apache HTTP server, or Maven repository manager if you do not want to use the public Red Hat Enterprise Maven repository.

Procedure

1. Log in to the Red Hat customer portal.
2. Navigate to the Software Downloads for Data Grid.
3. Download the Red Hat Data Grid 8.3 Maven Repository.
4. Extract the archived Maven repository to your local file system.
5. Open the **README.md** file and follow the appropriate installation instructions.

1.2. ADDING RED HAT MAVEN REPOSITORIES

Include the Red Hat GA repository in your Maven build environment to get Data Grid artifacts and dependencies.

Procedure

- Add the Red Hat GA repository to your Maven settings file, typically **~/.m2/settings.xml**, or directly in the **pom.xml** file of your project.

```
<repositories>
  <repository>
    <id>redhat-ga-repository</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>redhat-ga-repository</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </pluginRepository>
</pluginRepositories>
```

Reference

- Red Hat Enterprise Maven Repository

1.3. CONFIGURING YOUR DATA GRID POM

Maven uses configuration files called Project Object Model (POM) files to define projects and manage builds. POM files are in XML format and describe the module and component dependencies, build order, and targets for the resulting project packaging and output.

Procedure

1. Open your project **pom.xml** for editing.
2. Define the **version.infinispan** property with the correct Data Grid version.
3. Include the **infinispan-bom** in a **dependencyManagement** section.
   The Bill Of Materials (BOM) controls dependency versions, which avoids version conflicts and means you do not need to set the version for each Data Grid artifact you add as a dependency to your project.
4. Save and close **pom.xml**.

The following example shows the Data Grid version and BOM:

```
<properties>
  <version.infinispan>13.0.10.Final-redhat-00001</version.infinispan>
</properties>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-bom</artifactId>
      <version>${version.infinispan}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Next Steps

Add Data Grid artifacts as dependencies to your **pom.xml** as required.

CHAPTER 2. CREATING EMBEDDED CACHES

Data Grid provides an **EmbeddedCacheManager** API that lets you control both the Cache Manager and embedded cache lifecycles programmatically.

2.1. ADDING DATA GRID TO YOUR PROJECT

Add Data Grid to your project to create embedded caches in your applications.

Prerequisites

- Configure your project to get Data Grid artifacts from the Maven repository.

Procedure

- Add the **infinispan-core** artifact as a dependency in your **pom.xml** as follows:

```
<dependencies>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-core</artifactId>
  </dependency>
</dependencies>
```

2.2. CONFIGURING EMBEDDED CACHES

Data Grid provides a **GlobalConfigurationBuilder** API that controls the cache manager and a **ConfigurationBuilder** API that configures embedded caches.

Prerequisites

- Add the **infinispan-core** artifact as a dependency in your **pom.xml**.

Procedure

1. Initialize the default cache manager so you can add embedded caches.
2. Add at least one embedded cache with the **ConfigurationBuilder** API.
3. Invoke the **getOrCreateCache()** method that either creates embedded caches on all nodes in the cluster or returns caches that already exist.

```
// Set up a clustered cache manager.
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();
// Initialize the default cache manager.
DefaultCacheManager cacheManager = new DefaultCacheManager(global.build());
// Create a distributed cache with synchronous replication.
ConfigurationBuilder builder = new ConfigurationBuilder();
                    builder.clustering().cacheMode(CacheMode.DIST_SYNC);
// Obtain a volatile cache.
Cache<String, String> cache =
cacheManager.administration().withFlags(CacheContainerAdmin.AdminFlag.VOLATILE).getOrCreateC
ache("myCache", builder.build());
```

Additional resources

- EmbeddedCacheManager
- EmbeddedCacheManager Configuration
- org.infinispan.configuration.global.GlobalConfiguration
- org.infinispan.configuration.cache.ConfigurationBuilder

CHAPTER 3. ENABLING AND CONFIGURING DATA GRID STATISTICS AND JMX MONITORING

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

3.1. ENABLING STATISTICS IN EMBEDDED CACHES

Configure Data Grid to export statistics for the cache manager and embedded caches.

Procedure

1. Open your Data Grid configuration for editing.
2. Add the **statistics="true"** attribute or the **.statistics(true)** method.
3. Save and close your Data Grid configuration.

Embedded cache statistics

XML

```
<infinispan>
  <cache-container statistics="true">
    <distributed-cache statistics="true"/>
    <replicated-cache statistics="true"/>
  </cache-container>
</infinispan>
```

GlobalConfigurationBuilder

```
GlobalConfigurationBuilder global =
GlobalConfigurationBuilder.defaultClusteredBuilder().cacheContainer().statistics(true);
DefaultCacheManager cacheManager = new DefaultCacheManager(global.build());

Configuration builder = new ConfigurationBuilder();
builder.statistics().enable();
```

3.2. CONFIGURING DATA GRID METRICS

Data Grid 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, Data Grid generates gauges when you enable statistics but you can also configure it to generate histograms.

Procedure

1. Open your Data Grid 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

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

JSON

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

YAML

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

GlobalConfigurationBuilder

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

Verification

For embedded caches, you must add the necessary MicroProfile API and provider JARs to your classpath to export Data Grid metrics.

Additional resources

- Eclipse MicroProfile Metrics

3.3. REGISTERING JMX MBEANS

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

Procedure

1. Open your Data Grid 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

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

GlobalConfigurationBuilder

```
GlobalConfiguration global = GlobalConfigurationBuilder.defaultClusteredBuilder()
   .jmx().enable()
   .domain("org.mydomain");
```

3.3.1. Enabling JMX remote ports

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

Procedure

- Pass the following system properties to Data Grid at startup:

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
```

3.3.2. Data Grid MBeans

Data Grid 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 Data Grid cache and cluster health statistics.

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

Additional resources

- Data Grid JMX Components

3.3.3. Registering MBeans in custom MBean servers

Data Grid 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 Data Grid to register JMX MBeans.

Procedure

1. Open your Data Grid 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

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

JSON

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

YAML

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

GlobalConfigurationBuilder

```
GlobalConfiguration global = GlobalConfigurationBuilder.defaultClusteredBuilder()
  .jmx().enable()
  .domain("org.mydomain")
  .mBeanServerLookup(new com.acme.MyMBeanServerLookup());
```

CHAPTER 4. SETTING UP DATA GRID CLUSTER TRANSPORT

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

4.1. DEFAULT JGROUPS STACKS

Data Grid provides default JGroups stack files, **default-jgroups-*.xml**, in the **default-configs** directory inside the **infinispan-core-13.0.10.Final-redhat-00001.jar** file.

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

4.2. CLUSTER DISCOVERY PROTOCOLS

Data Grid 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 Data Grid 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 Data Grid clusters.
  For instance the DNS_PING protocol performs discovery through DNS server records.

NOTE

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

Additional resources

- JGroups Discovery Protocols
- JGroups cluster transport configuration for Data Grid 8.x (Red Hat knowledgebase article)

4.2.1. 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 Data Grid 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

4.2.2. 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 Data Grid 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[port2]}"
```

```
        port_range="0"
        num_initial_members="3"/>
```

Additional resources

- JGroups TCPPING

4.2.3. MPING

MPING uses IP multicast to discover the initial membership of Data Grid 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

4.2.4. TCPGOSSIP

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

You inject the address (**IP:PORT**) of the Gossip router into Data Grid 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

4.2.5. JDBC_PING

JDBC_PING uses shared databases to store information about Data Grid 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 Data Grid cluster on the network. When nodes leave Data Grid 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"/>
```

> **IMPORTANT**
>
> Add the appropriate JDBC driver to the classpath so Data Grid can use JDBC_PING.

Additional resources

- JDBC_PING
- JDBC_PING Wiki

4.2.6. DNS_PING

JGroups DNS_PING queries DNS servers to discover Data Grid 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)

4.2.7. Cloud discovery protocols

Data Grid 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
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 Data Grid.

Procedure

- Add the artifact dependencies to your project **pom.xml**.

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

4.3. USING THE DEFAULT JGROUPS STACKS

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

Data Grid 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 **addProperty()** method to set the JGroups stack file:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("qa-cluster")
        //Uses the default-jgroups-udp.xml stack for cluster transport.
        .addProperty("configurationFile", "default-jgroups-udp.xml")
        .build();
```

Verification

Data Grid logs the following message to indicate which stack it uses:

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

Additional resources

- JGroups cluster transport configuration for Data Grid 8.x (Red Hat knowledgebase article)

4.4. CUSTOMIZING JGROUPS STACKS

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

Data Grid 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 Data Grid logs to ensure it uses the stack.

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

Reference

- JGroups cluster transport configuration for Data Grid 8.x (Red Hat knowledgebase article)

4.4.1. 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.

4.5. USING JGROUPS SYSTEM PROPERTIES

Pass system properties to Data Grid 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:

```
java -cp ... -Djgroups.bind.port=1234 -Djgroups.bind.address=192.0.2.0
```

NOTE

When you embed Data Grid clusters in clustered Red Hat JBoss EAP applications, JGroups system properties can clash or override each other.

For example, you do not set a unique bind address for either your Data Grid cluster or your Red Hat JBoss EAP application. In this case both Data Grid and your Red Hat JBoss EAP application use the JGroups default property and attempt to form clusters using the same bind address.

4.5.1. 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
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

System Property	Description	Default Value	Required/ 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

4.5.2. System properties for cloud discovery protocols

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

4.5.2.1. 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

4.5.2.2. 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

4.5.2.3. 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

4.5.2.4. OpenShift

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

4.6. 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.

```
<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>
```

4.7. USING EXTERNAL JGROUPS STACKS

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

Procedure

1. Put custom JGroups stack files on the application classpath.
   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>
```

You can also use the **addProperty()** method in the **TransportConfigurationBuilder** class to specify a custom JGroups stack file as follows:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("prod-cluster")
        //Uses a custom JGroups stack for cluster transport.
        .addProperty("configurationFile", "my-jgroups-udp.xml")
        .build();
```

In this example, **my-jgroups-udp.xml** references a UDP stack with custom properties such as the following:

Custom UDP stack example

```
<config xmlns="urn:org:jgroups"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:org:jgroups http://www.jgroups.org/schema/jgroups-4.2.xsd">
    <UDP bind_addr="${jgroups.bind_addr:127.0.0.1}"
         mcast_addr="${jgroups.udp.mcast_addr:192.0.2.0}"
         mcast_port="${jgroups.udp.mcast_port:46655}"
         tos="8"
         ucast_recv_buf_size="20000000"
         ucast_send_buf_size="640000"
         mcast_recv_buf_size="25000000"
         mcast_send_buf_size="640000"
         max_bundle_size="64000"
         ip_ttl="${jgroups.udp.ip_ttl:2}"
         enable_diagnostics="false"
         thread_naming_pattern="pl"
         thread_pool.enabled="true"
         thread_pool.min_threads="2"
         thread_pool.max_threads="30"
         thread_pool.keep_alive_time="5000" />
    <!-- Other JGroups stack configuration goes here. -->
</config>
```

Additional resources

- org.infinispan.configuration.global.TransportConfigurationBuilder

4.8. USING CUSTOM JCHANNELS

Construct custom JGroups JChannels as in the following example:

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
```

```
JChannel jchannel = new JChannel();
// Configure the jchannel as needed.
JGroupsTransport transport = new JGroupsTransport(jchannel);
global.transport().transport(transport);
new DefaultCacheManager(global.build());
```

> **NOTE**
>
> Data Grid cannot use custom JChannels that are already connected.

Additional resources

- JGroups JChannel

4.9. ENCRYPTING CLUSTER TRANSPORT

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

4.9.1. JGroups encryption protocols

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

Data Grid 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 Data Grid configuration. This allows Data Grid clusters to generate and distribute secret keys.

> **IMPORTANT**
>
> 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 Data Grid 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 Data Grid configuration. This allows Data Grid clusters to obtain secret keys from keystores that you provide.

1. Nodes install the secret key from a keystore on the Data Grid 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. Data Grid 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.

4.9.2. Securing cluster transport with asymmetric encryption

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

Procedure

1. Create a keystore with certificate chains that enables Data Grid to verify node identity.
2. Place the keystore on the classpath for each node in the cluster.
   For Data Grid Server, you put the keystore in the $RHDG_HOME directory.
3. Add the **SSL_KEY_EXCHANGE** and **ASYM_ENCRYPT** protocols to a JGroups stack in your Data Grid 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 Data Grid 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>
```

Data Grid 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 Data Grid 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

4.9.3. Securing cluster transport with symmetric encryption

Configure Data Grid 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 Data Grid Server, you put the keystore in the $RHDG_HOME directory.
3. Add the **SYM_ENCRYPT** protocol to a JGroups stack in your Data Grid configuration.

```
<infinispan>
  <jgroups>
    <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the default TCP stack. -->
```

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

Verification

When you start your Data Grid 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>
```

Data Grid 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 Data Grid 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

4.10. TCP AND UDP PORTS FOR CLUSTER TRAFFIC

Data Grid 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

Data Grid uses the following ports for the JGroups RELAY2 protocol:

7900
: For Data Grid clusters running on OpenShift.

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.