Red Hat Data Grid 8.1

Migrating to Data Grid 8

Data Grid Migration Guide

Last Updated: 2023-11-23

Red Hat Data Grid 8.1 Migrating to Data Grid 8

Data Grid Migration Guide

Legal Notice

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide provides detailed information to help you successfully migrate to Red Hat Data Grid 8 from previous versions.

Table of Contents

RED HAT DATA GRID 5

DATA GRID DOCUMENTATION 6

DATA GRID DOWNLOADS 7

MAKING OPEN SOURCE MORE INCLUSIVE 8

CHAPTER 1. DATA GRID 8 9
1.1. MIGRATION TO DATA GRID 8 9
1.2. MIGRATION PATHS 9
1.3. COMPONENT DOWNLOADS 9
Maven repository 10
Data Grid Server 10
Modules for JBoss EAP 11
Tomcat session client 11
Hot Rod Node.js client 11
Source code 11

CHAPTER 2. MIGRATING DATA GRID SERVER DEPLOYMENTS 12
2.1. DATA GRID SERVER 8 12
2.2. DATA GRID SERVER CONFIGURATION 12
Dynamic configuration 12
Static configuration 13
Cache container configuration 13
Server configuration 14
2.3. DATA GRID SERVER ENDPOINT AND NETWORK CONFIGURATION 15
2.3.1. Interfaces 16
Data Grid Server 7.x network interface configuration 16
Data Grid Server 8 network interface configuration 16
2.3.2. Socket bindings 16
Data Grid Server 7.x socket binding configuration 16
Data Grid Server 8 single port configuration 16
2.3.3. Endpoints 17
Data Grid Server 7.x endpoint subsystem 17
Data Grid Server 8 endpoint configuration 17
2.4. DATA GRID SERVER SECURITY 17
2.4.1. Security realms 17
Supported security realms 18
2.4.2. Server identities 18
2.4.3. Endpoint authentication mechanisms 19
Hot Rod SASL authentication mechanisms 19
HTTP (REST) authentication mechanisms 20
2.4.4. Authenticating EAP applications 21
2.4.5. Logging 21
Access logs 21
2.5. SEPARATING DATA GRID SERVER ENDPOINTS 21
2.6. DATA GRID SERVER SHARED DATASOURCES 23
2.7. DATA GRID SERVER JMX AND METRICS 24
2.8. DATA GRID SERVER CHEATSHEET 25
Starting server instances 25
Starting the CLI 25

Creating users 25
Stopping server instances 25
Listing available command options 25
7.x to 8 reference 26

CHAPTER 3. MIGRATING DATA GRID CONFIGURATION 27
3.1. DATA GRID CACHE CONFIGURATION 27
3.1.1. Cache encoding 28
3.1.2. Cache health status 28
3.1.3. Changes to the Data Grid 8.1 configuration schema 29
New and modified elements and attributes 29
Deprecated elements and attributes 29
Removed elements and attributes 30
3.2. EVICTION CONFIGURATION 30
3.2.1. Storage types 30
Changes in Data Grid 8 30
Object storage in Data Grid 8 31
Off-heap storage in Data Grid 8 31
Off-heap address count 31
Binary storage in Data Grid 8 31
3.2.2. Eviction threshold 31
Eviction based on total number of entries 32
Eviction based on maximum amount of memory 32
3.2.3. Eviction strategies 32
Eviction algorithms 33
3.2.4. Eviction configuration comparison 33
Object storage and evict on number of entries 33
7.2 to 8.0 33
8.1 33
Object storage and evict on amount of memory 33
7.2 to 8.0 33
8.1 33
Binary storage and evict on number of entries 33
7.2 to 8.0 33
8.1 34
Binary storage and evict on amount of memory 34
7.2 to 8.0 34
8.1 34
Off-heap storage and evict on number of entries 34
7.2 to 8.0 34
8.1 34
Off-heap storage and evict on amount of memory 34
7.2 to 8.0 34
8.1 34
3.3. EXPIRATION CONFIGURATION 34
3.4. PERSISTENT CACHE STORES 35
Persistence SPI 35
Custom cache stores 35
Segmented cache stores 36
Single file cache stores 36
JDBC cache stores 36
JDBC connection factories 36
Segmentation 36

Write-behind 37
Removed cache stores and loaders 37
Cache store migrator 37
3.5. DATA GRID CLUSTER TRANSPORT 37
3.5.1. Transport security 38
3.6. DATA GRID AUTHORIZATION 39
Cache manager authorization 39
Implicit cache authorization 39

CHAPTER 4. MIGRATING TO DATA GRID 8 APIS 40
4.1. REST API 40
4.2. QUERY API 40
Indexing Data Grid caches 40
Enabling indexing in Data Grid 8 41
Querying values in caches 41

CHAPTER 5. MIGRATING APPLICATIONS TO DATA GRID 8 43
5.1. MARSHALLING IN DATA GRID 8 43
5.1.1. ProtoStream marshalling 43
Marshalling with Data Grid Server 43
Cache stores and ProtoStream 43
5.1.2. Alternative marshaller implementations 43
JBoss marshalling 44
5.2. MIGRATING APPLICATIONS TO THE AUTOPROTOSCHEMABUILDER ANNOTATION 44
5.2.1. Basic MessageMarshaller implementation 45
Migrated to the AutoProtoSchemaBuilder annotation 46
Important observations 47
5.2.2. MessageMarshaller implementation with custom types 47
Migrated code with an adapter class 48
Migrated code without an adapter class 50

CHAPTER 6. MIGRATING DATA GRID CLUSTERS ON RED HAT OPENSHIFT 52
6.1. DATA GRID ON OPENSHIFT 52
Creating Data Grid Services 52
Creating Cache service nodes in 7.3 52
Creating Data Grid service nodes in 7.3 52
Creating services in Data Grid 8 52
6.1.1. Container storage 53
6.1.2. Data Grid CLI 53
6.1.3. Data Grid console 53
6.1.4. Customizing Data Grid 53
6.1.5. Deployment configuration templates 53

CHAPTER 7. MIGRATING DATA BETWEEN CACHE STORES 54
7.1. CACHE STORE MIGRATOR 54
7.2. GETTING THE STORE MIGRATOR 54
7.3. CONFIGURING THE STORE MIGRATOR 55
7.3.1. Store Migrator Properties 56
7.4. MIGRATING CACHE STORES 60

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.

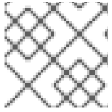
DATA GRID DOCUMENTATION

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

- Data Grid 8.1 Documentation
- Data Grid 8.1 Component Details
- Supported Configurations for Data Grid 8.1
- 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 .

CHAPTER 1. DATA GRID 8

Start the journey of migration to Data Grid 8 with a brief overview and a look at some of the basics.

1.1. MIGRATION TO DATA GRID 8

Data Grid 8 introduces significant changes from previous Data Grid versions, including a whole new architecture for server deployments.

While this makes certain aspects of migration more challenging for existing environments, the Data Grid team believe that these changes benefit users by reducing deployment complexity and administrative overhead.

In comparison to previous versions, migration to Data Grid 8 means you gain:

- Cloud-native design built for container platforms.
- Lighter memory footprint and less overall resource usage.
- Faster start times.
- Increased security through smaller attack surface.
- Better integration with Red Hat technologies and solutions.

And Data Grid 8 continues to give you the best possible in-memory datastorage capabilities built from tried and trusted, open-source technology.

1.2. MIGRATION PATHS

This documentation focuses on Data Grid 7.3 to Data Grid 8 migration but is still applicable for 7.x versions, starting from 7.0.1.

If you are planning a migration from Data Grid 6, this document might not capture everything you need. You should contact Red Hat support for advice specific to your deployment before migrating.

As always, please let us know if we can help you by improving this documentation.

1.3. COMPONENT DOWNLOADS

To start using Data Grid 8, you either:

- Download components from the Red Hat customer portal if you are installing Data Grid on bare metal or other host environment.
- Create an Data Grid Operator subscription if you are running on OpenShift.

This following information describes the available component downloads for bare metal deployments, which are different to previous versions of Data Grid.

Also see:

- Data Grid on OpenShift Migration
- Data Grid 8 Supported Configurations

Maven repository

Data Grid 8 no longer provides separate downloads from the Red Hat customer portal for the following components:

- Data Grid core libraries to create embedded caches in custom applications, referred to as "Library Mode" in previous versions.
- Hot Rod Java client.
- Utilities such as **StoreMigrator**.

Instead of making these components available as downloads, Data Grid provides Java artifacts through a Maven repository. This change means that you can use Maven to centrally manage dependencies, which provides better control over dependencies across projects.

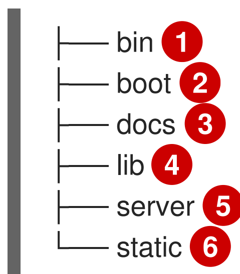
You can download the Data Grid Maven repository from the customer portal or pull Data Grid dependencies from the public Red Hat Enterprise Maven repository. Instructions for both methods are available in the Data Grid documentation.

- Configuring the Data Grid Maven Repository

Data Grid Server

Data Grid Server is distributed as an archive that you can download and extract to host file systems.

The archive distribution contains the following top-level folders:



```
├── bin 1
├── boot 2
├── docs 3
├── lib 4
├── server 5
└── static 6
```

1. Scripts to start and manage Data Grid Server as well as the Data Grid Command Line Interface (CLI).
2. Boot libraries.
3. Resources to help you configure and run Data Grid Server.
4. Run-time libraries for Data Grid Server. Note that this folder is intended for internal code only, not custom code libraries.
5. Root directory for Data Grid Server instances.
6. Static resources for Data Grid Console.

The **server** folder is the root directory for Data Grid Server instances and contains subdirectories for custom code libraries, configuration files, and data.

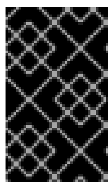
You can find more information about the filesystem and contents of the distributions in the *Data Grid Server Guide*.

- Data Grid Server Filesystem

- Data Grid Server README

Modules for JBoss EAP

You can use the modules for Red Hat JBoss EAP (EAP) to embed Data Grid caching functionality in your EAP applications.



IMPORTANT

In EAP 7.4 applications can directly handle the **infinispan** subsystem without the need to separately install Data Grid modules. After EAP 7.4 GA is released, Data Grid will no longer provide EAP modules for download.

Red Hat still offers support if you want to build and use your own Data Grid modules. However, Red Hat recommends that you use Data Grid APIs directly with EAP 7.4 because modules:

- Cannot use centrally managed Data Grid configuration that is shared across EAP applications. To use modules, you need to store configuration inside the application JAR or WAR.
- Often result in Java classloading issues that require debugging and additional overhead to implement.

You can find more information about the EAP modules that Data Grid provides in the *Data Grid Developer Guide*.

- Data Grid Modules for Red Hat JBoss EAP

Tomcat session client

The Tomcat session client lets you externalize HTTP sessions from JBoss Web Server (JWS) applications to Data Grid via the Apache Tomcat **org.apache.catalina.Manager** interface.

- Externalizing HTTP Sessions from JBoss Web Server (JWS) to Data Grid

Hot Rod Node.js client

The Hot Rod Node.js client is a reference JavaScript implementation for use with Data Grid Server clusters.

- Hot Rod Node.js Client API

Source code

Uncompiled source code for each Data Grid release.

CHAPTER 2. MIGRATING DATA GRID SERVER DEPLOYMENTS

Review the details in this section to plan and prepare a successful migration of Data Grid Server.

2.1. DATA GRID SERVER 8

Data Grid Server 8 is:

- Designed for modern system architectures.
- Built for containerized platforms.
- Optimized for native image compilation with Quarkus.

The transition to a cloud-native architecture means that Data Grid Server 8 is no longer based on Red Hat JBoss Enterprise Application Platform (EAP). Instead Data Grid Server 8 is based on the Netty project's client/server framework.

This change affects migration from previous versions because many of the facilities that integration with EAP provided are no longer relevant to Data Grid 8 or have changed.

For instance, while complexity of server configuration is greatly reduced in comparison to previous releases, you do need to adapt your existing configuration to a new schema. Data Grid 8 also provides more of a convention for server configuration than in previous versions where it was possible to achieve much more granular configuration. Additionally Data Grid Server no longer leverages Domain Mode to centrally manage configuration.

The Data Grid team acknowledge that these configuration changes place additional effort on our customers to migrate their existing clusters to Data Grid 8.

We believe that it is better to use container orchestration platforms, such as Red Hat OpenShift, to provision and administer Data Grid clusters along with automation engines, such as Red Hat Ansible, to manage Data Grid configuration. These technologies offer greater flexibility in that they are more generic and suitable for multiple disparate systems, rather than solutions that are more specific to Data Grid.

In terms of migration to Data Grid 8, it is worth noting that solutions like Red Hat Ansible are helpful with large-scale configuration deployment. However, that tooling might not necessarily aid the actual migration of your existing Data Grid configuration.

2.2. DATA GRID SERVER CONFIGURATION

Data Grid provides a scalable data layer that lets you intelligently and efficiently utilize available computing resources. To achieve this with Data Grid Server deployments, configuration is separated into two layers: dynamic and static.

Dynamic configuration

Dynamic configuration is mutable, changing at runtime as you create caches and add and remove nodes to and from the cluster.

After you deploy your Data Grid Server cluster, you create caches through the Data Grid CLI, Data Grid Console, or Hot Rod and REST endpoints. Data Grid Server permanently stores those caches as part of the cluster state that is distributed across nodes. Each joining node receives the complete cluster state that Data Grid Server automatically synchronizes across all nodes as changes occur.

Static configuration

Static configuration is immutable, remaining unchanged at runtime.

You define static configuration when setting up underlying mechanisms such as cluster transport, authentication and encryption, shared datasources, and so on.

By default Data Grid Server uses **$RHDG_HOME/server/conf/infinispan.xml** for static configuration.

The root element of the configuration is **infinispan** and declares two base schema:

```
<infinispan
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:infinispan:config:11.0 https://infinispan.org/schemas/infinispan-config-11.0.xsd
                        urn:infinispan:server:11.0 https://infinispan.org/schemas/infinispan-server-11.0.xsd"
    xmlns="urn:infinispan:config:11.0"
    xmlns:server="urn:infinispan:server:11.0">
```

- The **urn:infinispan:config** schema validates configuration for core Infinispan capabilities such as the cache container.
- The **urn:infinispan:server** schema validates configuration for Data Grid Server.

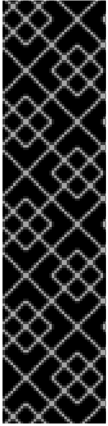
Cache container configuration

You use the **cache-container** element to configure the **CacheManager** interface that provides mechanisms to manage cache lifecycles:

```
<!-- Creates a cache manager named "default" that exports statistics. -->
<cache-container name="default"
                 statistics="true">
  <!-- Defines cluster transport properties, including the cluster name. -->
  <!-- Uses the default TCP stack for inter-cluster communication. -->
  <transport cluster="${infinispan.cluster.name}"
             stack="${infinispan.cluster.stack:tcp}"
             node-name="${infinispan.node.name:}"/>
</cache-container>
```

The **cache-container** element can also hold the following configuration elements:

- **security** for the cache manager.
- **metrics** for MicroProfile compatible metrics.
- **jmx** for JMX monitoring and administration.



IMPORTANT

In previous versions, you could define multiple **cache-container** elements in your Data Grid configuration to expose cache containers on different endpoints.

In Data Grid 8 you must not configure multiple cache containers because the Data Grid CLI and Console can handle only one cache manager per cluster. However you can change the name of the cache container to something more meaningful to your environment than "default", if necessary.

You should use separate Data Grid clusters to achieve multitenancy to ensure that cache managers do not interfere with each other.

Server configuration

You use the **server** element to configure underlying Data Grid Server mechanisms:

```
<server>
  <interfaces>
    <interface name="public"> 1
      <inet-address value="${infinispan.bind.address:127.0.0.1}"/> 2
    </interface>
  </interfaces>

  <socket-bindings default-interface="public" 3
                   port-offset="${infinispan.socket.binding.port-offset:0}"> 4
    <socket-binding name="default" 5
                    port="${infinispan.bind.port:11222}"/> 6
    <socket-binding name="memcached" port="11221"/> 7
  </socket-bindings>

  <security>
    <security-realms> 8
      <security-realm name="default"> 9
        <server-identities> 10
          <ssl>
            <keystore path="application.keystore" 11
                      keystore-password="password"
                      alias="server"
                      key-password="password"
                      generate-self-signed-certificate-host="localhost"/>
          </ssl>
        </server-identities>
        <properties-realm groups-attribute="Roles"> 12
          <user-properties path="users.properties" 13
                           relative-to="infinispan.server.config.path"
                           plain-text="true"/> 14
          <group-properties path="groups.properties" 15
                            relative-to="infinispan.server.config.path" />
        </properties-realm>
      </security-realm>
    </security-realms>
  </security>
```

```
  <endpoints socket-binding="default" security-realm="default" /> (16)

</server>
```

1. Creates an interface named "public" that makes the server available on your network.
2. Uses the **127.0.0.1** loopback address for the public interface.
3. Binds the public interface to the network ports where Data Grid Server endpoints listen for incoming client connections.
4. Specifies an offset of **0** for network ports.
5. Creates a socket binding named "default".
6. Specifies port **11222** for the socket binding.
7. Creates a socket binding for the Memcached connector at port **11221**.
8. Defines security realms that protect endpoints from network intrusion.
9. Creates a security realm named "default".
10. Configures SSL/TLS keystores for identity verification.
11. Specifies the keystore that contains server certificates.
12. Configures the "default" security realm to use properties files to define users and groups that map users to roles.
13. Names the properties file that contains Data Grid users.
14. Specifies that contents of the **users.properties** file are stored as plain text.
15. Names the properties file that maps Data Grid users to roles.
16. Configures endpoints with Hot Rod and REST connectors.

    This example shows implicit **hotrod-connector** and **rest-connector** elements, which is the default from Data Grid 8.2.
    Data Grid Server configuration in 8.0 and 8.1 use explicitly declared Hot Rod and REST connectors.

Additional resources

- Data Grid Server Guide
- Data Grid Server Reference

2.3. DATA GRID SERVER ENDPOINT AND NETWORK CONFIGURATION

This section describes Data Grid Server endpoint and network configuration when migrating from previous versions.

Data Grid 8 simplifies server endpoint configuration by using a single network interface and port to expose endpoints on the network.

2.3.1. Interfaces

Interfaces bind expose endpoints to network locations.

Data Grid Server 7.x network interface configuration
In Data Grid 7.x, the server configuration used different interfaces to separate administrative and management access from cache access.

```
<interfaces>
  <interface name="management">
    <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
  </interface>
  <interface name="public">
    <inet-address value="${jboss.bind.address:127.0.0.1}"/>
  </interface>
</interfaces>
```

Data Grid Server 8 network interface configuration
In Data Grid 8, there is one network interface for all client connections for administrative and management access as well as cache access.

```
<interfaces>
  <interface name="public">
    <inet-address value="${infinispan.bind.address:127.0.0.1}"/>
  </interface>
</interfaces>
```

2.3.2. Socket bindings

Socket bindings map network interfaces to ports where endpoints listen for client connections.

Data Grid Server 7.x socket binding configuration
In Data Grid 7.x, the server configuration used unique ports for management and administration, such as **9990** for the Management Console and port **9999** for the native management protocol. Older versions also used unique ports for each endpoint, such as **11222** for external Hot Rod access and **8080** for REST.

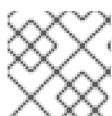
```
<socket-binding-group name="standard-sockets" default-interface="public" port-offset="${jboss.socket.binding.port-offset:0}">
  <socket-binding name="management-http" interface="management" port="${jboss.management.http.port:9990}"/>
  <socket-binding name="management-https" interface="management" port="${jboss.management.https.port:9993}"/>
  <socket-binding name="hotrod" port="11222"/>
  <socket-binding name="hotrod-internal" port="11223"/>
  <socket-binding name="hotrod-multi-tenancy" port="11224"/>
  <socket-binding name="memcached" port="11211"/>
  <socket-binding name="rest" port="8080"/>
  ...
</socket-binding-group>
```

Data Grid Server 8 single port configuration
Data Grid 8 uses a single port to handle all connections to the server. Hot Rod clients, REST clients, Data Grid CLI, and Data Grid Console all use port **11222**.

```
<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>
```

2.3.3. Endpoints

Endpoints listen for remote client connections and handle requests over protocols such as Hot Rod and HTTP (REST).



NOTE

Data Grid CLI uses the REST endpoint for all cache and administrative operations.

Data Grid Server 7.x endpoint subsystem

In Data Grid 7.x, the **endpoint** subsystem let you configure connectors for Hot Rod and REST endpoints.

```
<subsystem xmlns="urn:infinispan:server:endpoint:9.4">
  <hotrod-connector socket-binding="hotrod" cache-container="local">
    <topology-state-transfer lazy-retrieval="false" lock-timeout="1000" replication-timeout="5000"/>
  </hotrod-connector>
  <rest-connector socket-binding="rest" cache-container="local">
    <authentication security-realm="ApplicationRealm" auth-method="BASIC"/>
  </rest-connector>
</subsystem>
```

Data Grid Server 8 endpoint configuration

Data Grid 8 replaces the **endpoint** subsystem with an **endpoints** element. The **hotrod-connector** and **rest-connector** configuration elements and attributes are the same as previous versions.

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

Additional resources

- Data Grid Server Guide

2.4. DATA GRID SERVER SECURITY

Data Grid Server security configures authentication and encryption to prevent network attack and safeguard data.

2.4.1. Security realms

In Data Grid 8 security realms provide implicit configuration options that mean you do not need to provide as many settings as in previous versions. For example, if you define a Kerberos realm, you get Kerberos features. If you add a truststore, you get certificate authentication.

In Data Grid 7.x, there were two default security realms:

- **ManagementRealm** secures the Management API.
- **ApplicationRealm** secures endpoints and remote client connections.

Data Grid 8, on the other hand, provides a **security** element that lets you define multiple different security realms that you can use for Hot Rod and REST endpoints:

```
<security>
  <security-realms>
    ...
  </security-realms>
</security>
```

Supported security realms

- Property realms use property files, **users.properties** and **groups.properties**, to define users and groups that can access Data Grid.
- 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.
- Trust store realms use keystores that contain the public certificates of all clients that are allowed to access Data Grid.
- Token realms use external services to validate tokens and require providers that are compatible with RFC-7662 (OAuth2 Token Introspection) such as Red Hat SSO.

2.4.2. Server identities

Server identities use certificate chains to prove Data Grid Server identities to remote clients.

Data Grid 8 uses the same configuration to define SSL identities as in previous versions with some usability improvements.

- If a security realm contains an SSL identity, Data Grid automatically enables encryption for endpoints that use that security realm.
- For test and development environments, Data Grid includes a **generate-self-signed-certificate-host** attribute that automatically generates a keystore at startup.

```
<security-realm name="default">
  <server-identities>
    <ssl>
      <keystore path="..."
                relative-to="..."
                keystore-password="..."
                alias="..."
                key-password="..."
                generate-self-signed-certificate-host="..."/>
    </ssl>
  </server-identities>
  ...
<security-realm>
```

2.4.3. Endpoint authentication mechanisms

Hot Rod and REST endpoints use SASL or HTTP mechanisms to authenticate client connections.

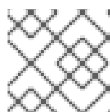
Data Grid 8 uses the same **authentication** element for **hotrod-connector** and **rest-connector** configuration as in Data Grid 7.x and earlier.

```
<hotrod-connector name="hotrod">
  <authentication>
    <sasl mechanisms="..." server-name="..."/>
  </authentication>
</hotrod-connector>
<rest-connector name="rest">
  <authentication>
    <mechanisms="..." server-principal="..."/>
  </authentication>
</rest-connector>
```

One key difference with previous versions is that Data Grid 8 supports additional authentication mechanisms for endpoints.

Hot Rod SASL authentication mechanisms

Hot Rod clients now use **SCRAM-SHA-512** as the default authentication mechanism instead of **DIGEST-MD5**.



NOTE

If you use property security realms, you must use the **PLAIN** authentication mechanism.

Authentication mechanism	Description	Related details
PLAIN	Uses credentials in plain-text format. You should use **PLAIN** authentication with encrypted connections only.	Similar to the **Basic** HTTP mechanism.
DIGEST-*	Uses hashing algorithms and nonce values. Hot Rod connectors support **DIGEST-MD5**, **DIGEST-SHA**, **DIGEST-SHA-256**, **DIGEST-SHA-384**, and **DIGEST-SHA-512** hashing algorithms, in order of strength.	Similar to the **Digest** HTTP mechanism.
SCRAM-*	Uses *salt* values in addition to hashing algorithms and nonce values. Hot Rod connectors support **SCRAM-SHA**, **SCRAM-SHA-256**, **SCRAM-SHA-384**, and **SCRAM-SHA-512** hashing algorithms, in order of strength.	Similar to the **Digest** HTTP mechanism.

Authentication mechanism	Description	Related details
GSSAPI	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding **kerberos** server identity in the realm configuration. In most cases, you also specify an **ldap-realm** to provide user membership information.	Similar to the **SPNEGO** HTTP mechanism.
GS2-KRB5	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding **kerberos** server identity in the realm configuration. In most cases, you also specify an **ldap-realm** to provide user membership information.	Similar to the **SPNEGO** HTTP mechanism.
EXTERNAL	Uses client certificates.	Similar to the **CLIENT_CERT** HTTP mechanism.
OAUTHBEARER	Uses OAuth tokens and requires a **token-realm** configuration.	Similar to the **BEARER_TOKEN** HTTP mechanism.

HTTP (REST) authentication mechanisms

Authentication mechanism	Description	Related details
BASIC	Uses credentials in plain-text format. You should use **BASIC** authentication with encrypted connections only.	Corresponds to the **Basic** HTTP authentication scheme and is similar to the **PLAIN** SASL mechanism.
DIGEST	Uses hashing algorithms and nonce values. REST connectors support **SHA-512**, **SHA-256** and **MD5** hashing algorithms.	Corresponds to the **Digest** HTTP authentication scheme and is similar to **DIGEST-*** SASL mechanisms.
SPNEGO	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding **kerberos** server identity in the realm configuration. In most cases, you also specify an **ldap-realm** to provide user membership information.	Corresponds to the **Negotiate** HTTP authentication scheme and is similar to the **GSSAPI** and **GS2-KRB5** SASL mechanisms.

Authentication mechanism	Description	Related details
BEARER_TOKEN	Uses OAuth tokens and requires a **token-realm** configuration.	Corresponds to the **Bearer** HTTP authentication scheme and is similar to **OAUTHBEARER** SASL mechanism.
CLIENT_CERT	Uses client certificates.	Similar to the **EXTERNAL** SASL mechanism.

2.4.4. Authenticating EAP applications

You can now add credentials to **hotrod-client.properties** on your EAP application classpath to authenticate with Data Grid through:

- Remote cache containers (**remote-cache-container**)
- Remote stores (**remote-store**)
- EAP modules

2.4.5. Logging

Data Grid uses Apache Log4j2 instead of the logging subsystem in previous versions that was based on JBossLogManager.

By default, Data Grid writes log messages to the following directory:
$RHDG_HOME/${infinispan.server.root}/log

server.log is the default log file.

Access logs
In previous versions Data Grid included a logger to audit security logs for the caches:

```
<authorization audit-logger="org.infinispan.security.impl.DefaultAuditLogger">
```

Data Grid 8 no longer provides this audit logger.

However you can use the logging categories for the Hot Rod and REST endpoints:

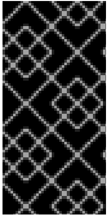
- **org.infinispan.HOTROD_ACCESS_LOG**
- **org.infinispan.REST_ACCESS_LOG**

Additional resources

- Data Grid Server Guide

2.5. SEPARATING DATA GRID SERVER ENDPOINTS

When migrating from previous versions, you can create different network locations for Data Grid endpoints to match your existing configuration. However, because Data Grid architecture has changed and now uses a single port for all client connections, not all options in previous versions are available.



IMPORTANT

Administration tools such as the Data Grid CLI and Console use the REST API. You cannot remove the REST API from your endpoint configuration without disabling the Data Grid CLI and Console. Likewise you cannot separate the REST endpoint to use different ports or socket bindings for cache access and administrative access.

Procedure

1. Define separate network interfaces for REST and Hot Rod endpoints.
   For example, define a "public" interface to expose the Hot Rod endpoint externally and a "private" interface to expose the REST endpoint on an network location that has restricted access.

```
<interfaces>
  <interface name="public">
    <inet-address value="${infinispan.bind.address:198.51.100.0}"/>
  </interface>
  <interface name="private">
    <inet-address value="${infinispan.bind.address:192.0.2.0}"/>
  </interface>
</interfaces>
```

   This configuration creates:

   - A "public" interface with the **198.51.100.0** IP address.
   - A "private" interface with the **192.0.2.0** IP address.

2. Configure separate socket bindings for the endpoints, as in the following example:

```
<socket-bindings default-interface="private"
                 port-offset="${infinispan.socket.binding.port-offset:0}">
  <socket-binding name="default"
                  port="${infinispan.bind.port:8080}"/>
  <socket-binding name="hotrod"
                  interface="public"
                  port="11222"/>
</socket-bindings>
```

   This example:

   - Sets the "private" interface as the default for socket bindings.
   - Creates a "default" socket binding that uses port **8080**.
   - Creates a "hotrod" socket binding that uses the "public" interface and port **11222**.

3. Create separate security realms for the endpoints, for example:

```
<security>
  <security-realms>
    <security-realm name="truststore">
      <server-identities>
        <ssl>
          <keystore path="server.p12"
```

```
              relative-to="infinispan.server.config.path"
              keystore-password="secret"
              alias="server"/>
    </ssl>
   </server-identities>
   <truststore-realm path="trust.p12"
                relative-to="infinispan.server.config.path"
                keystore-password="secret"/>
  </security-realm>
  <security-realm name="kerberos">
   <server-identities>
    <kerberos keytab-path="http.keytab"
            principal="HTTP/localhost@INFINISPAN.ORG"
            required="true"/>
   </server-identities>
  </security-realm>
 </security-realms>
</security>
```

This example:

- Configures a trust store security realm.
- Configures a Kerberos security realm.

4. Configure endpoints as follows:

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

5. Start Data Grid Server.
Logs contain the following messages that indicate the network locations where endpoints accept client connections:

```
[org.infinispan.SERVER] ISPN080004: Protocol HotRod listening on 198.51.100.0:11222
[org.infinispan.SERVER] ISPN080004: Protocol SINGLE_PORT listening on 192.0.2.0:8080
[org.infinispan.SERVER] ISPN080034: Server '<hostname>' listening on http://192.0.2.0:8080
```

Next steps

1. Access Data Grid Console from any browser at **http://192.0.2.0:8080**
2. Configure the Data Grid CLI to connect at the custom location, for example:

```
$ bin/cli.sh -c http://192.0.2.0:8080
```

2.6. DATA GRID SERVER SHARED DATASOURCES

Data Grid 7.x JDBC cache stores can use a **PooledConnectionFactory** to obtain database connections.

Data Grid 8 lets you create managed datasources in the server configuration to optimize connection pooling and performance for database connections with JDBC cache stores.

Datasource configurations are composed of two sections:

- **connection factory** that defines how to connect to the database.
- **connection pool** that defines how to pool and reuse connections and is based on Agroal.

You first define the datasource connection factory and connection pool in the server configuration and then add it to your JDBC cache store configuration.

For more information on migrating JDBC cache stores, see the *Migrating Cache Stores* section in this document.

Additional resources

- Data Grid Server Guide

2.7. DATA GRID SERVER JMX AND METRICS

Data Grid 8 exposes metrics via both JMX and a /**metrics** endpoint for integration with metrics tooling such as Prometheus.

The /**metrics** endpoint provides:

- Gauges that return values, such as JVM uptime or average number of seconds for cache operations.
- Histograms that show how long read, write, and remove operations take, in percentiles.

In previous versions, Prometheus metrics were collected by an agent that mapped JMX metrics instead of being supported natively.

Previous versions of Data Grid also used the JBoss Operations Network (JON) plug-in to obtain metrics and perform operations. Data Grid 8 no longer uses the JON plug-in.

Data Grid 8 separates JMX and Prometheus metrics into cache manager and cache level configurations.

```
<cache-container name="default"
                 statistics="true"> (1)
  <jmx enabled="true" /> (2)
</cache-container>
```

1. Enables statistics for the cache manager. This is the default.
2. Exports JMX MBeans, which includes all statistics and operations.

```
<distributed-cache name="mycache" statistics="true" /> (1)
```

1. Enables statistics for the cache.

Additional resources

- Data Grid Server Guide

2.8. DATA GRID SERVER CHEATSHEET

Use the following commands and examples as a quick reference for working with Data Grid Server.

Starting server instances

- Linux

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

- Microsoft Windows

```
$ bin\server.bat
```

Starting the CLI

- Linux

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

- Microsoft Windows

```
$ bin\cli.bat
```

Creating users

- Linux

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

- Microsoft Windows

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

Stopping server instances

- Single server instances

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

- Entire clusters

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

Listing available command options

Use the **-h** flag to list available command options for running servers.

- Linux

```
$ bin/server.sh -h
```

- Microsoft Windows

```
$ bin\server.bat -h
```

7.x to 8 reference

7.x	8.x
./standalone.sh -c clustered.xml	**./server.sh**
./standalone.sh	**./server.sh -c infinispan-local.xml**
-Djboss.default.multicast.address=234.99.54.20	**-Djgroups.mcast_addr=234.99.54.20**
-Djboss.bind.address=172.18.1.13	**-Djgroups.bind.address=172.18.1.13**
-Djboss.default.jgroups.stack=udp	**-j udp**

Additional resources

- Data Grid Server Guide

CHAPTER 3. MIGRATING DATA GRID CONFIGURATION

Find changes to Data Grid configuration that affect migration to Data Grid 8.

3.1. DATA GRID CACHE CONFIGURATION

Data Grid 8 provides empty cache containers by default. When you start Data Grid, it instantiates a cache manager so you can create caches at runtime.

However, in comparison with previous versions, there is no "default" cache out of the box.

In Data Grid 8, caches that you create through the **CacheContainerAdmin** API are permanent to ensure that they survive cluster restarts.

Permanent caches

```
.administration()
  .withFlags(AdminFlag.PERMANENT) (1)
  .getOrCreateCache("myPermanentCache", "org.infinispan.DIST_SYNC");
```

1. **AdminFlag.PERMANENT** is enabled by default to ensure that caches survive restarts.

You do not need to set this flag when you create caches. However, you must separately add persistent storage to Data Grid for data to survive restarts, for example:

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
  .addSingleFileStore()
  .location("/tmp/myDataStore")
  .maxEntries(5000);
```

Volatile caches

```
.administration()
  .withFlags(AdminFlag.VOLATILE) (1)
  .getOrCreateCache("myTemporaryCache", "org.infinispan.DIST_SYNC"); (2)
```

1. Sets the **VOLATILE** flag so caches are lost when Data Grid restarts.
2. Returns a cache named "myTemporaryCache" or creates one using the **DIST_SYNC** template.

Data Grid 8 provides cache templates for server installations that you can use to create caches with recommended settings.

You can get a list of available cache templates as follows:

- Use **Tab** auto-completion with the CLI:

  ```
  [//containers/default]> create cache --template=
  ```

- Use the REST API:

```
GET 127.0.0.1:11222/rest/v2/cache-managers/default/cache-configs/templates
```

3.1.1. Cache encoding

When you create remote caches you should configure the MediaType for keys and values. Configuring the MediaType guarantees the storage format for your data.

To encode caches, you specify the MediaType in your configuration. Unless you have others requirements, you should use ProtoStream, which stores your data in a language-neutral, backwards compatible format.

<encoding media-type="application/x-protostream"/>

Distributed cache configuration with encoding

```
<infinispan>
   <cache-container>
      <distributed-cache name="myCache" mode="SYNC">
        <encoding media-type="application/x-protostream"/>
         ...
      </distributed-cache>
   </cache-container>
</infinispan>
```

If you do not encode remote caches, Data Grid Server logs the following message:

```
WARN  (main) [org.infinispan.encoding.impl.StorageConfigurationManager] ISPN000599:
Configuration for cache 'mycache' does not define the encoding for keys or values. If you use
operations that require data conversion or queries, you should configure the cache with a specific
MediaType for keys or values.
```

In a future version, cache encoding will be required for operations where data conversion takes place; for example, cache indexing and searching the data container, remote task execution, reading and writing data in different formats from the Hot Rod and REST endpoints, as well as using remote filters, converters, and listeners.

3.1.2. Cache health status

Data Grid 7.x includes a Health Check API that returns health status of the cluster as well as caches within it.

Data Grid 8 also provides a Health API. For embedded and server installations, you can access the Health API via JMX with the following MBean:

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

Data Grid Server also exposes the Health API through the REST endpoint and the Data Grid Console.

Table 3.1. Health Status

7.x	8.x	Description

7.x	8.x	Description
HEALTHY	**HEALTHY**	Indicates a cache is operating as expected.
Rebalancing	**HEALTHY_REBALANCING**	Indicates a cache is in the rebalancing state but otherwise operating as expected.
Unhealthy	**DEGRADED**	Indicates a cache is not operating as expected and possibly requires troubleshooting.

Additional resources

- Configuring Data Grid Caches

3.1.3. Changes to the Data Grid 8.1 configuration schema

This topic lists changes to the Data Grid configuration schema between 8.0 and 8.1.

New and modified elements and attributes

- **stack** adds support for inline JGroups stack definitions.
- **stack.combine** and **stack.position** attributes let you override and modify JGroups stack definitions.
- **metrics** lets you configure how Data Grid exports metrics that are compatible with the Eclipse MicroProfile Metrics API.
- **context-initializer** lets you specify a **SerializationContextInitializer** implementation that initializes a Protostream-based marshaller for user types.
- **key-transformers** lets you register transformers that convert custom keys to String for indexing with Lucene.
- **statistics** now defaults to "false".

Deprecated elements and attributes

The following elements and attributes are now deprecated:

- **address-count** attribute for the **off-heap** element.
- **protocol** attribute for the **transaction** element.
- **duplicate-domains** attribute for the **jmx** element.
- **advanced-externalizer**
- **custom-interceptors**
- **state-transfer-executor**

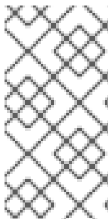
- **transaction-protocol**

Removed elements and attributes
The following elements and attributes were deprecated in a previous release and are now removed:

- **deadlock-detection-spin**
- **compatibility**
- **write-skew**
- **versioning**
- **data-container**
- **eviction**
- **eviction-thread-policy**

3.2. EVICTION CONFIGURATION

Data Grid 8 simplifies eviction configuration in comparison with previous versions. However, eviction configuration has undergone numerous changes across different Data Grid versions, which means migration might not be straightforward.



NOTE

As of Data Grid 7.2, the **memory** element replaces the **eviction** element in the configuration. This section refers to eviction configuration with the **memory** element only. For information on migrating configuration that uses the **eviction** element, refer to the Data Grid 7.2 documentation.

3.2.1. Storage types

Data Grid lets you control how to store entries in memory, with the following options:

- Store objects in JVM heap memory.
- Store bytes in native memory (off-heap).
- Store bytes in JVM heap memory.

Changes in Data Grid 8
In previous 7.x versions, and 8.0, you use **object**, **binary**, and **off-heap** elements to configure the storage type.

Starting with Data Grid 8.1, you use a **storage** attribute to store objects in JVM heap memory or as bytes in off-heap memory.

To store bytes in JVM heap memory, you use the **encoding** element to specify a binary storage format for your data.

Data Grid 7.x	Data Grid 8
<memory><object /></memory>	**<memory />**
<memory><off-heap /></memory>	**<memory storage="OFF_HEAP" />**
<memory><binary /></memory>	**<encoding media-type="..." />**

Object storage in Data Grid 8
By default, Data Grid 8.1 uses object storage (JVM heap):

```
<distributed-cache>
  <memory />
</distributed-cache>
```

You can also configure **storage="HEAP"** explicitly to store data as objects in JVM heap memory:

```
<distributed-cache>
  <memory storage="HEAP" />
</distributed-cache>
```

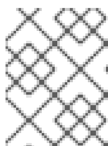
Off-heap storage in Data Grid 8
Set **"OFF_HEAP"** as the value of the **storage** attribute to store data as bytes in native memory:

```
<distributed-cache>
  <memory storage="OFF_HEAP" />
</distributed-cache>
```

Off-heap address count
In previous versions, the **address-count** attribute for **offheap** lets you specify the number of pointers that are available in the hash map to avoid collisions. With Data Grid 8.1, **address-count** is no longer used and off-heap memory is dynamically re-sized to avoid collisions.

Binary storage in Data Grid 8
Specify a binary storage format for cache entries with the **encoding** element:

```
<distributed-cache>
  <!--Configure MediaType for entries with binary formats.-->
  <encoding media-type="application/x-protostream"/>
  <memory ... />
</distributed-cache>
```



NOTE

As a result of this change, Data Grid no longer stores primitives and String mixed with **byte[]**, but stores only **byte[]**.

3.2.2. Eviction threshold

Eviction lets Data Grid control the size of the data container by removing entries when the container becomes larger than a configured threshold.

In Data Grid 7.x and 8.0, you specify two eviction types that define the maximum limit for entries in the cache:

- **COUNT** measures the number of entries in the cache.
- **MEMORY** measures the amount of memory that all entries in the cache take up.

Depending on the configuration you set, when either the count or the total amount of memory exceeds the maximum, Data Grid removes unused entries.

Data Grid 7.x and 8.0 also use the **size** attribute that defines the size of the data container as a long. Depending on the storage type you configure, eviction occurs either when the number of entries or amount of memory exceeds the value of the **size** attribute.

With Data Grid 8.1, the **size** attribute is deprecated along with **COUNT** and **MEMORY**. Instead, you configure the maximum size of the data container in one of two ways:

- Total number of entries with the **max-count** attribute.
- Maximum amount of memory, in bytes, with the **max-size** attribute.

Eviction based on total number of entries

```
<distributed-cache>
  <memory max-count="..." />
</distributed-cache>
```

Eviction based on maximum amount of memory

```
<distributed-cache>
  <memory max-size="..." />
</distributed-cache>
```

3.2.3. Eviction strategies

Eviction strategies control how Data Grid performs eviction.

Data Grid 7.x and 8.0 let you set one of the following eviction strategies with the **strategy** attribute:

Strategy	Description
NONE	Data Grid does not evict entries. This is the default setting unless you configure eviction.
REMOVE	Data Grid removes entries from memory so that the cache does not exceed the configured size. This is the default setting when you configure eviction.
MANUAL	Data Grid does not perform eviction. Eviction takes place manually by invoking the **evict()** method from the **Cache** API.

Strategy	Description
EXCEPTION	Data Grid does not write new entries to the cache if doing so would exceed the configured size. Instead of writing new entries to the cache, Data Grid throws a **ContainerFullException**.

With Data Grid 8.1, you can use the same strategies as in previous versions. However, the **strategy** attribute is replaced with the **when-full** attribute.

```
<distributed-cache>
  <memory when-full="<eviction_strategy>" />
</distributed-cache>
```

Eviction algorithms
With Data Grid 7.2, the ability to configure eviction algorithms was deprecated along with the Low Inter-Reference Recency Set (LIRS).

From version 7.2 onwards, Data Grid includes the Caffeine caching library that implements a variation of the Least Frequently Used (LFU) cache replacement algorithm known as TinyLFU. For off-heap storage, Data Grid uses a custom implementation of the Least Recently Used (LRU) algorithm.

3.2.4. Eviction configuration comparison

Compare eviction configuration between different Data Grid versions.

Object storage and evict on number of entries
7.2 to 8.0

```
<memory>
 <object size="1000000" eviction="COUNT" strategy="REMOVE"/>
</memory>
```

8.1

```
<memory max-count="1MB" when-full="REMOVE"/>
```

Object storage and evict on amount of memory
7.2 to 8.0

```
<memory>
 <object size="1000000" eviction="MEMORY" strategy="MANUAL"/>
</memory>
```

8.1

```
<memory max-size="1MB" when-full="MANUAL"/>
```

Binary storage and evict on number of entries
7.2 to 8.0

```
<memory>
  <binary size="500000000" eviction="MEMORY" strategy="EXCEPTION"/>
</memory>
```

8.1

```
<cache>
   <encoding media-type="application/x-protostream"/>
   <memory max-size="500 MB" when-full="EXCEPTION"/>
</cache>
```

Binary storage and evict on amount of memory
7.2 to 8.0

```
<memory>
  <binary size="500000000" eviction="COUNT" strategy="MANUAL"/>
</memory>
```

8.1

```
<memory max-count="500 MB" when-full="MANUAL"/>
```

Off-heap storage and evict on number of entries
7.2 to 8.0

```
<memory>
  <off-heap size="10000000" eviction="COUNT"/>
</memory>
```

8.1

```
<memory storage="OFF_HEAP" max-count="10MB"/>
```

Off-heap storage and evict on amount of memory
7.2 to 8.0

```
<memory>
  <off-heap size="1000000000" eviction="MEMORY"/>
</memory>
```

8.1

```
<memory storage="OFF_HEAP" max-size="1GB"/>
```

Additional resources

- Configuring Data Grid caches
- New eviction policy TinyLFU since RHDG 7.3 (Red Hat Knowledgebase)
- Product Documentation for Data Grid 7.2

3.3. EXPIRATION CONFIGURATION

Expiration removes entries from caches based on their lifespan or maximum idle time.

When migrating your configuration from Data Grid 7.x to 8, there are no changes that you need to make for expiration. The configuration remains the same:

Lifespan expiration

```
<expiration lifespan="1000" />
```

Max-idle expiration

```
<expiration max-idle="1000" interval="120000" />
```

For Data Grid 7.2 and earlier, using **max-idle** with clustered caches had technical limitations that resulted in performance degradation.

As of Data Grid 7.3, Data Grid sends touch commands to all owners in clustered caches when client read entries that have **max-idle** expiration values. This ensures that the entries have the same relative access time across the cluster.

Data Grid 8 sends the same touch commands for **max-idle** expiration across clusters. However there are some technical considerations you should take into account before you start using **max-idle**. Refer to *Configuring Data Grid caches* to read more about how expiration works and to review how the touch commands affect performance with clustered caches.

Additional resources

- Configuring Data Grid caches

3.4. PERSISTENT CACHE STORES

In comparison with Data Grid 7.x, there are some changes to cache store configuration in Data Grid 8.

Persistence SPI

Data Grid 8.1 introduces the **NonBlockingStore** interface for cache stores. The **NonBlockingStore** SPI exposes methods that must never block the invoking thread.

Cache stores that connect Data Grid to persistent data sources implement the **NonBlockingStore** interface.

For custom cache store implementations that use blocking operations, Data Grid provides a **BlockingManager** utility class to handle those operations.

The introduction of the **NonBlockingStore** interface deprecates the following interfaces:

- **CacheLoader**
- **CacheWriter**
- **AdvancedCacheLoader**
- **AdvancedCacheWriter**

Custom cache stores

Data Grid 8 lets you configure custom cache stores with the **store** element as in previous versions.

The following changes apply:

- The **singleton** attribute is removed. Use **shared=true** instead.
- The **segmented** attribute is added and defaults to **true**.

Segmented cache stores

As of Data Grid 8, cache store configuration defaults to **segmented="true"** and applies to the following cache store elements:

- **store**
- **file-store**
- **string-keyed-jdbc-store**
- **jpa-store**
- **remote-store**
- **rocksdb-store**
- **soft-index-file-store**

Single file cache stores

The **relative-to** attribute for Single File cache stores is removed in Data Grid 8. If your cache store configuration includes this attribute, Data Grid ignores it and uses only the **path** attribute to configure store location.

JDBC cache stores

JDBC cache stores must include an **xlmns** namespace declaration, which was not required in some Data Grid 7.x versions.

```
<persistence>
  <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:11.0" shared="true">
   ...
</persistence>
```

JDBC connection factories

Data Grid 7.x JDBC cache stores can use the following **ConnectionFactory** implementations to obtain a database connection:

- **ManagedConnectionFactory**
- **SimpleConnectionFactory**
- **PooledConnectionFactory**

Data Grid 8 now use connections factories based on Agroal, which is the same as Red Hat JBoss EAP, to connect to databases. It is no longer possible to use **c3p0.properties** and **hikari.properties** files.

Segmentation

JDBC String-Based cache store configuration that enables segmentation, which is now the default, must include the **segmentColumnName** and **segmentColumnType** parameters, as in the following programmatic examples:

MySQL Example

```
builder.table()
       .tableNamePrefix("ISPN")
       .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
       .dataColumnName("DATA_COLUMN").dataColumnType("VARBINARY(1000)")
       .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
       .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INTEGER")
```

PostgreSQL Example

```
builder.table()
       .tableNamePrefix("ISPN")
       .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
       .dataColumnName("DATA_COLUMN").dataColumnType("BYTEA")
       .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
       .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INTEGER");
```

Write-behind

The **thread-pool-size** attribute for Write-Behind mode is removed in Data Grid 8.

Removed cache stores and loaders

Data Grid 7.3 deprecates the following cache stores and loaders that are no longer available in Data Grid 8:

- Cassandra Cache Store
- REST Cache Store
- LevelDB Cache Store
- CLI Cache Loader

Cache store migrator

Cache stores in previous versions of Data Grid store data in a binary format that is not compatible with Data Grid 8.

Use the **StoreMigrator** utility to migrate data in persistent cache stores to Data Grid 8.

3.5. DATA GRID CLUSTER TRANSPORT

Data Grid uses JGroups technology to handle communication between clustered nodes.

JGroups stack configuration elements and attributes have not significantly changed from previous Data Grid versions.

As in previous versions, Data Grid provides preconfigured JGroups stacks that you can use as a starting point for building custom cluster transport configuration optimized for your network requirements. Likewise, Data Grid provides the ability to add JGroups stacks defined in external XML files to your **infinispan.xml**.

Data Grid 8 has brought usability improvements to make cluster transport configuration easier:

- Inline stacks let you configure JGroups stacks directly within **infinispan.xml** using the **jgroups** element.
- Declare JGroups schemas within **infinispan.xml** .

- Preconfigured JGroups stacks for UDP and TCP protocols.
- Inheritance attributes that let you extend JGroups stacks to adjust specific protocols and properties.

```
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:11.0 https://infinispan.org/schemas/infinispan-config-11.0.xsd
                          urn:infinispan:server:11.0 https://infinispan.org/schemas/infinispan-server-11.0.xsd
                          urn:org:jgroups http://www.jgroups.org/schema/jgroups-4.2.xsd" 1
      xmlns="urn:infinispan:config:11.0"
      xmlns:server="urn:infinispan:server:11.0">

  <jgroups> 2
    <stack name="xsite" extends="udp"> 3
      <relay.RELAY2 site="LON" xmlns="urn:org:jgroups"/>
      <remote-sites default-stack="tcp">
        <remote-site name="LON"/>
        <remote-site name="NYC"/>
      </remote-sites>
    </stack>
  </jgroups>

  <cache-container ...>
  ...
</infinispan>
```

1. Declares the JGroups 4.2 schema within **infinispan.xml**.
2. Adds a JGroups element to contain custom stack definitions.
3. Defines a JGroups protocol stack for cross-site replication.

3.5.1. Transport security

As in previous versions, Data Grid 8 uses the JGroups SYM_ENCRYPT and ASYM_ENCRYPT protocols to encrypt cluster communication.

Node authentication

In Data Grid 7.x, the JGroups SASL protocol enables nodes to authenticate against security realms in both embedded and remote server installations.

As of Data Grid 8, it is not possible to configure node authentication against security realms. Likewise Data Grid 8 does not recommend using the JGroups AUTH protocol for authenticating clustered nodes.

However, with embedded Data Grid installations, JGroups cluster transport includes a SASL configuration as part of the **jgroups** element. As in previous versions, the SASL configuration relies on JAAS notions, such as **CallbackHandlers**, to obtain certain information necessary for node authentication.

Additional resources

- Data Grid Server Guide

- Using Embedded Data Grid Caches
- Data Grid Security Guide

3.6. DATA GRID AUTHORIZATION

Data Grid uses role-based access control (RBAC) to restrict access to data and cluster encryption to secure communication between nodes.

Cache manager authorization

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

1. Requires user permission to control the cache manager lifecycle.
2. Specifies an implementation of **PrincipalRoleMapper** that maps Principals to roles.
3. Defines a set of roles and associated permissions.

Implicit cache authorization

Data Grid 8 improves usability by allowing caches to inherit authorization configuration from the **cache-container** so you do not need to explicitly configure roles and permissions for each cache.

```
<local-cache name="secured">
  <security>
    <authorization/> 1
  </security>
</local-cache>
```

1. Uses roles and permissions defined in the cache container.

Additional resources

- Data Grid Security Guide

CHAPTER 4. MIGRATING TO DATA GRID 8 APIS

Find changes to Data Grid APIs that affect migration to Data Grid 8.

API deprecations and removals

In addition to details in this section, you should also review API deprecations and removals.

See Data Grid Deprecated Features and Functionality (Red Hat Knowledgebase).

4.1. REST API

Data Grid 7.x used REST API v1 which is replaced with REST API v2 in Data Grid 8.

The default context path for REST API v2 is **<server_hostname>:11222/rest/v2/**. You must update any clients or scripts to use REST API v2.

The **performAsync** header was also removed from the REST endpoint. Clients that perform async operations with the REST endpoint should manage the request and response on their side to avoid blocking.

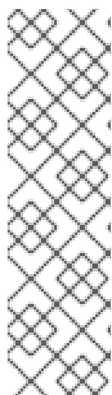
REST operations **PUT**, **POST** and **DELETE** methods now return status **204** (No content) instead of **200** if the request does not return resources.

Additional resources

- Data Grid REST API

4.2. QUERY API

Data Grid 8 brings an updated Query API that is easier to use and has a lighter design. You get more efficient query performance with better results when searching across values in distributed caches, in comparison with Data Grid 7.x.



NOTE

Because the Data Grid 8 Query API has gone through considerable refactoring, there are several features and functional resources that are now deprecated.

This topic focuses on changes that you need to make to your configuration when migrating from a previous version. Those changes should include planning to remove all deprecated interfaces, methods, or other configuration.

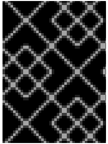
See the Data Grid Deprecations and Removals (Red Hat Knowledgebase) for the complete list of deprecated features and functionality.

Indexing Data Grid caches

The Data Grid Lucene Directory, the **InfinispanIndexManager** and **AffinityIndexManager** index managers, and the Infinispan Directory provider for Hibernate Search are deprecated in 8.0 and removed in 8.1.

The **auto-config** attribute is deprecated in 8.1 and planned for removal.

The **index()** method that configures the index mode configuration is deprecated. When you enable indexing in your configuration, Data Grid automatically chooses the best way to manage indexing.



IMPORTANT

Several indexing configuration values are no longer supported and result in fatal configuration errors if you include them.

You should make the following changes to your configuration:

- Change **.indexing().index(Index.NONE)** to **indexing().enabled(false)**
- Change all other enum values as follows: **indexing().enabled(true)**

Declaratively, you do not need to specify **enabled="true"** if your configuration contains other indexing configuration elements. However, you must call the **enabled()** method if you programmatically configure indexing. Likewise Data Grid configuration in JSON format must explicitly enable indexing, for example:

```
"indexing": {
    "enabled": "true"
    ...
    },
```

Indexed types

You must declare all indexed types in the indexing configuration or Data Grid logs warning messages when undeclared types are used with indexed caches. This requirement applies to both Java classes and Protobuf types.

Enabling indexing in Data Grid 8

- Declaratively

```
<distributed-cache name="my-cache">
  <indexing>
    <indexed-entities>
      <indexed-entity>com.acme.query.test.Car</indexed-entity>
      <indexed-entity>com.acme.query.test.Truck</indexed-entity>
    </indexed-entities>
  </indexing>
</distributed-cache>
```

- Programmatically

```
import org.infinispan.configuration.cache.*;

ConfigurationBuilder config=new ConfigurationBuilder();
config.indexing().enable().addIndexedEntity(Car.class).addIndexedEntity(Truck.class);
```

Querying values in caches

The **org.infinispan.query.SearchManager** interface is deprecated in Data Grid 8 and no longer supports Lucene and Hibernate Search native objects.

Removed methods

- **.getQuery()** methods that take Lucene Queries. Use the alternative methods that take Ickle queries from the **org.infinispan.query.Search** entry point instead.

Likewise it is no longer possible to specify multiple target entities classes when calling **.getQuery()**. The Ickle query string provides entities instead.

- **.buildQueryBuilderForClass()** that builds Hibernate Search queries directly. Use Ickle queries instead.

The **org.infinispan.query.CacheQuery** interface is also deprecated. You should obtain the **org.infinispan.query.dsl.Query** interface from the **Search.getQueryFactory()** method instead.

Note that instances of **org.infinispan.query.dsl.Query** no longer cache query results and allow queries to be re-executed when calling methods such as **list()**.

Entity mappings

You must now annotate fields that require sorting with **@SortableField** in all cases.

Additional resources

- Data Grid Query API
- Data Grid Deprecations and Removals

CHAPTER 5. MIGRATING APPLICATIONS TO DATA GRID 8

5.1. MARSHALLING IN DATA GRID 8

Marshalling capabilities are significantly refactored in Data Grid 8 to isolate internal objects and user objects.

Because Data Grid now handles marshalling of internal classes, you no longer need to handle those internal classes when configuring marshallers with embedded or remote caches.

5.1.1. ProtoStream marshalling

By default, Data Grid 8 uses the ProtoStream API to marshall data as Protocol Buffers, a language-neutral, backwards compatible format.

Protobuf encoding is a schema-defined format that is now a default standard for many applications and allows greater flexibility when transcoding data in comparison with JBoss Marshalling, which was the default in Data Grid 7.

Because the ProtoStream marshaller is based on the Protobuf format, Data Grid can convert to other encodings without first converting to a Java object. When using JBoss Marshalling, it is necessary to convert keys and values to Java objects before converting to any other format.

As part of your migration to Data Grid 8, you should start using ProtoStream marshalling for your Java classes.

From a high-level, to use the ProtoStream marshaller, you generate **SerializationContextInitializer** implementations with the ProtoStream processor. First, you add **@Proto** annotations to your Java classes and then use a ProtoStream processor that Data Grid provides to generate serialization contexts that contain:

- **.proto** schemas that provide a structured representation of your Java objects as Protobuf message types.
- Marshaller implementations to encode your Java objects to Protobuf format.

Depending on whether you use embedded or remote caches, Data Grid can automatically register your **SerializationContextInitializer** implementations.

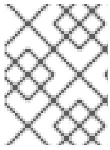
Marshalling with Data Grid Server
You should use only Protobuf encoding for remote caches in combination with the ProtoStream marshaller for any custom types.

Other marshaller implementations, such as JBoss marshalling, require you to use different cache encodings that are not compatible with the Data Grid CLI, Data Grid Console, or with Ickle queries.

Cache stores and ProtoStream
In Data Grid 7.x, data that you persist to a cache store is not compatible with the ProtoStream marshaller in Data Grid 8. You must use the **StoreMigrator** utility to migrate data from any Data Grid 7.x cache store to a Data Grid 8 cache store.

5.1.2. Alternative marshaller implementations

Data Grid does provide alternative marshaller implementations to ProtoStream help ease migration from older versions. You should use those alternative marshallers only as an interim solution while you migrate to ProtoStream marshalling.

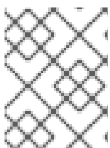


NOTE

For new projects Red Hat strongly recommends you use only ProtoStream marshalling to avoid any issues with future upgrades or migrations.

JBoss marshalling

In Data Grid 7, JBoss Marshalling is the default marshaller. In Data Grid 8, ProtoStream marshalling is the default.



NOTE

You should use **JavaSerializationMarshaller** instead of JBoss Marshalling if you have a client requirement to use Java serialization.

If you must use JBoss Marshalling as a temporary solution during migration to Data Grid 8, do the following:

Embedded caches

1. Add the **infinispan-jboss-marshalling** dependency to your classpath.
2. Configure Data Grid to use the **JBossUserMarshaller**, for example:

```
<serialization marshaller="org.infinispan.jboss.marshalling.core.JBossUserMarshaller"/>
```

3. Add your classes to the list of classes that Data Grid allows for deserialization.

Remote caches

Data Grid Server does not support JBoss Marshalling and the **GenericJBossMarshaller** is no longer automatically configured if the **infinispan-jboss-marshalling** module is on the classpath.

You must configure Hot Rod Java clients to use JBoss Marshalling as follows:

- **RemoteCacheManager**

```
.marshaller("org.infinispan.jboss.marshalling.commons.GenericJBossMarshaller");
```

- **hotrod-client.properties**

```
infinispan.client.hotrod.marshaller = GenericJBossMarshaller
```

Additional resources

- Cache Encoding and Marshalling

5.2. MIGRATING APPLICATIONS TO THE AUTOPROTOSCHEMABUILDER ANNOTATION

Previous versions of Data Grid use the **MessageMarshaller** interface in the ProtoStream API to configure marshalling.

Both the **MessageMarshaller** API and the **ProtoSchemaBuilder** annotation are deprecated as of Data Grid 8.1.1, which corresponds to ProtoStream 4.3.4.

Using the **MessageMarshaller** interface involves either:

- Manually creating Protobuf schema.
- Adding the **ProtoSchemaBuilder** annotation to Java classes and then generating Protobuf schema.

However, these techniques for configuring ProtoStream marshalling are not as efficient and reliable as the **AutoProtoSchemaBuilder** annotation, which is available starting with Data Grid 8.1.1. Simply add the **AutoProtoSchemaBuilder** annotation to your Java classes and to generate **SerializationContextInitializer** implementations that include Protobuf schema and associated marshallers.

Red Hat recommends that you start using the **AutoProtoSchemaBuilder** annotation to get the best results from the ProtoStream marshaller.

The following code examples demonstrate how you can migrate applications from the **MessageMarshaller** API to the **AutoProtoSchemaBuilder** annotation.

5.2.1. Basic MessageMarshaller implementation

This example contains some fields that use non-default types. The **text** field has a different order and the **fixed32** field conflicts with the generated Protobuf schema type because the code generator uses **int** type by default.

SimpleEntry.java

```
public class SimpleEntry {

  private String description;
  private Collection<String> text;
  private int intDefault;
  private Integer fixed32;

  // public Getter, Setter, equals and HashCode methods omitted for brevity
}
```

SimpleEntryMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;

public class SimpleEntryMarshaller implements MessageMarshaller<SimpleEntry> {

  @Override
  public void writeTo(ProtoStreamWriter writer, SimpleEntry testEntry) throws IOException {
    writer.writeString("description", testEntry.getDescription());
    writer.writeInt("intDefault", testEntry.getIntDefault());
    writer.writeInt("fix32", testEntry.getFixed32());
    writer.writeCollection("text", testEntry.getText(), String.class);
```

```
  }

  @Override
  public SimpleEntry readFrom(MessageMarshaller.ProtoStreamReader reader) throws IOException {
    SimpleEntry x = new SimpleEntry();

    x.setDescription(reader.readString("description"));
    x.setIntDefault(reader.readInt("intDefault"));
    x.setFixed32(reader.readInt("fix32"));
    x.setText(reader.readCollection("text", new LinkedList<String>(), String.class));

    return x;
  }
}
```

Resulting Protobuf schema

```
syntax = "proto2";

package example;

message SimpleEntry {
   required string description = 1;
   optional int32 intDefault = 2;
   optional fixed32 fix32 = 3;
   repeated string text = 4;
}
```

Migrated to the AutoProtoSchemaBuilder annotation

SimpleEntry.java

```
import org.infinispan.protostream.annotations.ProtoField;
import org.infinispan.protostream.descriptors.Type;

public class SimpleEntry {

  private String description;
  private Collection<String> text;
  private int intDefault;
  private Integer fixed32;

  @ProtoField(number = 1)
  public String getDescription() {...}

  @ProtoField(number = 4, collectionImplementation = LinkedList.class)
  public Collection<String> getText() {...}

  @ProtoField(number = 2, defaultValue = "0")
  public int getIntDefault() {...}

  @ProtoField(number = 3, type = Type.FIXED32)
  public Integer getFixed32() {...}
```

```
   // public Getter, Setter, equals and HashCode methods and convenient constructors omitted for
brevity
}
```

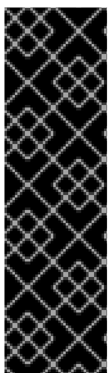
SimpleEntryInitializer.java

```
import org.infinispan.protostream.GeneratedSchema;
import org.infinispan.protostream.annotations.AutoProtoSchemaBuilder;

@AutoProtoSchemaBuilder(includeClasses = { SimpleEntry.class }, schemaFileName =
"simple.proto", schemaFilePath = "proto", schemaPackageName = "example")
public interface SimpleEntryInitializer extends GeneratedSchema {
}
```

Important observations

- **Field 2** is defined as **int** which the ProtoStream marshaller in previous versions did not check.
- Because the Java **int** field is not nullable the ProtoStream processor will fail.
  The Java **int** field must be **required** or initialized with a **defaultValue**.
  From a Java application perspective, the **int** field is initialized with "0" so you can use **defaultValue** without any impact as any put operation will set it. Change to **required** is not a problem from the stored data perspective if always present, but it might cause issues for different clients.
- **Field 3** must be explicitly set to **Type.FIXED32** for compatibility.
- The **text collection** must be set in the correct order for the resulting Protobuf schema.



IMPORTANT

The order of the **text collection** in your Protobuf schema must be the same before and after migration. Likewise, you must set the **fixed32** type during migration.

If not, client applications might throw the following exception and fail to start:

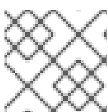
```
Exception ( ISPN004034: Unable to unmarshall bytes )
```

In other cases, you might observe incomplete or inaccurate results in your cached data.

5.2.2. MessageMarshaller implementation with custom types

This section provides an example migration for a **MessageMarshaller** implementation that contains fields that ProtoStream does not natively handle.

The following example uses the **BigInteger** class but applies to any class, even a Data Grid adapter or a custom class.



NOTE

The **BigInteger** class is immutable so does not have a no-argument constructor.

CustomTypeEntry.java

```
import java.math.BigInteger;

public class CustomTypeEntry {

  final String description;
  final BigInteger bigInt;

  // public Getter, Setter, equals and HashCode methods and convenient constructors omitted for
brevity
}
```

CustomTypeEntryMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;

public class CustomTypeEntryMarshaller implements MessageMarshaller<CustomTypeEntry> {

  @Override
  public void writeTo(ProtoStreamWriter writer, CustomTypeEntry testEntry) throws IOException {
    writer.writeString("description", testEntry.description);
    writer.writeString("bigInt", testEntry.bigInt.toString());
  }

  @Override
  public CustomTypeEntry readFrom(MessageMarshaller.ProtoStreamReader reader) throws
IOException {
    final String desc = reader.readString("description");
    final BigInteger bInt = new BigInteger(reader.readString("bigInt"));

    return new CustomTypeEntry(desc, bInt);
  }
}
```

CustomTypeEntry.proto

```
syntax = "proto2";

package example;

message CustomTypeEntry {
   required string description = 1;
   required string bigInt = 2;
}
```

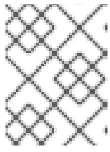
Migrated code with an adapter class

You can use the **ProtoAdapter** annotation to marshall a **CustomType** class in a way that generates Protobuf schema that is compatible with Protobuf schema that you created with **MessageMarshaller** implementations.

With this approach, you:

- Must not add annotations to the **CustomTypeEntry** class.

- Create a **CustomTypeEntryAdapter** class that uses the **@ProtoAdapter** annotation to control how the Protobuf schema and marshaller is generated.
- Include the **CustomTypeEntryAdapter** class with the **@AutoProtoSchemaBuilder** annotation.



NOTE

Because the **AutoProtoSchemaBuilder** annotation does not reference the **CustomTypeEntry** class, any annotations contained in that class are ignored.

The following example shows the **CustomTypeEntryAdapter** class that contains ProtoStream annotations for the **CustomTypeEntry** class:

CustomTypeEntryAdapter.java

```
import java.math.BigInteger;

import org.infinispan.protostream.annotations.ProtoAdapter;
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;

@ProtoAdapter(CustomTypeEntry.class)
public class CustomTypeEntryAdapter {

  @ProtoFactory
  public CustomTypeEntry create(String description, String bigInt) {
    return new CustomTypeEntry(description, new BigInteger(bigInt));
  }

  @ProtoField(number = 1, required = true)
  public String getDescription(CustomTypeEntry t) {
    return t.description;
  }

  @ProtoField(number = 2, required = true)
  public String getBigInt(CustomTypeEntry t) {
    return t.bigInt.toString();
  }
}
```

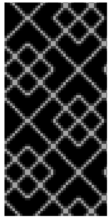
The following example shows the **SerializationContextInitializer** with **AutoProtoSchemaBuilder** annotations that reference the **CustomTypeEntryAdapter** class:

CustomTypeEntryInitializer.java

```
import org.infinispan.protostream.GeneratedSchema;
import org.infinispan.protostream.annotations.AutoProtoSchemaBuilder;

@AutoProtoSchemaBuilder(includeClasses = { CustomTypeEntryAdapter.class },
   schemaFileName = "custom.proto",
   schemaFilePath = "proto",
   schemaPackageName = "example")
public interface CustomTypeAdapterInitializer extends GeneratedSchema { }
```

Migrated code without an adapter class
Instead of creating an adapter class, you can add ProtoStream annotations directly to the **CustomTypeEntry** class.



IMPORTANT

In this example, the generated Protobuf schema is not compatible with data in caches that was added via the **MessageMarshaller** interface because the **BigInteger** is a separate message. Even if the adapter field writes the same String, it is not possible to unmarshall the data.

The following example shows the **CustomTypeEntry** class that directly contains ProtoStream annotations:

CustomTypeEntry.java

```
import java.math.BigInteger;

public class CustomTypeEntry {

  @ProtoField(number = 1)
  final String description;
  @ProtoField(number = 2)
  final BigInteger bigInt;

  @ProtoFactory
  public CustomTypeEntry(String description, BigInteger bigInt) {
    this.description = description;
    this.bigInt = bigInt;
  }

  // public Getter, Setter, equals and HashCode methods and convenient constructors omitted for
brevity
}
```

The following example shows the **SerializationContextInitializer** with **AutoProtoSchemaBuilder** annotations that reference the **CustomTypeEntry** and **BigIntegerAdapter** classes:

CustomTypeEntryInitializer.java

```
import org.infinispan.protostream.GeneratedSchema;
import org.infinispan.protostream.annotations.AutoProtoSchemaBuilder;
import org.infinispan.protostream.types.java.math.BigIntegerAdapter;

@AutoProtoSchemaBuilder(includeClasses = { CustomTypeEntry.class,
   BigIntegerAdapter.class },
   schemaFileName = "customtype.proto",
   schemaFilePath = "proto",
   schemaPackageName = "example")
public interface CustomTypeInitializer extends GeneratedSchema { }
```

When you generate the Protobuf schema from the preceding **SerializationContextInitializer** implementation, it results in the following Protobuf schema:

CustomTypeEntry.proto

```
syntax = "proto2";

package example;

message BigInteger {
   optional bytes bytes = 1;
}

message CustomTypeEntry {
   optional string description = 1;
   optional BigInteger bigInt = 2;
}
```

CHAPTER 6. MIGRATING DATA GRID CLUSTERS ON RED HAT OPENSHIFT

Review migration details for Data Grid clusters running on Red Hat OpenShift.

6.1. DATA GRID ON OPENSHIFT

Data Grid 8 introduces Data Grid Operator that provides operational intelligence and reduces management complexity for deploying Data Grid on OpenShift.

Red Hat supports Data Grid 8 on OpenShift only through Data Grid Operator subscriptions.

With Data Grid 8, Data Grid Operator handles most configuration for Data Grid clusters, including authentication, client keystores, external network access, and logging.

Creating Data Grid Services

Data Grid 7.3 introduced the Cache service and Data Grid service for creating Data Grid clusters on OpenShift.

To create these services in Data Grid 7.3, you import the service templates, if necessary, and then use template parameters and environment variables to configure the services.

Creating Cache service nodes in 7.3

```
$ oc new-app cache-service \
  -p APPLICATION_USER=${USERNAME} \
  -p APPLICATION_PASSWORD=${PASSWORD} \
  -p NUMBER_OF_INSTANCES=3 \
  -p REPLICATION_FACTOR=2
```

Creating Data Grid service nodes in 7.3

```
$ oc new-app datagrid-service \
  -p APPLICATION_USER=${USERNAME} \
  -p APPLICATION_PASSWORD=${PASSWORD} \
  -p NUMBER_OF_INSTANCES=3
  -e AB_PROMETHEUS_ENABLE=true
```

Creating services in Data Grid 8

1. Create an Data Grid Operator subscription.
2. Create an **Infinispan** Custom Resource (CR) to instantiate and configure Data Grid clusters.

```
apiVersion: infinispan.org/v1
kind: Infinispan
metadata:
  name: example-infinispan
spec:
  replicas: 2
  service:
    type: Cache 1
```

1 The **spec.service.type** field specifies whether you create Cache service or Data Grid service nodes.

6.1.1. Container storage

Data Grid 7.3 services use storage volumes mounted at **/opt/datagrid/standalone/data**.

Data Grid 8 services use persistent volume claims mounted at **/opt/infinispan/server/data**.

6.1.2. Data Grid CLI

Data Grid 7.3 let you access the CLI through remote shells only. Changes that you made to via the Data Grid 7.3 CLI were bound to the pod and did not survive restarts. With Data Grid 8 you can use the CLI as a fully functional mechanism for performing administrative operations with clusters on OpenShift or manipulating data.

6.1.3. Data Grid console

Data Grid 7.3 did not support the console on OpenShift. With Data Grid 8 you can use the console to monitor clusters running on OpenShift, perform administrative operations, and create caches remotely.

6.1.4. Customizing Data Grid

Data Grid 7.3 let you use the Source-to-Image (S2I) process and **ConfigMap** API to customize Data Grid server images running on OpenShift.

In Data Grid 8, Red Hat does not support customization of any Data Grid images from the Red Hat Container Registry.

Data Grid Operator handles the deployment and management of Data Grid 8 clusters on OpenShift.

As a result it is not possible to use custom:

- Discovery protocols
- Encryption mechanisms (SYM_ENCRYPT or ASYM_ENCRYPT)
- Persistent datasources

In Data Grid 8.0 and 8.1, Data Grid Operator does not allow you to deploy custom code such as **JAR** files or other artefacts.

6.1.5. Deployment configuration templates

The deployment configuration templates, and environment variables, that were available in Data Grid 7.3 are removed in Data Grid 8.

CHAPTER 7. MIGRATING DATA BETWEEN CACHE STORES

Data Grid provides a Java utility for migrating persisted data between cache stores.

In the case of upgrading Data Grid, functional differences between major versions do not allow backwards compatibility between cache stores. You can use **StoreMigrator** to convert your data so that it is compatible with the target version.

For example, upgrading to Data Grid 8.0 changes the default marshaller to Protostream. In previous Data Grid versions, cache stores use a binary format that is not compatible with the changes to marshalling. This means that Data Grid 8.0 cannot read from cache stores with previous Data Grid versions.

In other cases Data Grid versions deprecate or remove cache store implementations, such as JDBC Mixed and Binary stores. You can use **StoreMigrator** in these cases to convert to different cache store implementations.

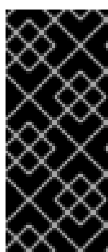
7.1. CACHE STORE MIGRATOR

Data Grid provides the **StoreMigrator.java** utility that recreates data for the latest Data Grid cache store implementations.

StoreMigrator takes a cache store from a previous version of Data Grid as source and uses a cache store implementation as target.

When you run **StoreMigrator**, it creates the target cache with the cache store type that you define using the **EmbeddedCacheManager** interface. **StoreMigrator** then loads entries from the source store into memory and then puts them into the target cache.

StoreMigrator also lets you migrate data from one type of cache store to another. For example, you can migrate from a JDBC String-Based cache store to a Single File cache store.



IMPORTANT

StoreMigrator cannot migrate data from segmented cache stores to:

- Non-segmented cache store.
- Segmented cache stores that have a different number of segments.

7.2. GETTING THE STORE MIGRATOR

StoreMigrator is available as part of the Data Grid tools library, **infinispan-tools**, and is included in the Maven repository.

Procedure

- Configure your **pom.xml** for **StoreMigrator** as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.infinispan.example</groupId>
    <artifactId>jdbc-migrator-example</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
      <dependency>
        <groupId>org.infinispan</groupId>
        <artifactId>infinispan-tools</artifactId>
      </dependency>
      <!-- Additional dependencies -->
    </dependencies>

    <build>
      <plugins>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <version>1.2.1</version>
          <executions>
            <execution>
              <goals>
                <goal>java</goal>
              </goals>
            </execution>
          </executions>
          <configuration>
            <mainClass>org.infinispan.tools.store.migrator.StoreMigrator</mainClass>
            <arguments>
              <argument>path/to/migrator.properties</argument>
            </arguments>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

7.3. CONFIGURING THE STORE MIGRATOR

Set properties for source and target cache stores in a **migrator.properties** file.

Procedure

1. Create a **migrator.properties** file.
2. Configure the source cache store in **migrator.properties**.
    a. Prepend all configuration properties with **source.** as in the following example:

    ```
    source.type=SOFT_INDEX_FILE_STORE
    source.cache_name=myCache
    source.location=/path/to/source/sifs
    ```

3. Configure the target cache store in **migrator.properties**.

a. Prepend all configuration properties with **target.** as in the following example:

```
target.type=SINGLE_FILE_STORE
target.cache_name=myCache
target.location=/path/to/target/sfs.dat
```

7.3.1. Store Migrator Properties

Configure source and target cache stores in a **StoreMigrator** properties.

Table 7.1. Cache Store Type Property

Property	Description	Required/Optional
type	Specifies the type of cache store type for a source or target. **.type=JDBC_STRING** **.type=JDBC_BINARY** **.type=JDBC_MIXED** **.type=LEVELDB** **.type=ROCKSDB** **.type=SINGLE_FILE_STORE** **.type=SOFT_INDEX_FILE_STORE** **.type=JDBC_MIXED**	Required

Table 7.2. Common Properties

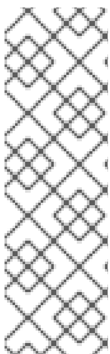
Property	Description	Example Value	Required/Optional
cache_name	Names the cache that the store backs.	**.cache_name=myCache**	Required

Property	Description	Example Value	Required/Optional
segment_count	Specifies the number of segments for target cache stores that can use segmentation. The number of segments must match **clustering.hash.numSegments** in the Data Grid configuration. In other words, the number of segments for a cache store must match the number of segments for the corresponding cache. If the number of segments is not the same, Data Grid cannot read data from the cache store.	**.segment_count=256**	Optional

Table 7.3. JDBC Properties

Property	Description	Required/Optional
dialect	Specifies the dialect of the underlying database.	Required
version	Specifies the marshaller version for source cache stores. Set one of the following values: * **8** for Data Grid 7.2.x * **9** for Data Grid 7.3.x * **10** Data Grid 8.x	Required for source stores only. For example: **source.version=9**
marshaller.class	Specifies a custom marshaller class.	Required if using custom marshallers.
marshaller.externalizers	Specifies a comma-separated list of custom **AdvancedExternalizer** implementations to load in this format: **[id]:<Externalizer class>**	Optional

Property	Description	Required/Optional
connection_pool.connection_url	Specifies the JDBC connection URL.	Required
connection_pool.driver_class	Specifies the class of the JDBC driver.	Required
connection_pool.username	Specifies a database username.	Required
connection_pool.password	Specifies a password for the database username.	Required
db.major_version	Sets the database major version.	Optional
db.minor_version	Sets the database minor version.	Optional
db.disable_upsert	Disables database upsert.	Optional
db.disable_indexing	Specifies if table indexes are created.	Optional
table.string.table_name_prefix	Specifies additional prefixes for the table name.	Optional
table.string.<id|data|timestamp>.name	Specifies the column name.	Required
table.string.<id|data|timestamp>.type	Specifies the column type.	Required
key_to_string_mapper	Specifies the **TwoWayKey2StringMapper** class.	Optional



NOTE

To migrate from Binary cache stores in older Data Grid versions, change **table.string.*** to **table.binary.*** in the following properties:

- **source.table.binary.table_name_prefix**
- **source.table.binary.<id\|data\|timestamp>.name**
- **source.table.binary.<id\|data\|timestamp>.type**

```
# Example configuration for migrating to a JDBC String-Based cache store
target.type=STRING
```

```
target.cache_name=myCache
target.dialect=POSTGRES
target.marshaller.class=org.example.CustomMarshaller
target.marshaller.externalizers=25:Externalizer1,org.example.Externalizer2
target.connection_pool.connection_url=jdbc:postgresql:postgres
target.connection_pool.driver_class=org.postrgesql.Driver
target.connection_pool.username=postgres
target.connection_pool.password=redhat
target.db.major_version=9
target.db.minor_version=5
target.db.disable_upsert=false
target.db.disable_indexing=false
target.table.string.table_name_prefix=tablePrefix
target.table.string.id.name=id_column
target.table.string.data.name=datum_column
target.table.string.timestamp.name=timestamp_column
target.table.string.id.type=VARCHAR
target.table.string.data.type=bytea
target.table.string.timestamp.type=BIGINT
target.key_to_string_mapper=org.infinispan.persistence.keymappers.
DefaultTwoWayKey2StringMapper
```

Table 7.4. RocksDB Properties

Property	Description	Required/Optional
location	Sets the database directory.	Required
compression	Specifies the compression type to use.	Optional

```
# Example configuration for migrating from a RocksDB cache store.
source.type=ROCKSDB
source.cache_name=myCache
source.location=/path/to/rocksdb/database
source.compression=SNAPPY
```

Table 7.5. SingleFileStore Properties

Property	Description	Required/Optional
location	Sets the directory that contains the cache store **.dat** file.	Required

```
# Example configuration for migrating to a Single File cache store.
target.type=SINGLE_FILE_STORE
target.cache_name=myCache
target.location=/path/to/sfs.dat
```

Table 7.6. SoftIndexFileStore Properties

Property	Description	Value
Required/Optional	**location**	Sets the database directory.
Required	**index_location**	Sets the database index directory.

```
# Example configuration for migrating to a Soft-Index File cache store.
target.type=SOFT_INDEX_FILE_STORE
target.cache_name=myCache
target.location=path/to/sifs/database
target.location=path/to/sifs/index
```

7.4. MIGRATING CACHE STORES

Run **StoreMigrator** to migrate data from one cache store to another.

Prerequisites

- Get **infinispan-tools.jar**.
- Create a **migrator.properties** file that configures the source and target cache stores.

Procedure

- If you build **infinispan-tools.jar** from source, do the following:
  1. Add **infinispan-tools.jar** and dependencies for your source and target databases, such as JDBC drivers, to your classpath.
  2. Specify **migrator.properties** file as an argument for **StoreMigrator**.
- If you pull **infinispan-tools.jar** from the Maven repository, run the following command: **mvn exec:java**