Red Hat Data Grid 8.4

Embedding Data Grid in Java Applications

Create embedded caches with Data Grid

Last Updated: 2024-04-19

Red Hat Data Grid 8.4 Embedding Data Grid in Java Applications

Create embedded caches with Data Grid

Legal Notice

Copyright © 2024 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

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

Table of Contents

RED HAT DATA GRID 5
DATA GRID DOCUMENTATION 6
DATA GRID DOWNLOADS 7
MAKING OPEN SOURCE MORE INCLUSIVE 8
CHAPTER 1. ADDING DATA GRID TO YOUR MAVEN REPOSITORY 9
1.1. DOWNLOADING THE MAVEN REPOSITORY 9
1.2. ADDING RED HAT MAVEN REPOSITORIES 9
1.3. CONFIGURING YOUR PROJECT POM 10
CHAPTER 2. CREATING EMBEDDED CACHES 11
2.1. ADDING DATA GRID TO YOUR PROJECT 11
2.2. CREATING AND USING EMBEDDED CACHES 11
2.3. CACHE API 13
2.3.1. AdvancedCache API 14
2.3.1.1. Flags 14
2.3.2. Asynchronous API 15
2.3.2.1. Why use such an API? 15
2.3.2.2. Which processes actually happen asynchronously? 15
CHAPTER 3. PROGRAMMATICALLY CONFIGURING USER ROLES AND PERMISSIONS 16
3.1. DATA GRID USER ROLES AND PERMISSIONS 16
3.1.1. Permissions 16
3.1.2. Role and permission mappers 18
3.1.2.1. Mapping users to roles and permissions in Data Grid 19
3.1.3. Configuring role mappers 19
Role mapper configuration 20
3.2. ENABLING AND CONFIGURING AUTHORIZATION FOR EMBEDDED CACHES 20
3.3. ADDING AUTHORIZATION ROLES AT RUNTIME 21
3.4. EXECUTING CODE WITH SECURE CACHES 22
3.5. CONFIGURING THE ACCESS CONTROL LIST (ACL) CACHE 22
ACL cache configuration 23
CHAPTER 4. ENABLING AND CONFIGURING DATA GRID STATISTICS AND JMX MONITORING 25
4.1. ENABLING STATISTICS IN EMBEDDED CACHES 25
Embedded cache statistics 25
4.2. CONFIGURING DATA GRID METRICS 25
Metrics configuration 26
4.3. REGISTERING JMX MBEANS 27
JMX configuration 27
4.3.1. Enabling JMX remote ports 28
4.3.2. Data Grid MBeans 29
4.3.3. Registering MBeans in custom MBean servers 29
JMX MBean server lookup configuration 30
4.4. EXPORTING METRICS DURING A STATE TRANSFER OPERATION 31
4.5. MONITORING THE STATUS OF CROSS-SITE REPLICATION 31
Monitoring cross-site replication with the REST API 32
Monitoring cross-site replication with the Prometheus metrics 34
CHAPTER 5. SETTING UP DATA GRID CLUSTER TRANSPORT 36

5.1. DEFAULT JGROUPS STACKS 36
5.2. CLUSTER DISCOVERY PROTOCOLS 36
5.2.1. PING 37
5.2.2. TCPPING 37
5.2.3. MPING 38
5.2.4. TCPGOSSIP 38
5.2.5. JDBC_PING 38
5.2.6. DNS_PING 39
5.2.7. Cloud discovery protocols 39
Providing dependencies for cloud discovery protocols 40
5.3. USING THE DEFAULT JGROUPS STACKS 40
5.4. CUSTOMIZING JGROUPS STACKS 41
5.4.1. Inheritance attributes 42
5.5. USING JGROUPS SYSTEM PROPERTIES 42
5.5.1. Cluster transport properties 43
5.5.2. System properties for cloud discovery protocols 44
5.5.2.1. Amazon EC2 44
5.5.2.2. Google Cloud Platform 45
5.5.2.3. Azure 45
5.5.2.4. OpenShift 45
5.6. USING INLINE JGROUPS STACKS 46
5.7. USING EXTERNAL JGROUPS STACKS 47
5.8. USING CUSTOM JCHANNELS 48
5.9. ENCRYPTING CLUSTER TRANSPORT 48
5.9.1. JGroups encryption protocols 48
5.9.2. Securing cluster transport with asymmetric encryption 49
5.9.3. Securing cluster transport with symmetric encryption 51
5.10. TCP AND UDP PORTS FOR CLUSTER TRAFFIC 52
Cross-site replication 52

CHAPTER 6. CLUSTERED LOCKS 53
6.1. LOCK API 53
6.2. USING CLUSTERED LOCKS 53
6.3. CONFIGURING INTERNAL CACHES FOR LOCKS 55

CHAPTER 7. EXECUTING CODE IN THE GRID 57
7.1. CLUSTER EXECUTOR 57
7.1.1. Filtering execution nodes 57
7.1.2. Timeout 58
7.1.3. Single Node Submission 58
7.1.3.1. Failover 58
7.1.4. Example: PI Approximation 58

CHAPTER 8. USING THE STREAMS API FOR CODE EXECUTION 61

CHAPTER 9. STREAMS 62
9.1. COMMON STREAM OPERATIONS 62
9.2. KEY FILTERING 62
9.3. SEGMENT BASED FILTERING 62
9.4. LOCAL/INVALIDATION 63
9.5. EXAMPLE 63
9.6. DISTRIBUTION/REPLICATION/SCATTERED 63
9.6.1. Rehash Aware 63
9.6.2. Serialization 63

9.7. PARALLEL COMPUTATION 66
9.8. TASK TIMEOUT 66
9.9. INJECTION 67
9.10. DISTRIBUTED STREAM EXECUTION 67
9.11. KEY BASED REHASH AWARE OPERATORS 68
9.12. INTERMEDIATE OPERATION EXCEPTIONS 68
9.13. EXAMPLES 69

CHAPTER 10. USING THE CDI EXTENSION 72
10.1. CDI DEPENDENCIES 72
10.2. INJECTING EMBEDDED CACHES 72
10.3. INJECTING REMOTE CACHES 74
10.4. JCACHE CACHING ANNOTATIONS 75
10.5. RECEIVING CACHE AND CACHE MANAGER EVENTS 77

CHAPTER 11. USING THE JCACHE API 78
11.1. CREATING EMBEDDED CACHES 78
11.1.1. Configuring embedded caches 78
11.2. STORE AND RETRIEVE DATA 79
11.3. COMPARING JAVA.UTIL.CONCURRENT.CONCURRENTMAP AND JAVAX.CACHE.CACHE APIS 79
11.4. CLUSTERING JCACHE INSTANCES 81

CHAPTER 12. MULTIMAP CACHE 83
12.1. MULTIMAP CACHE 83
12.1.1. Installation and configuration 83
12.1.2. MultimapCache API 83
12.1.3. Creating a Multimap Cache 85
12.1.3.1. Embedded mode 85
12.1.4. Limitations 85
12.1.4.1. Support for duplicates 85
12.1.4.2. Eviction 85
12.1.4.3. Transactions 85

CHAPTER 13. DATA GRID MODULES FOR RED HAT JBOSS EAP 86
13.1. INSTALLING DATA GRID MODULES 86
13.2. CONFIGURING APPLICATIONS TO USE DATA GRID MODULES 86

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.

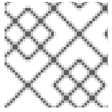
DATA GRID DOCUMENTATION

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

- Data Grid 8.4 Documentation
- Data Grid 8.4 Component Details
- Supported Configurations for Data Grid 8.4
- 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. ADDING DATA GRID TO YOUR MAVEN REPOSITORY

Data Grid Java distributions are available from Maven.

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

1.1. DOWNLOADING THE MAVEN REPOSITORY

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

Procedure

1. Log in to the Red Hat customer portal.
2. Navigate to the Software Downloads for Data Grid.
3. Download the Red Hat Data Grid 8.4 Maven Repository.
4. Extract the archived Maven repository to your local file system.
5. Open the **README.md** file and follow the appropriate installation instructions.

1.2. ADDING RED HAT MAVEN REPOSITORIES

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

Procedure

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

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

Reference

- Red Hat Enterprise Maven Repository

1.3. CONFIGURING YOUR PROJECT POM

Configure Project Object Model (POM) files in your project to use Data Grid dependencies for embedded caches, Hot Rod clients, and other capabilities.

Procedure

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

The following example shows the Data Grid version and BOM:

```
<properties>
  <version.infinispan>14.0.21.Final-redhat-00001</version.infinispan>
</properties>

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

Next Steps

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

CHAPTER 2. CREATING EMBEDDED CACHES

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

2.1. ADDING DATA GRID TO YOUR PROJECT

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

Prerequisites

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

Procedure

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

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

2.2. CREATING AND USING EMBEDDED CACHES

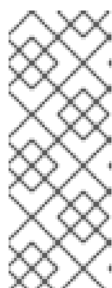
Data Grid provides a **GlobalConfigurationBuilder** API that controls the Cache Manager and a **ConfigurationBuilder** API that configures caches.

Prerequisites

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

Procedure

1. Initialize a **CacheManager**.



NOTE

You must always call the **cacheManager.start()** method to initialize a **CacheManager** before you can create caches. Default constructors do this for you but there are overloaded versions of the constructors that do not.

Cache Managers are also heavyweight objects and Data Grid recommends instantiating only one instance per JVM.

2. Use the **ConfigurationBuilder** API to define cache configuration.
3. Obtain caches with **getCache()**, **createCache()**, or **getOrCreateCache()** methods.
   Data Grid recommends using the **getOrCreateCache()** method because it either creates a cache on all nodes or returns an existing cache.

4. If necessary use the **PERMANENT** flag for caches to survive restarts.

5. Stop the **CacheManager** by calling the **cacheManager.stop()** method to release JVM resources and gracefully shutdown any caches.

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

getCache() method

Invoke the **getCache(String)** method to obtain caches, as follows:

```
Cache<String, String> myCache = manager.getCache("myCache");
```

The preceding operation creates a cache named **myCache**, if it does not already exist, and returns it.

Using the **getCache()** method creates the cache only on the node where you invoke the method. In other words, it performs a local operation that must be invoked on each node across the cluster. Typically, applications deployed across multiple nodes obtain caches during initialization to ensure that caches are *symmetric* and exist on each node.

createCache() method

Invoke the **createCache()** method to create caches dynamically across the entire cluster.

```
Cache<String, String> myCache = manager.administration().createCache("myCache",
"myTemplate");
```

The preceding operation also automatically creates caches on any nodes that subsequently join the cluster.

Caches that you create with the **createCache()** method are ephemeral by default. If the entire cluster shuts down, the cache is not automatically created again when it restarts.

PERMANENT flag

Use the PERMANENT flag to ensure that caches can survive restarts.

```
Cache<String, String> myCache =
manager.administration().withFlags(AdminFlag.PERMANENT).createCache("myCache",
"myTemplate");
```

For the PERMANENT flag to take effect, you must enable global state and set a configuration storage provider.

For more information about configuration storage providers, see GlobalStateConfigurationBuilder#configurationStorage().

Additional resources

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

2.3. CACHE API

Data Grid provides a Cache interface that exposes simple methods for adding, retrieving and removing entries, including atomic mechanisms exposed by the JDK's ConcurrentMap interface. Based on the cache mode used, invoking these methods will trigger a number of things to happen, potentially even including replicating an entry to a remote node or looking up an entry from a remote node, or potentially a cache store.

For simple usage, using the Cache API should be no different from using the JDK Map API, and hence migrating from simple in-memory caches based on a Map to Data Grid's Cache should be trivial.

Performance Concerns of Certain Map Methods

Certain methods exposed in Map have certain performance consequences when used with Data Grid, such as size() , values() , keySet() and entrySet() . Specific methods on the **keySet**, **values** and **entrySet** are fine for use please see their Javadoc for further details.

Attempting to perform these operations globally would have large performance impact as well as become a scalability bottleneck. As such, these methods should only be used for informational or debugging purposes only.

It should be noted that using certain flags with the withFlags() method can mitigate some of these concerns, please check each method's documentation for more details.

Mortal and Immortal Data

Further to simply storing entries, Data Grid's cache API allows you to attach mortality information to data. For example, simply using put(key, value) would create an *immortal* entry, i.e., an entry that lives in the cache forever, until it is removed (or evicted from memory to prevent running out of memory). If, however, you put data in the cache using put(key, value, lifespan, timeunit) , this creates a *mortal* entry, i.e., an entry that has a fixed lifespan and expires after that lifespan.

In addition to *lifespan* , Data Grid also supports *maxIdle* as an additional metric with which to determine expiration. Any combination of lifespans or maxIdles can be used.

putForExternalRead operation

Data Grid's Cache class contains a different 'put' operation called putForExternalRead . This operation is particularly useful when Data Grid is used as a temporary cache for data that is persisted elsewhere. Under heavy read scenarios, contention in the cache should not delay the real transactions at hand, since caching should just be an optimization and not something that gets in the way.

To achieve this, **putForExternalRead()** acts as a put call that only operates if the key is not present in the cache, and fails fast and silently if another thread is trying to store the same key at the same time. In

this particular scenario, caching data is a way to optimise the system and it's not desirable that a failure in caching affects the on-going transaction, hence why failure is handled differently. **putForExternalRead()** is considered to be a fast operation because regardless of whether it's successful or not, it doesn't wait for any locks, and so returns to the caller promptly.

To understand how to use this operation, let's look at basic example. Imagine a cache of Person instances, each keyed by a PersonId , whose data originates in a separate data store. The following code shows the most common pattern of using putForExternalRead within the context of this example:

```
// Id of the person to look up, provided by the application
PersonId id = ...;

// Get a reference to the cache where person instances will be stored
Cache<PersonId, Person> cache = ...;

// First, check whether the cache contains the person instance
// associated with with the given id
Person cachedPerson = cache.get(id);

if (cachedPerson == null) {
  // The person is not cached yet, so query the data store with the id
  Person person = dataStore.lookup(id);

  // Cache the person along with the id so that future requests can
  // retrieve it from memory rather than going to the data store
  cache.putForExternalRead(id, person);
} else {
  // The person was found in the cache, so return it to the application
  return cachedPerson;
}
```

Note that putForExternalRead should never be used as a mechanism to update the cache with a new Person instance originating from application execution (i.e. from a transaction that modifies a Person's address). When updating cached values, please use the standard put operation, otherwise the possibility of caching corrupt data is likely.

2.3.1. AdvancedCache API

In addition to the simple Cache interface, Data Grid offers an AdvancedCache interface, geared towards extension authors. The AdvancedCache offers the ability to access certain internal components and to apply flags to alter the default behavior of certain cache methods. The following code snippet depicts how an AdvancedCache can be obtained:

```
AdvancedCache advancedCache = cache.getAdvancedCache();
```

2.3.1.1. Flags

Flags are applied to regular cache methods to alter the behavior of certain methods. For a list of all available flags, and their effects, see the Flag enumeration. Flags are applied using AdvancedCache.withFlags() . This builder method can be used to apply any number of flags to a cache invocation, for example:

```
advancedCache.withFlags(Flag.CACHE_MODE_LOCAL, Flag.SKIP_LOCKING)
  .withFlags(Flag.FORCE_SYNCHRONOUS)
  .put("hello", "world");
```

2.3.2. Asynchronous API

In addition to synchronous API methods like Cache.put() , Cache.remove() , etc., Data Grid also has an asynchronous, non-blocking API where you can achieve the same results in a non-blocking fashion.

These methods are named in a similar fashion to their blocking counterparts, with "Async" appended. E.g., Cache.putAsync() , Cache.removeAsync() , etc. These asynchronous counterparts return a CompletableFuture that contains the actual result of the operation.

For example, in a cache parameterized as **Cache<String, String>**, **Cache.put(String key, String value)** returns **String** while **Cache.putAsync(String key, String value)** returns **CompletableFuture<String>**.

2.3.2.1. Why use such an API?

Non-blocking APIs are powerful in that they provide all of the guarantees of synchronous communications - with the ability to handle communication failures and exceptions - with the ease of not having to block until a call completes. This allows you to better harness parallelism in your system. For example:

```
Set<CompletableFuture<?>> futures = new HashSet<>();
futures.add(cache.putAsync(key1, value1)); // does not block
futures.add(cache.putAsync(key2, value2)); // does not block
futures.add(cache.putAsync(key3, value3)); // does not block

// the remote calls for the 3 puts will effectively be executed
// in parallel, particularly useful if running in distributed mode
// and the 3 keys would typically be pushed to 3 different nodes
// in the cluster

// check that the puts completed successfully
for (CompletableFuture<?> f: futures) f.get();
```

2.3.2.2. Which processes actually happen asynchronously?

There are 4 things in Data Grid that can be considered to be on the critical path of a typical write operation. These are, in order of cost:

- network calls
- marshalling
- writing to a cache store (optional)
- locking

Using the async methods will take the network calls and marshalling off the critical path. For various technical reasons, writing to a cache store and acquiring locks, however, still happens in the caller's thread.

CHAPTER 3. PROGRAMMATICALLY CONFIGURING USER ROLES AND PERMISSIONS

Configure security authorization programmatically when using embedded caches in Java applications.

3.1. DATA GRID USER ROLES AND PERMISSIONS

Data Grid includes several roles that provide users with permissions to access caches and Data Grid resources.

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

Additional resources

- org.infinispan.security.AuthorizationPermission Enum
- Data Grid configuration schema reference

3.1.1. Permissions

User roles are sets of permissions with different access levels.

Table 3.1. Cache Manager permissions

Permission	Function	Description
CONFIGURATION	**defineConfiguration**	Defines new cache configurations.

LISTEN	**addListener**	Registers listeners against a Cache Manager.
LIFECYCLE	**stop**	Stops the Cache Manager.
CREATE	**createCache**, **removeCache**	Create and remove container resources such as caches, counters, schemas, and scripts.
MONITOR	**getStats**	Allows access to JMX statistics and the **metrics** endpoint.
ALL	-	Includes all Cache Manager permissions.

Table 3.2. Cache permissions

Permission	Function	Description
READ	**get**, **contains**	Retrieves entries from a cache.
WRITE	**put**, **putIfAbsent**, **replace**, **remove**, **evict**	Writes, replaces, removes, evicts data in a cache.
EXEC	**distexec**, **streams**	Allows code execution against a cache.
LISTEN	**addListener**	Registers listeners against a cache.
BULK_READ	**keySet**, **values**, **entrySet**, **query**	Executes bulk retrieve operations.
BULK_WRITE	**clear**, **putAll**	Executes bulk write operations.
LIFECYCLE	**start**, **stop**	Starts and stops a cache.

ADMIN	**getVersion**, **addInterceptor***, **removeInterceptor**, **getInterceptorChain**, **getEvictionManager**, **getComponentRegistry**, **getDistributionManager**, **getAuthorizationManager**, **evict**, **getRpcManager**, **getCacheConfiguration**, **getCacheManager**, **getInvocationContextContainer**, **setAvailability**, **getDataContainer**, **getStats**, **getXAResource**	Allows access to underlying components and internal structures.
MONITOR	**getStats**	Allows access to JMX statistics and the **metrics** endpoint.
ALL	-	Includes all cache permissions.
ALL_READ	-	Combines the READ and BULK_READ permissions.
ALL_WRITE	-	Combines the WRITE and BULK_WRITE permissions.

Additional resources

- Data Grid Security API

3.1.2. Role and permission mappers

Data Grid implements users as a collection of principals. Principals represent either an individual user identity, such as a username, or a group to which the users belong. Internally, these are implemented with the **javax.security.auth.Subject** class.

To enable authorization, the principals must be mapped to role names, which are then expanded into a set of permissions.

Data Grid includes the **PrincipalRoleMapper** API for associating security principals to roles, and the **RolePermissionMapper** API for associating roles with specific permissions.

Data Grid provides the following role and permission mapper implementations:

Cluster role mapper

Stores principal to role mappings in the cluster registry.

Cluster permission mapper

Stores role to permission mappings in the cluster registry. Allows you to dynamically modify user roles and permissions.

Identity role mapper

Uses the principal name as the role name. The type or format of the principal name depends on the source. For example, in an LDAP directory the principal name could be a Distinguished Name (DN).

Common name role mapper

Uses the Common Name (CN) as the role name. You can use this role mapper with an LDAP directory or with client certificates that contain Distinguished Names (DN); for example **cn=managers,ou=people,dc=example,dc=com** maps to the **managers** role.

3.1.2.1. Mapping users to roles and permissions in Data Grid

Consider the following user retrieved from an LDAP server, as a collection of DNs:

```
CN=myapplication,OU=applications,DC=mycompany
CN=dataprocessors,OU=groups,DC=mycompany
CN=finance,OU=groups,DC=mycompany
```

Using the **Common name role mapper**, the user would be mapped to the following roles:

```
dataprocessors
finance
```

Data Grid has the following role definitions:

```
dataprocessors: ALL_WRITE ALL_READ
finance: LISTEN
```

The user would have the following permissions:

```
ALL_WRITE ALL_READ LISTEN
```

Additional resources

- Data Grid Security API
- org.infinispan.security.PrincipalRoleMapper
- org.infinispan.security.RolePermissionMapper
- org.infinispan.security.mappers.IdentityRoleMapper
- org.infinispan.security.mappers.CommonNameRoleMapper

3.1.3. Configuring role mappers

Data Grid enables the cluster role mapper and cluster permission mapper by default. To use a different implementation for role mapping, you must configure the role mappers.

Procedure

1. Open your Data Grid configuration for editing.

2. Declare the role mapper as part of the security authorization in the Cache Manager configuration.

3. Save the changes to your configuration.

With embedded caches you can programmatically configure role and permission mappers with the **principalRoleMapper()** and **rolePermissionMapper()** methods.

Role mapper configuration

XML


```xml
<cache-container>
  <security>
    <authorization>
      <common-name-role-mapper />
    </authorization>
  </security>
</cache-container>
```


JSON


```json
{
  "infinispan" : {
    "cache-container" : {
      "security" : {
        "authorization" : {
          "common-name-role-mapper": {}
        }
      }
    }
  }
}
```


YAML


```yaml
infinispan:
  cacheContainer:
    security:
      authorization:
        commonNameRoleMapper: ~
```


Additional resources

- Data Grid configuration schema reference

3.2. ENABLING AND CONFIGURING AUTHORIZATION FOR EMBEDDED CACHES

When using embedded caches, you can configure authorization with the **GlobalSecurityConfigurationBuilder** and **ConfigurationBuilder** classes.

Procedure

1. Construct a **GlobalConfigurationBuilder** and enable security authorization with the **security().authorization().enable()** method.

2. Specify a role mapper with the **principalRoleMapper()** method.

3. If required, define custom role and permission mappings with the **role()** and **permission()** methods.

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
 global.security().authorization().enable()
         .principalRoleMapper(new ClusterRoleMapper())
         .role("myroleone").permission(AuthorizationPermission.ALL_WRITE)
         .role("myroletwo").permission(AuthorizationPermission.ALL_READ);
```

4. Enable authorization for caches in the **ConfigurationBuilder**.

   - Add all roles from the global configuration.

   ```
   ConfigurationBuilder config = new ConfigurationBuilder();
   config.security().authorization().enable();
   ```

   - Explicitly define roles for a cache so that Data Grid denies access for users who do not have the role.

   ```
   ConfigurationBuilder config = new ConfigurationBuilder();
   config.security().authorization().enable().role("myroleone");
   ```

Additional resources

- org.infinispan.configuration.global.GlobalSecurityConfigurationBuilder
- org.infinispan.configuration.cache.ConfigurationBuilder

3.3. ADDING AUTHORIZATION ROLES AT RUNTIME

Dynamically map roles to permissions when using security authorization with Data Grid caches.

Prerequisites

- Configure authorization for embedded caches.
- Have **ADMIN** permissions for Data Grid.

Procedure

1. Obtain the **RolePermissionMapper** instance.

2. Define new roles with the **addRole()** method.

```
MutableRolePermissionMapper mapper = (MutableRolePermissionMapper)
cacheManager.getCacheManagerConfiguration().security().authorization().rolePermissionMap
per();
```

```
mapper.addRole(Role.newRole("myroleone", true, AuthorizationPermission.ALL_WRITE,
AuthorizationPermission.LISTEN));
mapper.addRole(Role.newRole("myroletwo", true, AuthorizationPermission.READ,
AuthorizationPermission.WRITE));
```

Additional resources

- org.infinispan.security.RolePermissionMapper

3.4. EXECUTING CODE WITH SECURE CACHES

When you construct a **DefaultCacheManager** for an embedded cache that uses security authorization, the Cache Manager returns a **SecureCache** that checks the security context before invoking any operations. A **SecureCache** also ensures that applications cannot retrieve lower-level insecure objects such as **DataContainer**. For this reason, you must execute code with a Data Grid user that has a role with the appropriate level of permission.

Prerequisites

- Configure authorization for embedded caches.

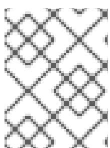
Procedure

1. If necessary, retrieve the current Subject from the Data Grid context or **AccessControlContext**:

```
Security.getSubject();
```

2. Wrap method calls in a **PrivilegedAction** to execute them with the Subject.

```
Security.doAs(mySubject, (PrivilegedAction<String>)() -> cache.put("key", "value"));
```



NOTE

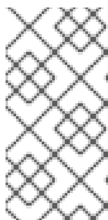
You can use the **Security.doAs()** or **Subject.doAs()** method. Data Grid recommends **Security.doAs()** for better performance.

Additional resources

- org.infinispan.security.Security
- org.infinispan.security.SecureCache

3.5. CONFIGURING THE ACCESS CONTROL LIST (ACL) CACHE

When you grant or deny roles to users, Data Grid stores details about which users can access your caches internally. This ACL cache improves performance for security authorization by avoiding the need for Data Grid to calculate if users have the appropriate permissions to perform read and write operations for every request.

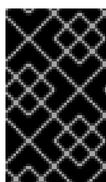


NOTE

Whenever you grant or deny roles to users, Data Grid flushes the ACL cache to ensure it applies user permissions correctly. This means that Data Grid must recalculate cache permissions for all users each time you grant or deny roles. For best performance you should not frequently or repeatedly grant and deny roles in production environments.

Procedure

1. Open your Data Grid configuration for editing.
2. Specify the maximum number of entries for the ACL cache with the **cache-size** attribute. Entries in the ACL cache have a cardinality of **caches * users**. You should set the maximum number of entries to a value that can hold information for all your caches and users. For example, the default size of **1000** is appropriate for deployments with up to 100 caches and 10 users.
3. Set the timeout value, in milliseconds, with the **cache-timeout** attribute. If Data Grid does not access an entry in the ACL cache within the timeout period that entry is evicted. When the user subsequently attempts cache operations then Data Grid recalculates their cache permissions and adds an entry to the ACL cache.



IMPORTANT

Specifying a value of **0** for either the **cache-size** or **cache-timeout** attribute disables the ACL cache. You should disable the ACL cache only if you disable authorization.

4. Save the changes to your configuration.

ACL cache configuration

XML

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

JSON

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

```
      }
    }
  }
}
```

YAML

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

Additional resources

- Data Grid configuration schema reference

CHAPTER 4. ENABLING AND CONFIGURING DATA GRID STATISTICS AND JMX MONITORING

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

4.1. ENABLING STATISTICS IN EMBEDDED CACHES

Configure Data Grid to export statistics for the Cache Manager and embedded caches.

Procedure

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

Embedded cache statistics

XML

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

GlobalConfigurationBuilder

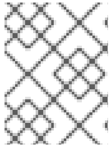
```java
GlobalConfigurationBuilder global =
GlobalConfigurationBuilder.defaultClusteredBuilder().cacheContainer().statistics(true);
DefaultCacheManager cacheManager = new DefaultCacheManager(global.build());

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

4.2. CONFIGURING DATA GRID METRICS

Data Grid generates metrics that are compatible with any monitoring system.

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

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



NOTE

Data Grid metrics are provided at the **vendor** scope. Metrics related to the JVM are provided in the **base** scope.

Prerequisites

- You must add Micrometer Core and Micrometer Registry Prometheus JARs to your classpath to export Data Grid metrics for embedded caches.

Procedure

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

Metrics configuration

XML

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

JSON

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

YAML

```yaml
infinispan:
  cacheContainer:
    statistics: "true"
```

```
metrics:
  gauges: "true"
  histograms: "true"
```

GlobalConfigurationBuilder

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

Additional resources

- Micrometer Prometheus

4.3. REGISTERING JMX MBEANS

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

Procedure

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

JMX configuration

XML

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

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
```

```
      "jmx" : {
        "enabled" : "true",
        "domain" : "example.com"
      }
    }
  }
}
```

YAML

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

GlobalConfigurationBuilder

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

4.3.1. Enabling JMX remote ports

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

You can enable remote JMX ports using one of the following approaches:

- Enable remote JMX ports that require authentication to one of the Data Grid Server security realms.
- Enable remote JMX ports manually using the standard Java management configuration options.

Prerequisites

- For remote JMX with authentication, define JMX specific user roles using the default security realm. Users must have **controlRole** with read/write access or the **monitorRole** with read-only access to access any JMX resources.

Procedure

Start Data Grid Server with a remote JMX port enabled using one of the following ways:

- Enable remote JMX through port **9999**.

  ```
  bin/server.sh --jmx 9999
  ```

WARNING

Using remote JMX with SSL disabled is not intended for production environments.

- Pass the following system properties to Data Grid Server at startup.

```
bin/server.sh -Dcom.sun.management.jmxremote.port=9999 -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl=false
```

WARNING

Enabling remote JMX with no authentication or SSL is not secure and not recommended in any environment. Disabling authentication and SSL allows unauthorized users to connect to your server and access the data hosted there.

Additional resources

- Creating security realms

4.3.2. Data Grid MBeans

Data Grid exposes JMX MBeans that represent manageable resources.

org.infinispan:type=Cache
: Attributes and operations available for cache instances.

org.infinispan:type=CacheManager
: Attributes and operations available for Cache Managers, including Data Grid cache and cluster health statistics.

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

Additional resources

- Data Grid JMX Components

4.3.3. Registering MBeans in custom MBean servers

Data Grid includes an **MBeanServerLookup** interface that you can use to register MBeans in custom MBeanServer instances.

Prerequisites

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

Procedure

1. Open your Data Grid configuration for editing.
2. Add the **mbean-server-lookup** attribute or field to the JMX configuration for the Cache Manager.
3. Specify fully qualified name (FQN) of your **MBeanServerLookup** implementation.
4. Save and close your client configuration.

JMX MBean server lookup configuration

XML

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

JSON

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

YAML

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

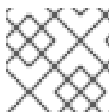
GlobalConfigurationBuilder

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

4.4. EXPORTING METRICS DURING A STATE TRANSFER OPERATION

You can export time metrics for clustered caches that Data Grid redistributes across nodes.

A state transfer operation occurs when a clustered cache topology changes, such as a node joining or leaving a cluster. During a state transfer operation, Data Grid exports metrics from each cache, so that you can determine a cache’s status. A state transfer exposes attributes as properties, so that Data Grid can export metrics from each cache.



NOTE

You cannot perform a state transfer operation in invalidation mode.

Data Grid generates time metrics that are compatible with the REST API and the JMX API.

Prerequisites

- Configure Data Grid metrics.
- Enable metrics for your cache type, such as embedded cache or remote cache.
- Initiate a state transfer operation by changing your clustered cache topology.

Procedure

- Choose one of the following methods:
  - Configure Data Grid to use the REST API to collect metrics.
  - Configure Data Grid to use the JMX API to collect metrics.

Additional resources

- Enabling and configuring Data Grid statistics and JMX monitoring (Data Grid caches)
- **StateTransferManager** (Data Grid 14.0 API)

4.5. MONITORING THE STATUS OF CROSS-SITE REPLICATION

Monitor the site status of your backup locations to detect interruptions in the communication between the sites. When a remote site status changes to **offline**, Data Grid stops replicating your data to the backup location. Your data become out of sync and you must fix the inconsistencies before bringing the clusters back online.

Monitoring cross-site events is necessary for early problem detection. Use one of the following monitoring strategies:

- Monitoring cross-site replication with the REST API
- Monitoring cross-site replication with the Prometheus metrics or any other monitoring system

Monitoring cross-site replication with the REST API

Monitor the status of cross-site replication for all caches using the REST endpoint. You can implement a custom script to poll the REST endpoint or use the following example.

Prerequisites

- Enable cross-site replication.

Procedure

1. Implement a script to poll the REST endpoint.
   The following example demonstrates how you can use a Python script to poll the site status every five seconds.

```
#!/usr/bin/python3
import time
import requests
from requests.auth import HTTPDigestAuth


class InfinispanConnection:

    def __init__(self, server: str = 'http://localhost:11222', cache_manager: str = 'default',
                 auth: tuple = ('admin', 'change_me')) -> None:
        super().__init__()
        self.__url = f'{server}/rest/v2/cache-managers/{cache_manager}/x-site/backups/'
        self.__auth = auth
        self.__headers = {
            'accept': 'application/json'
        }

    def get_sites_status(self):
        try:
            rsp = requests.get(self.__url, headers=self.__headers, auth=HTTPDigestAuth(self.__auth[0],
self.__auth[1]))
            if rsp.status_code != 200:
                return None
            return rsp.json()
        except:
            return None


# Specify credentials for Data Grid user with permission to access the REST endpoint
USERNAME = 'admin'
PASSWORD = 'change_me'
# Set an interval between cross-site status checks
POLL_INTERVAL_SEC = 5
# Provide a list of servers
SERVERS = [
```

```
    InfinispanConnection('http://127.0.0.1:11222', auth=(USERNAME, PASSWORD)),
    InfinispanConnection('http://127.0.0.1:12222', auth=(USERNAME, PASSWORD))
]
#Specify the names of remote sites
REMOTE_SITES = [
    'nyc'
]
#Provide a list of caches to monitor
CACHES = [
    'work',
    'sessions'
]


def on_event(site: str, cache: str, old_status: str, new_status: str):
    # TODO implement your handling code here
    print(f'site={site} cache={cache} Status changed {old_status} -> {new_status}')


def __handle_mixed_state(state: dict, site: str, site_status: dict):
    if site not in state:
        state[site] = {c: 'online' if c in site_status['online'] else 'offline' for c in CACHES}
        return

    for cache in CACHES:
        __update_cache_state(state, site, cache, 'online' if cache in site_status['online'] else 'offline')


def __handle_online_or_offline_state(state: dict, site: str, new_status: str):
    if site not in state:
        state[site] = {c: new_status for c in CACHES}
        return

    for cache in CACHES:
        __update_cache_state(state, site, cache, new_status)


def __update_cache_state(state: dict, site: str, cache: str, new_status: str):
    old_status = state[site].get(cache)
    if old_status != new_status:
        on_event(site, cache, old_status, new_status)
        state[site][cache] = new_status


def update_state(state: dict):
    rsp = None
    for conn in SERVERS:
        rsp = conn.get_sites_status()
        if rsp:
            break
    if rsp is None:
        print('Unable to fetch site status from any server')
        return

    for site in REMOTE_SITES:
        site_status = rsp.get(site, {})
```

```
        new_status = site_status.get('status')
        if new_status == 'mixed':
            __handle_mixed_state(state, site, site_status)
        else:
            __handle_online_or_offline_state(state, site, new_status)


if __name__ == '__main__':
    _state = {}
    while True:
        update_state(_state)
        time.sleep(POLL_INTERVAL_SEC)
```

When a site status changes from **online** to **offline** or vice-versa, the function **on_event** is invoked.

If you want to use this script, you must specify the following variables:

- **USERNAME** and **PASSWORD**: The username and password of Data Grid user with permission to access the REST endpoint.
- **POLL_INTERVAL_SEC**: The number of seconds between polls.
- **SERVERS**: The list of Data Grid Servers at this site. The script only requires a single valid response but the list is provided to allow fail over.
- **REMOTE_SITES**: The list of remote sites to monitor on these servers.
- **CACHES**: The list of cache names to monitor.

Additional resources

- REST API: Getting status of backup locations

Monitoring cross-site replication with the Prometheus metrics

Prometheus, and other monitoring systems, let you configure alerts to detect when a site status changes to **offline**.

TIP

Monitoring cross-site latency metrics can help you to discover potential issues.

Prerequisites

- Enable cross-site replication.

Procedure

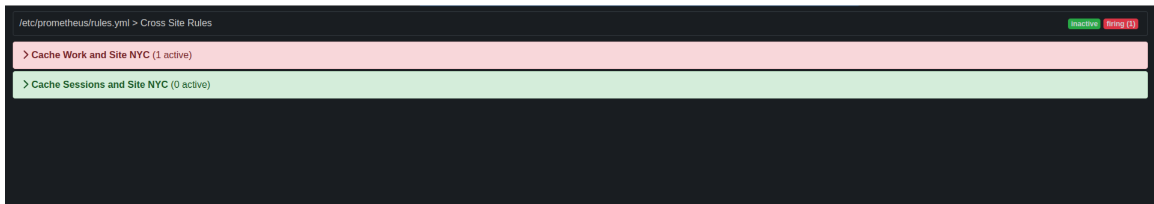
1. Configure Data Grid metrics.
2. Configure alerting rules using the Prometheus metrics format.
    - For the site status, use **1** for **online** and **0** for **offline**.
    - For the **expr** filed, use the following format: **vendor_cache_manager_default_cache_<cache name>_x_site_admin_<site name>_status**.

In the following example, Prometheus alerts you when the **NYC** site gets **offline** for cache named **work** or **sessions**.

```
groups:
- name: Cross Site Rules
  rules:
  - alert: Cache Work and Site NYC
    expr: vendor_cache_manager_default_cache_work_x_site_admin_nyc_status == 0
  - alert: Cache Sessions and Site NYC
    expr: vendor_cache_manager_default_cache_sessions_x_site_admin_nyc_status ==
0
```

The following image shows an alert that the **NYC** site is **offline** for cache **work**.

Figure 4.1. Prometheus Alert



Additional resources

- Configuring Data Grid metrics
- Prometheus Alerting Overview
- Grafana Alerting Documentation
- Openshift Managing Alerts

CHAPTER 5. SETTING UP DATA GRID CLUSTER TRANSPORT

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

5.1. DEFAULT JGROUPS STACKS

Data Grid provides default JGroups stack files, **default-jgroups-*.xml**, in the **default-configs** directory inside the **infinispan-core-14.0.21.Final-redhat-00001.jar** file.

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

Additional resources

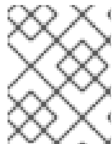
- JGroups Protocols

5.2. CLUSTER DISCOVERY PROTOCOLS

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

There are two types of discovery mechanisms that Data Grid can use:

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



NOTE

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

Additional resources

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

5.2.1. PING

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

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

PING configuration example

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

Additional resources

- JGroups PING

5.2.2. TCPPING

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

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

TCPPING configuration example

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

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

Additional resources

- JGroups TCPPING

5.2.3. MPING

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

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

MPING configuration example

```
<MPING mcast_addr="${jgroups.mcast_addr:239.6.7.8}"
       mcast_port="${jgroups.mcast_port:46655}"
       num_discovery_runs="3"
       ip_ttl="${jgroups.udp.ip_ttl:2}"/>
```

Additional resources

- JGroups MPING

5.2.4. TCPGOSSIP

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

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

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

Gossip router configuration example

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

Additional resources

- JGroups Gossip Router

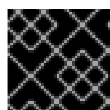
5.2.5. JDBC_PING

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

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

JDBC_PING configuration example

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



IMPORTANT

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

Additional resources

- JDBC_PING
- JDBC_PING Wiki

5.2.6. DNS_PING

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

DNS_PING configuration example

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

Additional resources

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

5.2.7. Cloud discovery protocols

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

Discovery protocol	Default stack file	Artifact	Version
aws.S3_PING	default-jgroups-ec2.xml	org.jgroups.aws:jgroups-aws	2.0.1.Final
GOOGLE_PING2	default-jgroups-google.xml	org.jgroups.google:jgroups-google	1.0.0.Final
azure.AZURE_PING	default-jgroups-azure.xml	org.jgroups.azure:jgroups-azure	2.0.0.Final

Providing dependencies for cloud discovery protocols
To use **aws.S3_PING**, **GOOGLE_PING2**, or **azure.AZURE_PING** cloud discovery protocols, you need to provide dependent libraries to Data Grid.

Procedure

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

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

Additional resources

- JGroups aws.S3_PING
- JGroups GOOGLE_PING2
- JGroups azure.AZURE_PING

5.3. USING THE DEFAULT JGROUPS STACKS

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

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

Procedure

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

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

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

- Use the **addProperty()** method to set the JGroups stack file:

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

Verification

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

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

Additional resources

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

5.4. CUSTOMIZING JGROUPS STACKS

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

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

Procedure

1. Create a new JGroups stack declaration in your **infinispan.xml** file.
2. Add the **extends** attribute and specify a JGroups stack to inherit properties from.
3. Use the **stack.combine** attribute to modify properties for protocols configured in the inherited stack.
4. Use the **stack.position** attribute to define the location for your custom stack.
5. Specify the stack name as the value for the **stack** attribute in the **transport** configuration.
   For example, you might evaluate using a Gossip router and symmetric encryption with the default TCP stack as follows:

```
<infinispan>
 <jgroups>
   <!-- Creates a custom JGroups stack named "my-stack". -->
   <!-- Inherits properties from the default TCP stack. -->
   <stack name="my-stack" extends="tcp">
     <!-- Uses TCPGOSSIP as the discovery mechanism instead of MPING -->
     <TCPGOSSIP initial_hosts="${jgroups.tunnel.gossip_router_hosts:localhost[12001]}"
          stack.combine="REPLACE"
          stack.position="MPING" />
     <!-- Removes the FD_SOCK2 protocol from the stack. -->
     <FD_SOCK2 stack.combine="REMOVE"/>
     <!-- Modifies the timeout value for the VERIFY_SUSPECT2 protocol. -->
     <VERIFY_SUSPECT2 timeout="2000"/>
     <!-- Adds SYM_ENCRYPT to the stack after VERIFY_SUSPECT2. -->
     <SYM_ENCRYPT sym_algorithm="AES"
              keystore_name="mykeystore.p12"
              keystore_type="PKCS12"
              store_password="changeit"
              key_password="changeit"
              alias="myKey"
              stack.combine="INSERT_AFTER"
              stack.position="VERIFY_SUSPECT2" />
   </stack>
 </jgroups>
```

```
  <cache-container name="default" statistics="true">
   <!-- Uses "my-stack" for cluster transport. -->
   <transport cluster="${infinispan.cluster.name}"
          stack="my-stack"
          node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

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

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

Reference

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

5.4.1. Inheritance attributes

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

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

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

5.5. USING JGROUPS SYSTEM PROPERTIES

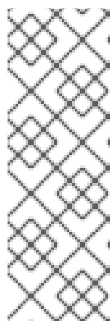
Pass system properties to Data Grid at startup to tune cluster transport.

Procedure

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

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

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



NOTE

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

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

5.5.1. Cluster transport properties

Use the following properties to customize JGroups cluster transport.

System Property	Description	Default Value	Required/ Optional
jgroups.bind.address	Bind address for cluster transport.	**SITE_LOCAL**	Optional
jgroups.bind.port	Bind port for the socket.	**7800**	Optional
jgroups.mcast_addr	IP address for multicast, both discovery and inter-cluster communication. The IP address must be a valid "class D" address that is suitable for IP multicast.	**239.6.7.8**	Optional
jgroups.mcast_port	Port for the multicast socket.	**46655**	Optional
jgroups.ip_ttl	Time-to-live (TTL) for IP multicast packets. The value defines the number of network hops a packet can make before it is dropped.	2	Optional
jgroups.thread_pool.min_threads	Minimum number of threads for the thread pool.	0	Optional

System Property	Description	Default Value	Required/ Optional
jgroups.thread_pool.max_threads	Maximum number of threads for the thread pool.	200	Optional
jgroups.join_timeout	Maximum number of milliseconds to wait for join requests to succeed.	2000	Optional
jgroups.thread_dumps_threshold	Number of times a thread pool needs to be full before a thread dump is logged.	10000	Optional
jgroups.fd.port-offset	Offset from **jgroups.bind.port** port for the FD (failure detection protocol) socket.	**50000** (port **57800**)	Optional
jgroups.frag_size	Maximum number of bytes in a message. Messages larger than that are fragmented.	60000	Optional
jgroups.diag.enabled	Enables JGroups diagnostic probing.	false	Optional

Additional resources

- JGroups system properties
- JGroups protocol list

5.5.2. System properties for cloud discovery protocols

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

5.5.2.1. Amazon EC2

System properties for configuring **aws.S3_PING**.

System Property	Description	Default Value	Required/ Optional
jgroups.s3.region_name	Name of the Amazon S3 region.	No default value.	Optional

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

5.5.2.2. Google Cloud Platform

System properties for configuring **GOOGLE_PING2**.

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

5.5.2.3. Azure

System properties for azure.AZURE_PING`.

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

5.5.2.4. OpenShift

System properties for **DNS_PING**.

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

5.6. USING INLINE JGROUPS STACKS

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

Procedure

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

```
<infinispan>
  <!-- Contains one or more JGroups stack definitions. -->
  <jgroups>
    <!-- Defines a custom JGroups stack named "prod". -->
    <stack name="prod">
      <TCP bind_port="7800" port_range="30" recv_buf_size="20000000"
send_buf_size="640000"/>
      <RED/>
      <MPING break_on_coord_rsp="true"
             mcast_addr="${jgroups.mping.mcast_addr:239.2.4.6}"
             mcast_port="${jgroups.mping.mcast_port:43366}"
             num_discovery_runs="3"
             ip_ttl="${jgroups.udp.ip_ttl:2}"/>
      <MERGE3 />
      <FD_SOCK2 />
      <FD_ALL3 timeout="3000" interval="1000" timeout_check_interval="1000" />
      <VERIFY_SUSPECT2 timeout="1000" />
      <pbcast.NAKACK2 use_mcast_xmit="false" xmit_interval="200"
xmit_table_num_rows="50"
                      xmit_table_msgs_per_row="1024" xmit_table_max_compaction_time="30000"
/>
      <UNICAST3 conn_close_timeout="5000" xmit_interval="200" xmit_table_num_rows="50"
                xmit_table_msgs_per_row="1024" xmit_table_max_compaction_time="30000" />
      <pbcast.STABLE desired_avg_gossip="2000" max_bytes="1M" />
      <pbcast.GMS print_local_addr="false" join_timeout="${jgroups.join_timeout:2000}" />
      <UFC max_credits="4m" min_threshold="0.40" />
      <MFC max_credits="4m" min_threshold="0.40" />
      <FRAG4 />
    </stack>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Uses "prod" for cluster transport. -->
    <transport cluster="${infinispan.cluster.name}"
           stack="prod"
```

```
        node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

5.7. USING EXTERNAL JGROUPS STACKS

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

Procedure

1. Put custom JGroups stack files on the application classpath.
   Alternatively you can specify an absolute path when you declare the external stack file.

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

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

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

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

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

Custom UDP stack example

```
<config xmlns="urn:org:jgroups"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:org:jgroups http://www.jgroups.org/schema/jgroups-4.2.xsd">
    <UDP bind_addr="${jgroups.bind_addr:127.0.0.1}"
         mcast_addr="${jgroups.udp.mcast_addr:239.0.2.0}"
         mcast_port="${jgroups.udp.mcast_port:46655}"
         tos="8"
         ucast_recv_buf_size="20000000"
         ucast_send_buf_size="640000"
         mcast_recv_buf_size="25000000"
         mcast_send_buf_size="640000"
```

```
        bundler.max_size="64000"
        ip_ttl="${jgroups.udp.ip_ttl:2}"
        diag.enabled="false"
        thread_naming_pattern="pl"
        thread_pool.enabled="true"
        thread_pool.min_threads="2"
        thread_pool.max_threads="30"
        thread_pool.keep_alive_time="5000" />
    <!-- Other JGroups stack configuration goes here. -->
</config>
```

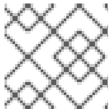
Additional resources

- org.infinispan.configuration.global.TransportConfigurationBuilder

5.8. USING CUSTOM JCHANNELS

Construct custom JGroups JChannels as in the following example:

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



NOTE

Data Grid cannot use custom JChannels that are already connected.

Additional resources

- JGroups JChannel

5.9. ENCRYPTING CLUSTER TRANSPORT

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

5.9.1. JGroups encryption protocols

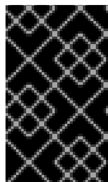
To secure cluster traffic, you can configure Data Grid nodes to encrypt JGroups message payloads with secret keys.

Data Grid nodes can obtain secret keys from either:

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

Retrieving secret keys from coordinator nodes

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



IMPORTANT

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

Asymmetric encryption secures cluster traffic as follows:

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

Retrieving secret keys from shared keystores

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

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

Comparison of asymmetric and symmetric encryption

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

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

5.9.2. Securing cluster transport with asymmetric encryption

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

Procedure

1. Create a keystore with certificate chains that enables Data Grid to verify node identity.
2. Place the keystore on the classpath for each node in the cluster.
For Data Grid Server, you put the keystore in the $RHDG_HOME directory.

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

```
<infinispan>
  <jgroups>
    <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the default TCP
stack. -->
    <stack name="encrypt-tcp" extends="tcp">
      <!-- Adds a keystore that nodes use to perform certificate authentication. -->
      <!-- Uses the stack.combine and stack.position attributes to insert
SSL_KEY_EXCHANGE into the default TCP stack after VERIFY_SUSPECT2. -->
      <SSL_KEY_EXCHANGE keystore_name="mykeystore.jks"
                        keystore_password="changeit"
                        stack.combine="INSERT_AFTER"
                        stack.position="VERIFY_SUSPECT2"/>
      <!-- Configures ASYM_ENCRYPT -->
      <!-- Uses the stack.combine and stack.position attributes to insert ASYM_ENCRYPT into
the default TCP stack before pbcast.NAKACK2. -->
      <!-- The use_external_key_exchange = "true" attribute configures nodes to use the
`SSL_KEY_EXCHANGE` protocol for certificate authentication. -->
      <ASYM_ENCRYPT asym_keylength="2048"
                    asym_algorithm="RSA"
                    change_key_on_coord_leave = "false"
                    change_key_on_leave = "false"
                    use_external_key_exchange = "true"
                    stack.combine="INSERT_BEFORE"
                    stack.position="pbcast.NAKACK2"/>
    </stack>
  </jgroups>
  <cache-container name="default" statistics="true">
    <!-- Configures the cluster to use the JGroups stack. -->
    <transport cluster="${infinispan.cluster.name}"
               stack="encrypt-tcp"
               node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

Verification

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

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

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

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

Additional resources

- JGroups 4 Manual

- JGroups 4.2 Schema

5.9.3. Securing cluster transport with symmetric encryption

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

Procedure

1. Create a keystore that contains a secret key.
2. Place the keystore on the classpath for each node in the cluster.
   For Data Grid Server, you put the keystore in the $RHDG_HOME directory.
3. Add the **SYM_ENCRYPT** protocol to a JGroups stack in your Data Grid configuration.

```
<infinispan>
  <jgroups>
    <!-- Creates a secure JGroups stack named "encrypt-tcp" that extends the default TCP stack. -->
    <stack name="encrypt-tcp" extends="tcp">
      <!-- Adds a keystore from which nodes obtain secret keys. -->
      <!-- Uses the stack.combine and stack.position attributes to insert SYM_ENCRYPT into the
default TCP stack after VERIFY_SUSPECT2. -->
      <SYM_ENCRYPT keystore_name="myKeystore.p12"
                   keystore_type="PKCS12"
                   store_password="changeit"
                   key_password="changeit"
                   alias="myKey"
                   stack.combine="INSERT_AFTER"
                   stack.position="VERIFY_SUSPECT2"/>
    </stack>
  </jgroups>
  <cache-container name="default" statistics="true">
    <!-- Configures the cluster to use the JGroups stack. -->
    <transport cluster="${infinispan.cluster.name}"
               stack="encrypt-tcp"
               node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

Verification

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

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

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

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

Additional resources

Additional resources

- JGroups 4 Manual
- JGroups 4.2 Schema

5.10. TCP AND UDP PORTS FOR CLUSTER TRAFFIC

Data Grid uses the following ports for cluster transport messages:

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

Cross-site replication

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

7900

For Data Grid clusters running on OpenShift.

7800

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

7801

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

CHAPTER 6. CLUSTERED LOCKS

Clustered locks are data structures that are distributed and shared across nodes in a Data Grid cluster. Clustered locks allow you to run code that is synchronized between nodes.

6.1. LOCK API

Data Grid provides a **ClusteredLock** API that lets you concurrently execute code on a cluster when using Data Grid in embedded mode.

The API consists of the following:

- **ClusteredLock** exposes methods to implement clustered locks.
- **ClusteredLockManager** exposes methods to define, configure, retrieve, and remove clustered locks.
- **EmbeddedClusteredLockManagerFactory** initializes **ClusteredLockManager** implementations.

Ownership

Data Grid supports **NODE** ownership so that all nodes in a cluster can use a lock.

Reentrancy

Data Grid clustered locks are non-reentrant so any node in the cluster can acquire a lock but only the node that creates the lock can release it.

If two consecutive lock calls are sent for the same owner, the first call acquires the lock if it is available and the second call is blocked.

Reference

- EmbeddedClusteredLockManagerFactory
- ClusteredLockManager
- ClusteredLock

6.2. USING CLUSTERED LOCKS

Learn how to use clustered locks with Data Grid embedded in your application.

Prerequisites

- Add the **infinispan-clustered-lock** dependency to your **pom.xml**:

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-lock</artifactId>
</dependency>
```

Procedure

1. Initialize the **ClusteredLockManager** interface from a Cache Manager. This interface is the entry point for defining, retrieving, and removing clustered locks.
2. Give a unique name for each clustered lock.
3. Acquire locks with the **lock.tryLock(1, TimeUnit.SECONDS)** method.

```
// Set up a clustered Cache Manager.
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();

// Configure the cache mode, in this case it is distributed and synchronous.
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC);

// Initialize a new default Cache Manager.
DefaultCacheManager cm = new DefaultCacheManager(global.build(), builder.build());

// Initialize a Clustered Lock Manager.
ClusteredLockManager clm1 = EmbeddedClusteredLockManagerFactory.from(cm);

// Define a clustered lock named 'lock'.
clm1.defineLock("lock");

// Get a lock from each node in the cluster.
ClusteredLock lock = clm1.get("lock");

AtomicInteger counter = new AtomicInteger(0);

// Acquire the lock as follows.
// Each 'lock.tryLock(1, TimeUnit.SECONDS)' method attempts to acquire the lock.
// If the lock is not available, the method waits for the timeout period to elapse. When the lock is
acquired, other calls to acquire the lock are blocked until the lock is released.
CompletableFuture<Boolean> call1 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r, ex) -> {
   if (r) {
      System.out.println("lock is acquired by the call 1");
      lock.unlock().whenComplete((nil, ex2) -> {
         System.out.println("lock is released by the call 1");
         counter.incrementAndGet();
      });
   }
});

CompletableFuture<Boolean> call2 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r, ex) -> {
   if (r) {
      System.out.println("lock is acquired by the call 2");
      lock.unlock().whenComplete((nil, ex2) -> {
         System.out.println("lock is released by the call 2");
         counter.incrementAndGet();
      });
   }
});

CompletableFuture<Boolean> call3 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r, ex) -> {
   if (r) {
      System.out.println("lock is acquired by the call 3");
      lock.unlock().whenComplete((nil, ex2) -> {
```

```
            System.out.println("lock is released by the call 3");
            counter.incrementAndGet();
        });
    }
});

CompletableFuture.allOf(call1, call2, call3).whenComplete((r, ex) -> {
    // Print the value of the counter.
    System.out.println("Value of the counter is " + counter.get());

    // Stop the Cache Manager.
    cm.stop();
});
```

6.3. CONFIGURING INTERNAL CACHES FOR LOCKS

Clustered Lock Managers include an internal cache that stores lock state. You can configure the internal cache either declaratively or programmatically.

Procedure

1. Define the number of nodes in the cluster that store the state of clustered locks. The default value is **-1**, which replicates the value to all nodes.
2. Specify one of the following values for the cache reliability, which controls how clustered locks behave when clusters split into partitions or multiple nodes leave:
    - **AVAILABLE**: Nodes in any partition can concurrently operate on locks.
    - **CONSISTENT**: Only nodes that belong to the majority partition can operate on locks. This is the default value.
    - Programmatic configuration

```
import org.infinispan.lock.configuration.ClusteredLockManagerConfiguration;
import org.infinispan.lock.configuration.ClusteredLockManagerConfigurationBuilder;
import org.infinispan.lock.configuration.Reliability;
...

GlobalConfigurationBuilder global =
GlobalConfigurationBuilder.defaultClusteredBuilder();

final ClusteredLockManagerConfiguration config =
global.addModule(ClusteredLockManagerConfigurationBuilder.class).numOwner(2).relia
bility(Reliability.AVAILABLE).create();

DefaultCacheManager cm = new DefaultCacheManager(global.build());

ClusteredLockManager clm1 = EmbeddedClusteredLockManagerFactory.from(cm);

clm1.defineLock("lock");
```

    - Declarative configuration

```
<infinispan
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:infinispan:config:14.0
https://infinispan.org/schemas/infinispan-config-14.0.xsd"
        xmlns="urn:infinispan:config:14.0">

    <cache-container default-cache="default">
        <transport/>
        <local-cache name="default">
            <locking concurrency-level="100" acquire-timeout="1000"/>
        </local-cache>
        <clustered-locks xmlns="urn:infinispan:config:clustered-locks:14.0"
                         num-owners = "3"
                         reliability="AVAILABLE">
            <clustered-lock name="lock1" />
            <clustered-lock name="lock2" />
        </clustered-locks>
    </cache-container>
    <!-- Cache configuration goes here. -->
</infinispan>
```

Reference

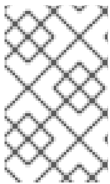
- ClusteredLockManagerConfiguration
- Clustered Locks Configuration Schema

CHAPTER 7. EXECUTING CODE IN THE GRID

The main benefit of a cache is the ability to very quickly lookup a value by its key, even across machines. In fact this use alone is probably the reason many users use Data Grid. However Data Grid can provide many more benefits that aren't immediately apparent. Since Data Grid is usually used in a cluster of machines we also have features available that can help utilize the entire cluster for performing the user's desired workload.

7.1. CLUSTER EXECUTOR

Since you have a group of machines, it makes sense to leverage their combined computing power for executing code on all of them them. The Cache Manager comes with a nice utility that allows you to execute arbitrary code in the cluster. Note this feature requires no Cache to be used. This Cluster Executor can be retrieved by calling executor() on the **EmbeddedCacheManager**. This executor is retrievable in both clustered and non clustered configurations.



NOTE

The ClusterExecutor is specifically designed for executing code where the code is not reliant upon the data in a cache and is used instead as a way to help users to execute code easily in the cluster.

This manager was built specifically using Java 8 and such has functional APIs in mind, thus all methods take a functional interface as an argument. Also since these arguments will be sent to other nodes they need to be serializable. We even used a nice trick to ensure our lambdas are immediately Serializable. That is by having the arguments implement both Serializable and the real argument type (ie. Runnable or Function). The JRE will pick the most specific class when determining which method to invoke, so in that case your lambdas will always be serializable. It is also possible to use an Externalizer to possibly reduce message size further.

The manager by default will submit a given command to all nodes in the cluster including the node where it was submitted from. You can control on which nodes the task is executed on by using the **filterTargets** methods as is explained in the section.

7.1.1. Filtering execution nodes

It is possible to limit on which nodes the command will be ran. For example you may want to only run a computation on machines in the same rack. Or you may want to perform an operation once in the local site and again on a different site. A cluster executor can limit what nodes it sends requests to at the scope of same or different machine, rack or site level.

SameRack.java

```
EmbeddedCacheManager manager = ...;
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_RACK).submit(...)
```

To use this topology base filtering you must enable topology aware consistent hashing through Server Hinting.

You can also filter using a predicate based on the **Address** of the node. This can also be optionally combined with topology based filtering in the previous code snippet.

We also allow the target node to be chosen by any means using a **Predicate** that will filter out which nodes can be considered for execution. Note this can also be combined with Topology filtering at the same time to allow even more fine control of where you code is executed within the cluster.

Predicate.java

```
EmbeddedCacheManager manager = ...;
// Just filter
manager.executor().filterTargets(a -> a.equals(..)).submit(...)
// Filter only those in the desired topology
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_SITE, a -> a.equals(..)).submit(...)
```

7.1.2. Timeout

Cluster Executor allows for a timeout to be set per invocation. This defaults to the distributed sync timeout as configured on the Transport Configuration. This timeout works in both a clustered and non clustered Cache Manager. The executor may or may not interrupt the threads executing a task when the timeout expires. However when the timeout occurs any **Consumer** or **Future** will be completed passing back a **TimeoutException**. This value can be overridden by ivoking the timeout method and supplying the desired duration.

7.1.3. Single Node Submission

Cluster Executor can also run in single node submission mode instead of submitting the command to all nodes it will instead pick one of the nodes that would have normally received the command and instead submit it it to only one. Each submission will possibly use a different node to execute the task on. This can be very useful to use the ClusterExecutor as a **java.util.concurrent.Executor** which you may have noticed that ClusterExecutor implements.

SingleNode.java

```
EmbeddedCacheManager manager = ...;
manager.executor().singleNodeSubmission().submit(...)
```

7.1.3.1. Failover

When running in single node submission it may be desirable to also allow the Cluster Executor handle cases where an exception occurred during the processing of a given command by retrying the command again. When this occurs the Cluster Executor will choose a single node again to resubmit the command to up to the desired number of failover attempts. Note the chosen node could be any node that passes the topology or predicate check. Failover is enabled by invoking the overridden singleNodeSubmission method. The given command will be resubmitted again to a single node until either the command completes without exception or the total submission amount is equal to the provided failover count.

7.1.4. Example: PI Approximation

This example shows how you can use the ClusterExecutor to estimate the value of PI.

Pi approximation can greatly benefit from parallel distributed execution via Cluster Executor. Recall that area of the square is Sa = 4r2 and area of the circle is Ca=pi*r2. Substituting r2 from the second equation into the first one it turns out that pi = 4 * Ca/Sa. Now, image that we can shoot very large number of darts into a square; if we take ratio of darts that land inside a circle over a total number of darts shot we will approximate Ca/Sa value. Since we know that pi = 4 * Ca/Sa we can easily derive

approximate value of pi. The more darts we shoot the better approximation we get. In the example below we shoot 1 billion darts but instead of "shooting" them serially we parallelize work of dart shooting across the entire Data Grid cluster. Note this will work in a cluster of 1 was well, but will be slower.

```
public class PiAppx {

  public static void main (String [] arg){
    EmbeddedCacheManager cacheManager = ..
    boolean isCluster = ..

    int numPoints = 1_000_000_000;
    int numServers = isCluster ? cacheManager.getMembers().size() : 1;
    int numberPerWorker = numPoints / numServers;

    ClusterExecutor clusterExecutor = cacheManager.executor();
    long start = System.currentTimeMillis();
    // We receive results concurrently - need to handle that
    AtomicLong countCircle = new AtomicLong();
    CompletableFuture<Void> fut = clusterExecutor.submitConsumer(m -> {
      int insideCircleCount = 0;
      for (int i = 0; i < numberPerWorker; i++) {
        double x = Math.random();
        double y = Math.random();
        if (insideCircle(x, y))
          insideCircleCount++;
      }
      return insideCircleCount;
    }, (address, count, throwable) -> {
      if (throwable != null) {
        throwable.printStackTrace();
        System.out.println("Address: " + address + " encountered an error: " + throwable);
      } else {
        countCircle.getAndAdd(count);
      }
    });
    fut.whenComplete((v, t) -> {
      // This is invoked after all nodes have responded with a value or exception
      if (t != null) {
        t.printStackTrace();
        System.out.println("Exception encountered while waiting:" + t);
      } else {
        double appxPi = 4.0 * countCircle.get() / numPoints;

        System.out.println("Distributed PI appx is " + appxPi +
            " using " + numServers + " node(s), completed in " + (System.currentTimeMillis() - start) +
" ms");
      }
    });

    // May have to sleep here to keep alive if no user threads left
  }

  private static boolean insideCircle(double x, double y) {
    return (Math.pow(x - 0.5, 2) + Math.pow(y - 0.5, 2))
```

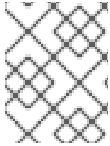
```
            <= Math.pow(0.5, 2);
   }
}
```

CHAPTER 8. USING THE STREAMS API FOR CODE EXECUTION

Efficiently process data stored in Data Grid caches using the **Streams** API.

CHAPTER 9. STREAMS

You may want to process a subset or all data in the cache to produce a result. This may bring thoughts of Map Reduce. Data Grid allows the user to do something very similar but utilizes the standard JRE APIs to do so. Java 8 introduced the concept of a Stream which allows functional-style operations on collections rather than having to procedurally iterate over the data yourself. Stream operations can be implemented in a fashion very similar to MapReduce. Streams, just like MapReduce allow you to perform processing upon the entirety of your cache, possibly a very large data set, but in an efficient way.



NOTE

Streams are the preferred method when dealing with data that exists in the cache because streams automatically adjust to cluster topology changes.

Also since we can control how the entries are iterated upon we can more efficiently perform the operations in a cache that is distributed if you want it to perform all of the operations across the cluster concurrently.

A stream is retrieved from the entrySet, keySet or values collections returned from the Cache by invoking the stream or parallelStream methods.

9.1. COMMON STREAM OPERATIONS

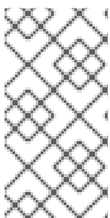
This section highlights various options that are present irrespective of what type of underlying cache you are using.

9.2. KEY FILTERING

It is possible to filter the stream so that it only operates upon a given subset of keys. This can be done by invoking the filterKeys method on the **CacheStream**. This should always be used over a Predicate filter and will be faster if the predicate was holding all keys.

If you are familiar with the **AdvancedCache** interface you may be wondering why you even use getAll over this keyFilter. There are some small benefits (mostly smaller payloads) to using getAll if you need the entries as is and need them all in memory in the local node. However if you need to do processing on these elements a stream is recommended since you will get both distributed and threaded parallelism for free.

9.3. SEGMENT BASED FILTERING



NOTE

This is an advanced feature and should only be used with deep knowledge of Data Grid segment and hashing techniques. These segments based filtering can be useful if you need to segment data into separate invocations. This can be useful when integrating with other tools such as Apache Spark.

This option is only supported for replicated and distributed caches. This allows the user to operate upon a subset of data at a time as determined by the KeyPartitioner. The segments can be filtered by invoking filterKeySegments method on the **CacheStream**. This is applied after the key filter but before any intermediate operations are performed.

9.4. LOCAL/INVALIDATION

A stream used with a local or invalidation cache can be used just the same way you would use a stream on a regular collection. Data Grid handles all of the translations if necessary behind the scenes and works with all of the more interesting options (ie. storeAsBinary and a cache loader). Only data local to the node where the stream operation is performed will be used, for example invalidation only uses local entries.

9.5. EXAMPLE

The code below takes a cache and returns a map with all the cache entries whose values contain the string "JBoss"

```
Map<Object, String> jbossValues =
cache.entrySet().stream()
    .filter(e -> e.getValue().contains("JBoss"))
    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

9.6. DISTRIBUTION/REPLICATION/SCATTERED

This is where streams come into their stride. When a stream operation is performed it will send the various intermediate and terminal operations to each node that has pertinent data. This allows processing the intermediate values on the nodes owning the data, and only sending the final results back to the originating nodes, improving performance.

9.6.1. Rehash Aware

Internally the data is segmented and each node only performs the operations upon the data it owns as a primary owner. This allows for data to be processed evenly, assuming segments are granular enough to provide for equal amounts of data on each node.

When you are utilizing a distributed cache, the data can be reshuffled between nodes when a new node joins or leaves. Distributed Streams handle this reshuffling of data automatically so you don't have to worry about monitoring when nodes leave or join the cluster. Reshuffled entries may be processed a second time, and we keep track of the processed entries at the key level or at the segment level (depending on the terminal operation) to limit the amount of duplicate processing.

It is possible but highly discouraged to disable rehash awareness on the stream. This should only be considered if your request can handle only seeing a subset of data if a rehash occurs. This can be done by invoking CacheStream.disableRehashAware() The performance gain for most operations when a rehash doesn't occur is completely negligible. The only exceptions are for iterator and forEach, which will use less memory, since they do not have to keep track of processed keys.

WARNING

Please rethink disabling rehash awareness unless you really know what you are doing.

9.6.2. Serialization

Since the operations are sent across to other nodes they must be serializable by Data Grid marshalling. This allows the operations to be sent to the other nodes.

The simplest way is to use a CacheStream instance and use a lambda just as you would normally. Data Grid overrides all of the various Stream intermediate and terminal methods to take Serializable versions of the arguments (ie. SerializableFunction, SerializablePredicate…) You can find these methods at CacheStream. This relies on the spec to pick the most specific method as defined here.

In our previous example we used a **Collector** to collect all the results into a **Map**. Unfortunately the Collectors class doesn't produce Serializable instances. Thus if you need to use these, there are two ways to do so:

One option would be to use the CacheCollectors class which allows for a **Supplier<Collector>** to be provided. This instance could then use the Collectors to supply a **Collector** which is not serialized.

```
Map<Object, String> jbossValues = cache.entrySet().stream()
          .filter(e -> e.getValue().contains("Jboss"))
          .collect(CacheCollectors.serializableCollector(() -> Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue)));
```

Alternatively, you can avoid the use of CacheCollectors and instead use the overloaded **collect** methods that take **Supplier<Collector>**. These overloaded **collect** methods are only available via **CacheStream** interface.

```
Map<Object, String> jbossValues = cache.entrySet().stream()
          .filter(e -> e.getValue().contains("Jboss"))
          .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

If however you are not able to use the **Cache** and **CacheStream** interfaces you cannot utilize **Serializable** arguments and you must instead cast the lambdas to be **Serializable** manually by casting the lambda to multiple interfaces. It is not a pretty sight but it gets the job done.

```
Map<Object, String> jbossValues = map.entrySet().stream()
          .filter((Serializable & Predicate<Map.Entry<Object, String>>) e ->
e.getValue().contains("Jboss"))
          .collect(CacheCollectors.serializableCollector(() -> Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue)));
```

The recommended and most performant way is to use an **AdvancedExternalizer** as this provides the smallest payload. Unfortunately this means you cannot use lamdbas as advanced externalizers require defining the class before hand.

You can use an advanced externalizer as shown below:

```
Map<Object, String> jbossValues = cache.entrySet().stream()
      .filter(new ContainsFilter("Jboss"))
      .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

class ContainsFilter implements Predicate<Map.Entry<Object, String>> {
  private final String target;

  ContainsFilter(String target) {
    this.target = target;
  }
```

```
    @Override
    public boolean test(Map.Entry<Object, String> e) {
      return e.getValue().contains(target);
    }
  }

  class JbossFilterExternalizer implements AdvancedExternalizer<ContainsFilter> {

    @Override
    public Set<Class<? extends ContainsFilter>> getTypeClasses() {
      return Util.asSet(ContainsFilter.class);
    }

    @Override
    public Integer getId() {
      return CUSTOM_ID;
    }

    @Override
    public void writeObject(ObjectOutput output, ContainsFilter object) throws IOException {
      output.writeUTF(object.target);
    }

    @Override
    public ContainsFilter readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      return new ContainsFilter(input.readUTF());
    }
  }
```

You could also use an advanced externalizer for the collector supplier to reduce the payload size even further.

```
Map<Object, String> map = (Map<Object, String>) cache.entrySet().stream()
          .filter(new ContainsFilter("Jboss"))
          .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

 class ToMapCollectorSupplier<K, U> implements Supplier<Collector<Map.Entry<K, U>, ?, Map<K,
U>>> {
    static final ToMapCollectorSupplier INSTANCE = new ToMapCollectorSupplier();

    private ToMapCollectorSupplier() { }

    @Override
    public Collector<Map.Entry<K, U>, ?, Map<K, U>> get() {
      return Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue);
    }
  }

  class ToMapCollectorSupplierExternalizer implements
AdvancedExternalizer<ToMapCollectorSupplier> {

    @Override
    public Set<Class<? extends ToMapCollectorSupplier>> getTypeClasses() {
      return Util.asSet(ToMapCollectorSupplier.class);
    }
```

```
    @Override
    public Integer getId() {
      return CUSTOM_ID;
    }

    @Override
    public void writeObject(ObjectOutput output, ToMapCollectorSupplier object) throws IOException
{
    }

    @Override
    public ToMapCollectorSupplier readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      return ToMapCollectorSupplier.INSTANCE;
    }
  }
```

9.7. PARALLEL COMPUTATION

Distributed streams by default try to parallelize as much as possible. It is possible for the end user to control this and actually they always have to control one of the options. There are 2 ways these streams are parallelized.

Local to each node When a stream is created from the cache collection the end user can choose between invoking stream or parallelStream method. Depending on if the parallel stream was picked will enable multiple threading for each node locally. Note that some operations like a rehash aware iterator and forEach operations will always use a sequential stream locally. This could be enhanced at some point to allow for parallel streams locally.

Users should be careful when using local parallelism as it requires having a large number of entries or operations that are computationally expensive to be faster. Also it should be noted that if a user uses a parallel stream with **forEach** that the action should not block as this would be executed on the common pool, which is normally reserved for computation operations.

Remote requests When there are multiple nodes it may be desirable to control whether the remote requests are all processed at the same time concurrently or one at a time. By default all terminal operations except the iterator perform concurrent requests. The iterator, method to reduce overall memory pressure on the local node, only performs sequential requests which actually performs slightly better.

If a user wishes to change this default however they can do so by invoking the sequentialDistribution or parallelDistribution methods on the **CacheStream**.

9.8. TASK TIMEOUT

It is possible to set a timeout value for the operation requests. This timeout is used only for remote requests timing out and it is on a per request basis. The former means the local execution will not timeout and the latter means if you have a failover scenario as described above the subsequent requests each have a new timeout. If no timeout is specified it uses the replication timeout as a default timeout. You can set the timeout in your task by doing the following:

```
CacheStream<Map.Entry<Object, String>> stream = cache.entrySet().stream();
stream.timeout(1, TimeUnit.MINUTES);
```

For more information about this, please check the java doc in timeout javadoc.

9.9. INJECTION

The Stream has a terminal operation called forEach which allows for running some sort of side effect operation on the data. In this case it may be desirable to get a reference to the **Cache** that is backing this Stream. If your **Consumer** implements the CacheAware interface the **injectCache** method be invoked before the accept method from the **Consumer** interface.

9.10. DISTRIBUTED STREAM EXECUTION

Distributed streams execution works in a fashion very similar to map reduce. Except in this case we are sending zero to many intermediate operations (map, filter etc.) and a single terminal operation to the various nodes. The operation basically comes down to the following:

1. The desired segments are grouped by which node is the primary owner of the given segment
2. A request is generated to send to each remote node that contains the intermediate and terminal operations including which segments it should process
    a. The terminal operation will be performed locally if necessary
    b. Each remote node will receive this request and run the operations and subsequently send the response back
3. The local node will then gather the local response and remote responses together performing any kind of reduction required by the operations themselves.
4. Final reduced response is then returned to the user

In most cases all operations are fully distributed, as in the operations are all fully applied on each remote node and usually only the last operation or something related may be reapplied to reduce the results from multiple nodes. One important note is that intermediate values do not actually have to be serializable, it is the last value sent back that is the part desired (exceptions for various operations will be highlighted below).

Terminal operator distributed result reductions The following paragraphs describe how the distributed reductions work for the various terminal operators. Some of these are special in that an intermediate value may be required to be serializable instead of the final result.

allMatch noneMatch anyMatch

The allMatch operation is ran on each node and then all the results are logically anded together locally to get the appropriate value. The noneMatch and anyMatch operations use a logical or instead. These methods also have early termination support, stopping remote and local operations once the final result is known.

collect

The collect method is interesting in that it can do a few extra steps. The remote node performs everything as normal except it doesn't perform the final finisher upon the result and instead sends back the fully combined results. The local thread then combines the remote and local result into a value which is then finally finished. The key here to remember is that the final value doesn't have to be serializable but rather the values produced from the supplier and combiner methods.

count

The count method just adds the numbers together from each node.

findAny findFirst

The findAny operation returns just the first value they find, whether it was from a remote node or locally. Note this supports early termination in that once a value is found it will not process others. Note the findFirst method is special since it requires a sorted intermediate operation, which is detailed in the exceptions section.

max min

The max and min methods find the respective min or max value on each node then a final reduction is performed locally to ensure only the min or max across all nodes is returned.

reduce

The various reduce methods 1 , 2 , 3 will end up serializing the result as much as the accumulator can do. Then it will accumulate the local and remote results together locally, before combining if you have provided that. Note this means a value coming from the combiner doesn't have to be Serializable.

9.11. KEY BASED REHASH AWARE OPERATORS

The iterator, spliterator and forEach are unlike the other terminal operators in that the rehash awareness has to keep track of what keys per segment have been processed instead of just segments. This is to guarantee an exactly once (iterator & spliterator) or at least once behavior (forEach) even under cluster membership changes.

The **iterator** and **spliterator** operators when invoked on a remote node will return back batches of entries, where the next batch is only sent back after the last has been fully consumed. This batching is done to limit how many entries are in memory at a given time. The user node will hold onto which keys it has processed and when a given segment is completed it will release those keys from memory. This is why sequential processing is preferred for the iterator method, so only a subset of segment keys are held in memory at once, instead of from all nodes.

The **forEach()** method also returns batches, but it returns a batch of keys after it has finished processing at least a batch worth of keys. This way the originating node can know what keys have been processed already to reduce chances of processing the same entry again. Unfortunately this means it is possible to have an at least once behavior when a node goes down unexpectedly. In this case that node could have been processing a batch and not yet completed one and those entries that were processed but not in a completed batch will be ran again when the rehash failure operation occurs. Note that adding a node will not cause this issue as the rehash failover doesn't occur until all responses are received.

These operations batch sizes are both controlled by the same value which can be configured by invoking distributedBatchSize method on the **CacheStream**. This value will default to the **chunkSize** configured in state transfer. Unfortunately this value is a tradeoff with memory usage vs performance vs at least once and your mileage may vary.

Using iterator with replicated and distributed caches

When a node is the primary or backup owner of all requested segments for a distributed stream, Data Grid performs the **iterator** or **spliterator** terminal operations locally, which optimizes performance as remote iterations are more resource intensive.

This optimization applies to both replicated and distributed caches. However, Data Grid performs iterations remotely when using cache stores that are both **shared** and have **write-behind** enabled. In this case performing the iterations remotely ensures consistency.

9.12. INTERMEDIATE OPERATION EXCEPTIONS

There are some intermediate operations that have special exceptions, these are skip, peek, sorted 12. & distinct. All of these methods have some sort of artificial iterator implanted in the stream processing to

guarantee correctness, they are documented as below. Note this means these operations may cause possibly severe performance degradation.

Skip

An artificial iterator is implanted up to the intermediate skip operation. Then results are brought locally so it can skip the appropriate amount of elements.

Sorted

WARNING: This operation requires having all entries in memory on the local node. An artificial iterator is implanted up to the intermediate sorted operation. All results are sorted locally. There are possible plans to have a distributed sort which returns batches of elements, but this is not yet implemented.

Distinct

WARNING: This operation requires having all or nearly all entries in memory on the local node. Distinct is performed on each remote node and then an artificial iterator returns those distinct values. Then finally all of those results have a distinct operation performed upon them.

The rest of the intermediate operations are fully distributed as one would expect.

9.13. EXAMPLES

Word Count

Word count is a classic, if overused, example of map/reduce paradigm. Assume we have a mapping of key → sentence stored on Data Grid nodes. Key is a String, each sentence is also a String, and we have to count occurrence of all words in all sentences available. The implementation of such a distributed task could be defined as follows:

```
public class WordCountExample {

  /**
   * In this example replace c1 and c2 with
   * real Cache references
   *
   * @param args
   */
  public static void main(String[] args) {
    Cache<String, String> c1 = ...;
    Cache<String, String> c2 = ...;

    c1.put("1", "Hello world here I am");
    c2.put("2", "Infinispan rules the world");
    c1.put("3", "JUDCon is in Boston");
    c2.put("4", "JBoss World is in Boston as well");
    c1.put("12","JBoss Application Server");
    c2.put("15", "Hello world");
    c1.put("14", "Infinispan community");
    c2.put("15", "Hello world");

    c1.put("111", "Infinispan open source");
    c2.put("112", "Boston is close to Toronto");
    c1.put("113", "Toronto is a capital of Ontario");
    c2.put("114", "JUDCon is cool");
    c1.put("211", "JBoss World is awesome");
    c2.put("212", "JBoss rules");
    c1.put("213", "JBoss division of RedHat ");
```

```
        c2.put("214", "RedHat community");

        Map<String, Long> wordCountMap = c1.entrySet().parallelStream()
            .map(e -> e.getValue().split("\\s"))
            .flatMap(Arrays::stream)
            .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.counting()));
    }
}
```

In this case it is pretty simple to do the word count from the previous example.

However what if we want to find the most frequent word in the example? If you take a second to think about this case you will realize you need to have all words counted and available locally first. Thus we actually have a few options.

We could use a finisher on the collector, which is invoked on the user thread after all the results have been collected. Some redundant lines have been removed from the previous example.

```
public class WordCountExample {
    public static void main(String[] args) {
        // Lines removed

        String mostFrequentWord = c1.entrySet().parallelStream()
            .map(e -> e.getValue().split("\\s"))
            .flatMap(Arrays::stream)
            .collect(() -> Collectors.collectingAndThen(
                Collectors.groupingBy(Function.identity(), Collectors.counting()),
                    wordCountMap -> {
                        String mostFrequent = null;
                        long maxCount = 0;
                            for (Map.Entry<String, Long> e : wordCountMap.entrySet()) {
                                int count = e.getValue().intValue();
                                if (count > maxCount) {
                                    maxCount = count;
                                    mostFrequent = e.getKey();
                                }
                            }
                            return mostFrequent;
                    }));

}
```

Unfortunately the last step is only going to be ran in a single thread, which if we have a lot of words could be quite slow. Maybe there is another way to parallelize this with Streams.

We mentioned before we are in the local node after processing, so we could actually use a stream on the map results. We can therefore use a parallel stream on the results.

```
public class WordFrequencyExample {
    public static void main(String[] args) {
        // Lines removed

        Map<String, Long> wordCount = c1.entrySet().parallelStream()
                .map(e -> e.getValue().split("\\s"))
                .flatMap(Arrays::stream)
                .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.counting()));
```

```
    Optional<Map.Entry<String, Long>> mostFrequent =
wordCount.entrySet().parallelStream().reduce(
        (e1, e2) -> e1.getValue() > e2.getValue() ? e1 : e2);
```

This way you can still utilize all of the cores locally when calculating the most frequent element.

Remove specific entries

Distributed streams can also be used as a way to modify data where it lives. For example you may want to remove all entries in your cache that contain a specific word.

```
public class RemoveBadWords {
  public static void main(String[] args) {
    // Lines removed
    String word = ..

    c1.entrySet().parallelStream()
      .filter(e -> e.getValue().contains(word))
      .forEach((c, e) -> c.remove(e.getKey()));
```

If we carefully note what is serialized and what is not, we notice that only the word along with the operations are serialized across to other nods as it is captured by the lambda. However the real saving piece is that the cache operation is performed on the primary owner thus reducing the amount of network traffic required to remove these values from the cache. The cache is not captured by the lambda as we provide a special BiConsumer method override that when invoked on each node passes the cache to the BiConsumer

One thing to keep in mind using the **forEach** command in this manner is that the underlying stream obtains no locks. The cache remove operation will still obtain locks naturally, but the value could have changed from what the stream saw. That means that the entry could have been changed after the stream read it but the remove actually removed it.

We have specifically added a new variant which is called **LockedStream**.

Plenty of other examples

The **Streams** API is a JRE tool and there are lots of examples for using it. Just remember that your operations need to be Serializable in some way.

CHAPTER 10. USING THE CDI EXTENSION

Data Grid provides an extension that integrates with the CDI (Contexts and Dependency Injection) programming model and allows you to:

- Configure and inject caches into CDI Beans and Java EE components.
- Configure cache managers.
- Receive cache and cache manager level events.
- Control data storage and retrieval using JCache annotations.

10.1. CDI DEPENDENCIES

Update your **pom.xml** with one of the following dependencies to include the Data Grid CDI extension in your project:

Embedded (Library) Mode

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-embedded</artifactId>
</dependency>
```

Server Mode

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-remote</artifactId>
</dependency>
```

10.2. INJECTING EMBEDDED CACHES

Set up CDI beans to inject embedded caches.

Procedure

1. Create a cache qualifier annotation.

```
...
import javax.inject.Qualifier;

@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GreetingCache { (1)
}
```

(1) Creates a **@GreetingCache** qualifier.

2. Add a producer method that defines the cache configuration.

```
...
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.cdi.ConfigureCache;
import javax.enterprise.inject.Produces;

public class Config {

   @ConfigureCache("mygreetingcache") 1
   @GreetingCache 2
   @Produces
   public Configuration greetingCacheConfiguration() {
      return new ConfigurationBuilder()
               .memory()
                  .size(1000)
               .build();
   }
}
```

1 Names the cache to inject.

2 Adds the cache qualifier.

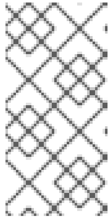
3. Add a producer method that creates a clustered Cache Manager, if required

```
...
package org.infinispan.configuration.global.GlobalConfigurationBuilder;

public class Config {

   @GreetingCache 1
   @Produces
   @ApplicationScoped 2
   public EmbeddedCacheManager defaultClusteredCacheManager() { 3
     return new DefaultCacheManager(
       new GlobalConfigurationBuilder().transport().defaultTransport().build();
   }
}
```

1 Adds the cache qualifier.

2 Creates the bean once for the application. Producers that create Cache Managers should always include the **@ApplicationScoped** annotation to avoid creating multiple Cache Managers.

3 Creates a new **DefaultCacheManager** instance that is bound to the **@GreetingCache** qualifier.



NOTE

Cache managers are heavy weight objects. Having more than one Cache Manager running in your application can degrade performance. When injecting multiple caches, either add the qualifier of each cache to the Cache Manager producer method or do not add any qualifier.

4. Add the **@GreetingCache** qualifier to your cache injection point.

```
...
import javax.inject.Inject;

public class GreetingService {

    @Inject @GreetingCache
    private Cache<String, String> cache;

    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

10.3. INJECTING REMOTE CACHES

Set up CDI beans to inject remote caches.

Procedure

1. Create a cache qualifier annotation.

```
@Remote("mygreetingcache") (1)
@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RemoteGreetingCache { (2)
}
```

(1) names the cache to inject.

(2) creates a **@RemoteGreetingCache** qualifier.

2. Add the **@RemoteGreetingCache** qualifier to your cache injection point.

```
public class GreetingService {

    @Inject @RemoteGreetingCache
    private RemoteCache<String, String> cache;
```

```
    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

Tips for injecting remote caches

- You can inject remote caches without using qualifiers.

```
...
@Inject
@Remote("greetingCache")
private RemoteCache<String, String> cache;
```

- If you have more than one Data Grid cluster, you can create separate remote Cache Manager producers for each cluster.

```
...
import javax.enterprise.context.ApplicationScoped;

public class Config {

    @RemoteGreetingCache
    @Produces
    @ApplicationScoped (1)
    public ConfigurationBuilder builder = new ConfigurationBuilder(); (2)
        builder.addServer().host("localhost").port(11222);
        return new RemoteCacheManager(builder.build());
    }
}
```

1. creates the bean once for the application. Producers that create Cache Managers should always include the **@ApplicationScoped** annotation to avoid creating multiple Cache Managers, which are heavy weight objects.
2. creates a new **RemoteCacheManager** instance that is bound to the **@RemoteGreetingCache** qualifier.

10.4. JCACHE CACHING ANNOTATIONS

You can use the following JCache caching annotations with CDI managed beans when JCache artifacts are on the classpath:

@CacheResult

caches the results of method calls.

@CachePut

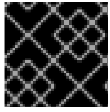
caches method parameters.

@CacheRemoveEntry

removes entries from a cache.

@CacheRemoveAll

removes all entries from a cache.



IMPORTANT

Target type: You can use these JCache caching annotations on methods only.

To use JCache caching annotations, declare interceptors in the **beans.xml** file for your application.

Managed Environments (Application Server)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  version="1.2" bean-discovery-mode="annotated">

 <interceptors>
   <class>org.infinispan.jcache.annotation.InjectedCacheResultInterceptor</class>
   <class>org.infinispan.jcache.annotation.InjectedCachePutInterceptor</class>
   <class>org.infinispan.jcache.annotation.InjectedCacheRemoveEntryInterceptor</class>
   <class>org.infinispan.jcache.annotation.InjectedCacheRemoveAllInterceptor</class>
 </interceptors>
</beans>
```

Non-managed Environments (Standalone)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  version="1.2" bean-discovery-mode="annotated">

 <interceptors>
   <class>org.infinispan.jcache.annotation.CacheResultInterceptor</class>
   <class>org.infinispan.jcache.annotation.CachePutInterceptor</class>
   <class>org.infinispan.jcache.annotation.CacheRemoveEntryInterceptor</class>
   <class>org.infinispan.jcache.annotation.CacheRemoveAllInterceptor</class>
 </interceptors>
</beans>
```

JCache Caching Annotation Examples

The following example shows how the **@CacheResult** annotation caches the results of the **GreetingService.greet()** method:

```
import javax.cache.interceptor.CacheResult;
```

```
public class GreetingService {

    @CacheResult
    public String greet(String user) {
        return "Hello" + user;
    }
}
```

With JCache annotations, the default cache uses the fully qualified name of the annotated method with its parameter types, for example:
org.infinispan.example.GreetingService.greet(java.lang.String)

To use caches other than the default, use the **cacheName** attribute to specify the cache name as in the following example:

```
@CacheResult(cacheName = "greeting-cache")
```

10.5. RECEIVING CACHE AND CACHE MANAGER EVENTS

You can use CDI Events to receive Cache and Cache Manager level events.

- Use the **@Observes** annotation as in the following example:

```
import javax.enterprise.event.Observes;
import org.infinispan.notifications.cachemanagerlistener.event.CacheStartedEvent;
import org.infinispan.notifications.cachelistener.event.*;

public class GreetingService {

    // Cache level events
    private void entryRemovedFromCache(@Observes CacheEntryCreatedEvent event) {
        ...
    }

    // Cache manager level events
    private void cacheStarted(@Observes CacheStartedEvent event) {
        ...
    }
}
```

CHAPTER 11. USING THE JCACHE API

Data Grid provides an implementation of the JCache (JSR-107) API that specifies a standard Java API for caching temporary Java objects in memory. Caching Java objects can help get around bottlenecks arising from using data that is expensive to retrieve or data that is hard to calculate. Caching these type of objects in memory can help speed up application performance by retrieving the data directly from memory instead of doing an expensive roundtrip or recalculation.

11.1. CREATING EMBEDDED CACHES

Prerequisites

1. Ensure that **cache-api** is on your classpath.
2. Add the following dependency to your **pom.xml**:

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-jcache</artifactId>
</dependency>
```

Procedure

- Create embedded caches that use the default JCache API configuration as follows:

```
import javax.cache.*;
import javax.cache.configuration.*;

// Retrieve the system wide Cache Manager
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
```

11.1.1. Configuring embedded caches

- Pass the URI for custom Data Grid configuration to the **CachingProvider.getCacheManager(URI)** call as follows:

```
import java.net.URI;
import javax.cache.*;
import javax.cache.configuration.*;

// Load configuration from an absolute filesystem path
URI uri = URI.create("file:///path/to/infinispan.xml");
// Load configuration from a classpath resource
// URI uri = this.getClass().getClassLoader().getResource("infinispan.xml").toURI();

// Create a Cache Manager using the above configuration
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager(uri,
this.getClass().getClassLoader(), null);
```

WARNING

By default, the JCache API specifies that data should be stored as **storeByValue**, so that object state mutations outside of operations to the cache, won't have an impact in the objects stored in the cache. Data Grid has so far implemented this using serialization/marshalling to make copies to store in the cache, and that way adhere to the spec. Hence, if using default JCache configuration with Data Grid, data stored must be marshallable.

Alternatively, JCache can be configured to store data by reference (just like Data Grid or JDK Collections work). To do that, simply call:

```
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>().setStoreByValue(false));
```

11.2. STORE AND RETRIEVE DATA

Even though JCache API does not extend neither java.util.Map not java.util.concurrent.ConcurrentMap, it providers a key/value API to store and retrieve data:

```
import javax.cache.*;
import javax.cache.configuration.*;

CacheManager cacheManager = Caching.getCachingProvider().getCacheManager();
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
cache.put("hello", "world"); // Notice that javax.cache.Cache.put(K) returns void!
String value = cache.get("hello"); // Returns "world"
```

Contrary to standard java.util.Map, javax.cache.Cache comes with two basic put methods called put and getAndPut. The former returns **void** whereas the latter returns the previous value associated with the key. So, the equivalent of java.util.Map.put(K) in JCache is javax.cache.Cache.getAndPut(K).

TIP

Even though JCache API only covers standalone caching, it can be plugged with a persistence store, and has been designed with clustering or distribution in mind. The reason why javax.cache.Cache offers two put methods is because standard java.util.Map put call forces implementors to calculate the previous value. When a persistent store is in use, or the cache is distributed, returning the previous value could be an expensive operation, and often users call standard java.util.Map.put(K) without using the return value. Hence, JCache users need to think about whether the return value is relevant to them, in which case they need to call javax.cache.Cache.getAndPut(K) , otherwise they can call java.util.Map.put(K, V) which avoids returning the potentially expensive operation of returning the previous value.

11.3. COMPARING JAVA.UTIL.CONCURRENT.CONCURRENTMAP AND JAVAX.CACHE.CACHE APIS

Here's a brief comparison of the data manipulation APIs provided by java.util.concurrent.ConcurrentMap and javax.cache.Cache APIs.

Operation	java.util.concurrent.ConcurrentMap<K, V>	javax.cache.Cache<K, V>
store and no return	N/A	**void put(K key)**
store and return previous value	**V put(K key)**	**V getAndPut(K key)**
store if not present	**V putIfAbsent(K key, V value)**	**boolean putIfAbsent(K key, V value)**
retrieve	**V get(Object key)**	**V get(K key)**
delete if present	**V remove(Object key)**	**boolean remove(K key)**
delete and return previous value	**V remove(Object key)**	**V getAndRemove(K key)**
delete conditional	**boolean remove(Object key, Object value)**	**boolean remove(K key, V oldValue)**
replace if present	**V replace(K key, V value)**	**boolean replace(K key, V value)**
replace and return previous value	**V replace(K key, V value)**	**V getAndReplace(K key, V value)**
replace conditional	**boolean replace(K key, V oldValue, V newValue)**	**boolean replace(K key, V oldValue, V newValue)**

Comparing the two APIs, it's obvious to see that, where possible, JCache avoids returning the previous value to avoid operations doing expensive network or IO operations. This is an overriding principle in the design of JCache API. In fact, there's a set of operations that are present in java.util.concurrent.ConcurrentMap , but are not present in the javax.cache.Cache because they could be expensive to compute in a distributed cache. The only exception is iterating over the contents of the cache:

Operation	java.util.concurrent.ConcurrentMap<K, V>	javax.cache.Cache<K, V>
calculate size of cache	**int size()**	N/A
return all keys in the cache	**Set<K> keySet()**	N/A
return all values in the cache	**Collection<V> values()**	N/A

Operation	java.util.concurrent.ConcurrentMap<K, V>	javax.cache.Cache<K, V>
return all entries in the cache	**Set<Map.Entry<K, V>> entrySet()**	N/A
iterate over the cache	use **iterator()** method on keySet, values or entrySet	**Iterator<Cache.Entry<K, V>> iterator()**

11.4. CLUSTERING JCACHE INSTANCES

Data Grid JCache implementation goes beyond the specification in order to provide the possibility to cluster caches using the standard API. Given a Data Grid configuration file configured to replicate caches like this:

infinispan.xml

```
<infinispan>
  <cache-container default-cache="namedCache">
    <transport cluster="jcache-cluster" />
    <replicated-cache name="namedCache" />
  </cache-container>
</infinispan>
```

You can create a cluster of caches using this code:

```
import javax.cache.*;
import java.net.URI;

// For multiple Cache Managers to be constructed with the standard JCache API
// and live in the same JVM, either their names, or their classloaders, must
// be different.
// This example shows how to force their classloaders to be different.
// An alternative method would have been to duplicate the XML file and give
// it a different name, but this results in unnecessary file duplication.
ClassLoader tccl = Thread.currentThread().getContextClassLoader();
CacheManager cacheManager1 = Caching.getCachingProvider().getCacheManager(
     URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));
CacheManager cacheManager2 = Caching.getCachingProvider().getCacheManager(
     URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));

Cache<String, String> cache1 = cacheManager1.getCache("namedCache");
Cache<String, String> cache2 = cacheManager2.getCache("namedCache");

cache1.put("hello", "world");
String value = cache2.get("hello"); // Returns "world" if clustering is working

// --

public static class TestClassLoader extends ClassLoader {
  public TestClassLoader(ClassLoader parent) {
```

```
        super(parent);
    }
}
```

CHAPTER 12. MULTIMAP CACHE

MutimapCache is a type of Data Grid Cache that maps keys to values in which each key can contain multiple values.

12.1. MULTIMAP CACHE

MutimapCache is a type of Data Grid Cache that maps keys to values in which each key can contain multiple values.

12.1.1. Installation and configuration

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-multimap</artifactId>
</dependency>
```

12.1.2. MultimapCache API

MultimapCache API exposes several methods to interact with the Multimap Cache. These methods are non-blocking in most cases; see limitations for more information.

```
public interface MultimapCache<K, V> {

  CompletableFuture<Optional<CacheEntry<K, Collection<V>>>> getEntry(K key);

  CompletableFuture<Void> remove(SerializablePredicate<? super V> p);

  CompletableFuture<Void> put(K key, V value);

  CompletableFuture<Collection<V>> get(K key);

  CompletableFuture<Boolean> remove(K key);

  CompletableFuture<Boolean> remove(K key, V value);

  CompletableFuture<Void> remove(Predicate<? super V> p);

  CompletableFuture<Boolean> containsKey(K key);

  CompletableFuture<Boolean> containsValue(V value);

  CompletableFuture<Boolean> containsEntry(K key, V value);

  CompletableFuture<Long> size();

  boolean supportsDuplicates();

}
```

CompletableFuture<Void> put(K key, V value)

Puts a key-value pair in the multimap cache.

```
MultimapCache<String, String> multimapCache = ...;

multimapCache.put("girlNames", "marie")
             .thenCompose(r1 -> multimapCache.put("girlNames", "oihana"))
             .thenCompose(r3 -> multimapCache.get("girlNames"))
             .thenAccept(names -> {
                          if(names.contains("marie"))
                              System.out.println("Marie is a girl name");

                           if(names.contains("oihana"))
                              System.out.println("Oihana is a girl name");
                        });
```

The output of this code is as follows:

```
Marie is a girl name
Oihana is a girl name
```

CompletableFuture<Collection<V>> get(K key)

Asynchronous that returns a view collection of the values associated with key in this multimap cache, if any. Any changes to the retrieved collection won't change the values in this multimap cache. When this method returns an empty collection, it means the key was not found.

CompletableFuture<Boolean> remove(K key)

Asynchronous that removes the entry associated with the key from the multimap cache, if such exists.

CompletableFuture<Boolean> remove(K key, V value)

Asynchronous that removes a key-value pair from the multimap cache, if such exists.

CompletableFuture<Void> remove(Predicate<? super V> p)

Asynchronous method. Removes every value that match the given predicate.

CompletableFuture<Boolean> containsKey(K key)

Asynchronous that returns true if this multimap contains the key.

CompletableFuture<Boolean> containsValue(V value)

Asynchronous that returns true if this multimap contains the value in at least one key.

CompletableFuture<Boolean> containsEntry(K key, V value)

Asynchronous that returns true if this multimap contains at least one key-value pair with the value.

CompletableFuture<Long> size()

Asynchronous that returns the number of key-value pairs in the multimap cache. It doesn't return the distinct number of keys.

boolean supportsDuplicates()

Asynchronous that returns true if the multimap cache supports duplicates. This means that the content of the multimap can be 'a' → ['1', '1', '2']. For now this method will always return false, as duplicates are not

yet supported. The existence of a given value is determined by 'equals' and `hashcode' method's contract.

12.1.3. Creating a Multimap Cache

Currently the MultimapCache is configured as a regular cache. This can be done either by code or XML configuration. See how to configure a regular cache in Configuring Data Grid caches.

12.1.3.1. Embedded mode

```
// create or obtain your EmbeddedCacheManager
EmbeddedCacheManager cm = ... ;

// create or obtain a MultimapCacheManager passing the EmbeddedCacheManager
MultimapCacheManager multimapCacheManager =
EmbeddedMultimapCacheManagerFactory.from(cm);

// define the configuration for the multimap cache
multimapCacheManager.defineConfiguration(multimapCacheName, c.build());

// get the multimap cache
multimapCache = multimapCacheManager.get(multimapCacheName);
```

12.1.4. Limitations

In almost every case the Multimap Cache will behave as a regular Cache, but some limitations exist in the current version, as follows:

12.1.4.1. Support for duplicates

Duplicates are not supported yet. This means that the multimap won't contain any duplicate key-value pair. Whenever put method is called, if the key-value pair already exist, this key-value par won't be added. Methods used to check if a key-value pair is already present in the Multimap are the **equals** and **hashcode**.

12.1.4.2. Eviction

For now, the eviction works per key, and not per key-value pair. This means that whenever a key is evicted, all the values associated with the key will be evicted too.

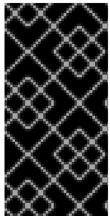
12.1.4.3. Transactions

Implicit transactions are supported through the auto-commit and all the methods are non blocking. Explicit transactions work without blocking in most of the cases. Methods that will block are **size**, **containsEntry** and **remove(Predicate<? super V> p)**

CHAPTER 13. DATA GRID MODULES FOR RED HAT JBOSS EAP

To use Data Grid inside applications deployed to Red Hat JBoss EAP, you should install Data Grid modules that:

- Let you deploy applications without packaging Data Grid JAR files in your WAR or EAR file.
- Allow you to use a Data Grid version that is independent to the one bundled with Red Hat JBoss EAP.



IMPORTANT

Red Hat JBoss EAP (EAP) applications can directly handle the **infinispan** subsystem without the need to separately install Data Grid modules. Red Hat provides support for this functionality since EAP version 7.4. However, your deployment requires the EAP modules to use advanced capabilities such as indexing and querying.

13.1. INSTALLING DATA GRID MODULES

Download and install Data Grid modules for Red Hat JBoss EAP.

Prerequisites

1. JDK 8 or later.
2. An existing Red Hat JBoss EAP installation.

Procedure

1. Log in to the Red Hat customer portal.
2. Download the ZIP archive for the modules from the Data Grid software downloads.
3. Extract the ZIP archive and copy the contents of **modules** to the **modules** directory of your Red Hat JBoss EAP installation so that you get the resulting structure: **$EAP_HOME/modules/system/add-ons/rhdg/org/infinispan/rhdg-8.4**

13.2. CONFIGURING APPLICATIONS TO USE DATA GRID MODULES

After you install Data Grid modules for Red Hat JBoss EAP, configure your application to use Data Grid functionality.

Procedure

1. In your project **pom.xml** file, mark the required Data Grid dependencies as *provided*.
2. Configure your artifact archiver to generate the appropriate **MANIFEST.MF** file.

pom.xml

```
<dependencies>
  <dependency>
```

```
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-core</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-cachestore-jdbc</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <configuration>
          <archive>
            <manifestEntries>
              <Dependencies>org.infinispan:rhdg-8.4 services</Dependencies>
            </manifestEntries>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

Data Grid functionality is packaged as a single module, **org.infinispan**, that you can add as an entry to your application’s manifest as follows:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:rhdg-8.4 services
```

AWS dependencies

If you require AWS dependencies, such as S3_PING, add the following module to your application’s manifest:

```
Manifest-Version: 1.0
Dependencies: com.amazonaws.aws-java-sdk:rhdg-8.4 services
```