



Red Hat Data Grid 8.3

Hot Rod Java Client Guide

Configure and use Hot Rod Java clients

Red Hat Data Grid 8.3 Hot Rod Java Client Guide

Configure and use Hot Rod Java clients

Legal Notice

Copyright © 2022 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

Hot Rod Java clients give you high-performance remote access to Data Grid clusters.

Table of Contents

RED HAT DATA GRID	4
DATA GRID DOCUMENTATION	5
DATA GRID DOWNLOADS	6
MAKING OPEN SOURCE MORE INCLUSIVE	7
PROVIDING FEEDBACK ON RED HAT DOCUMENTATION	8
CHAPTER 1. HOT ROD JAVA CLIENTS	9
1.1. HOT ROD PROTOCOL	9
1.2. CLIENT INTELLIGENCE	9
Configuration	10
1.3. REQUEST BALANCING	10
1.4. CLIENT FAILOVER	11
1.5. HOT ROD CLIENT COMPATIBILITY WITH DATA GRID SERVER	11
CHAPTER 2. CONFIGURING THE DATA GRID MAVEN REPOSITORY	13
2.1. DOWNLOADING THE DATA GRID MAVEN REPOSITORY	13
2.2. ADDING RED HAT MAVEN REPOSITORIES	13
2.3. CONFIGURING YOUR DATA GRID POM	14
CHAPTER 3. HOT ROD JAVA CLIENT CONFIGURATION	15
3.1. ADDING HOT ROD JAVA CLIENT DEPENDENCIES	15
3.2. CONFIGURING HOT ROD CLIENT CONNECTIONS	15
Configuring Hot Rod URIs	16
Adding properties outside the classpath	16
3.2.1. Defining Data Grid Clusters in Client Configuration	16
3.2.2. Manually Switching Data Grid Clusters	17
3.2.3. Configuring Connection Pools	17
3.3. CONFIGURING AUTHENTICATION MECHANISMS FOR HOT ROD CLIENTS	18
Basic Callback Handler	20
OAUTHBEARER with Token Callback Handler	20
Custom CallbackHandler	20
3.3.1. Creating GSSAPI Login Contexts	21
3.3.2. SASL authentication mechanisms	22
3.4. CONFIGURING HOT ROD CLIENT ENCRYPTION	23
3.5. ENABLING HOT ROD CLIENT STATISTICS	25
Hot Rod Java client statistics	25
3.6. NEAR CACHES	26
3.6.1. Configuring Near Caches	26
3.7. FORCING RETURN VALUES	27
3.8. CREATING REMOTE CACHES FROM HOT ROD CLIENTS	28
CHAPTER 4. HOT ROD CLIENT API	30
4.1. REMOTECACHE API	30
4.1.1. Unsupported Methods	30
4.2. REMOTE ITERATOR API	31
4.2.1. Deploying Custom Filters to Data Grid Server	31
4.3. METADATAVALUE API	32
4.4. STREAMING API	33
4.5. COUNTER API	34

4.6. CREATING EVENT LISTENERS	34
4.6.1. Removing Event Listeners	35
4.6.2. Filtering Events	35
4.6.3. Skipping Notifications	37
4.6.4. Customizing Events	37
4.6.5. Filter and Custom Events	40
4.6.6. Event Marshalling	42
4.6.6.1. Deploying Protostream Marshallers	42
4.6.7. Listener State Handling	42
4.6.8. Listener Failure Handling	42
4.7. HOT ROD JAVA CLIENT TRANSACTIONS	43
4.7.1. Configuring the Server	43
4.7.2. Configuring Hot Rod Clients	44
4.7.2.1. TransactionManagerLookup Interface	44
4.7.3. Transaction Modes	44
4.7.4. Detecting Conflicts with Transactions	45
4.7.5. Using the Configured Transaction Manager and Transaction Mode	47

RED HAT DATA GRID

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

Schemaless data structure

Flexibility to store different objects as key-value pairs.

Grid-based data storage

Designed to distribute and replicate data across clusters.

Elastic scaling

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

Data interoperability

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

DATA GRID DOCUMENTATION

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

- [Data Grid 8.3 Documentation](#)
- [Data Grid 8.3 Component Details](#)
- [Supported Configurations for Data Grid 8.3](#)
- [Data Grid 8 Feature Support](#)
- [Data Grid Deprecated Features and Functionality](#)

DATA GRID DOWNLOADS

Access the [Data Grid Software Downloads](#) on the Red Hat customer portal.



NOTE

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

MAKING OPEN SOURCE MORE INCLUSIVE

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

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

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



NOTE

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

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

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

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

Thank you for the valuable feedback.

CHAPTER 1. HOT ROD JAVA CLIENTS

Access Data Grid remotely through the Hot Rod Java client API.

1.1. HOT ROD PROTOCOL

Hot Rod is a binary TCP protocol that Data Grid offers high-performance client-server interactions with the following capabilities:

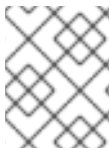
- Load balancing. Hot Rod clients can send requests across Data Grid clusters using different strategies.
- Failover. Hot Rod clients can monitor Data Grid cluster topology changes and automatically switch to available nodes.
- Efficient data location. Hot Rod clients can find key owners and make requests directly to those nodes, which reduces latency.

1.2. CLIENT INTELLIGENCE

Hot Rod clients use intelligence mechanisms to efficiently send requests to Data Grid Server clusters. By default, the Hot Rod protocol has the **HASH_DISTRIBUTION_AWARE** intelligence mechanism enabled.

BASIC intelligence

Clients do not receive topology change events for Data Grid clusters, such as nodes joining or leaving, and use only the list of Data Grid Server network locations that you add to the client configuration.



NOTE

Enable **BASIC** intelligence to use the Hot Rod client configuration when a Data Grid Server does not send internal and hidden cluster topology to the Hot Rod client.

TOPOLOGY_AWARE intelligence

Clients receive and store topology change events for Data Grid clusters to dynamically keep track of Data Grid Servers on the network.

To receive cluster topology, clients need the network location, either IP address or host name, of at least one Hot Rod server at startup. After the client connects, Data Grid Server transmits the topology to the client. When Data Grid Server nodes join or leave the cluster, Data Grid transmits an updated topology to the client.

HASH_DISTRIBUTION_AWARE intelligence

Clients receive and store topology change events for Data Grid clusters in addition to hashing information that enables clients to identify which nodes store specific keys.

For example, consider a **put(k,v)** operation. The client calculates the hash value for the key so it can locate the exact Data Grid Server node on which the data resides. Clients can then connect directly to that node to perform read and write operations.

The benefit of **HASH_DISTRIBUTION_AWARE** intelligence is that Data Grid Server does not need to look up values based on key hashes, which uses less server-side resources. Another benefit is that Data Grid Server responds to client requests more quickly because they do not need to make additional

network roundtrips.

Configuration

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clientIntelligence(ClientIntelligence.BASIC);
```

hotrod-client.properties

```
infinispan.client.hotrod.client_intelligence=BASIC
```

Additional resources

- org.infinispan.client.hotrod.configuration.ClientIntelligence

1.3. REQUEST BALANCING

Hot Rod Java clients balance requests to Data Grid Server clusters so that read and write operations are spread across nodes.

Clients that use **BASIC** or **TOPOLOGY_AWARE** intelligence use request balancing for all requests. Clients that use **HASH_DISTRIBUTION_AWARE** intelligence send requests directly to the node that stores the desired key. If the node does not respond, the clients then fall back to request balancing.

The default balancing strategy is round-robin, so Hot Rod clients perform request balancing as in the following example where **s1**, **s2**, **s3** are nodes in a Data Grid cluster:

```
// Connect to the Data Grid cluster
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());
// Obtain the remote cache
RemoteCache<String, String> cache = cacheManager.getCache("test");

//Hot Rod client sends a request to the "s1" node
cache.put("key1", "aValue");
//Hot Rod client sends a request to the "s2" node
cache.put("key2", "aValue");
//Hot Rod client sends a request to the "s3" node
String value = cache.get("key1");
//Hot Rod client sends the next request to the "s1" node again
cache.remove("key2");
```

Custom balancing policies

You can use custom **FailoverRequestBalancingStrategy** implementations if you add your class in the Hot Rod client configuration.

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer()
```

```
.host("127.0.0.1")
.port(11222)
.balancingStrategy(new MyCustomBalancingStrategy());
```

hotrod-client.properties

```
infinispan.client.hotrod.request_balancing_strategy=my.package.MyCustomBalancingStrategy
```

Additional resources

- [org.infinispan.client.hotrod.FailoverRequestBalancingStrategy](#)

1.4. CLIENT FAILOVER

Hot Rod clients can automatically failover when Data Grid cluster topologies change. For instance, Hot Rod clients that are topology-aware can detect when one or more Data Grid servers fail.

In addition to failover between clustered Data Grid servers, Hot Rod clients can failover between Data Grid clusters.

For example, you have a Data Grid cluster running in New York (**NYC**) and another cluster running in London (**LON**). Clients sending requests to **NYC** detect that no nodes are available so they switch to the cluster in **LON**. Clients then maintain connections to **LON** until you manually switch clusters or failover happens again.

Transactional Caches with Failover

Conditional operations, such as **putIfAbsent()**, **replace()**, **remove()**, have strict method return guarantees. Likewise, some operations can require previous values to be returned.

Even though Hot Rod clients can failover, you should use transactional caches to ensure that operations do not partially complete and leave conflicting entries on different nodes.

1.5. HOT ROD CLIENT COMPATIBILITY WITH DATA GRID SERVER

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

TIP

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

Data Grid 8 and later

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

Data Grid 7.3 and earlier

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

Additional resources

- [Hot Rod protocol reference](#)
- [Connecting Hot Rod clients to servers with different versions](#) (Red Hat Knowledgebase)

CHAPTER 2. CONFIGURING THE DATA GRID MAVEN REPOSITORY

Data Grid Java distributions are available from Maven.

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

2.1. DOWNLOADING THE DATA GRID MAVEN REPOSITORY

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

Procedure

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

2.2. 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](#)

2.3. CONFIGURING YOUR DATA GRID POM

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

Procedure

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

The following example shows the Data Grid version and BOM:

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

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

Next Steps

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

CHAPTER 3. HOT ROD JAVA CLIENT CONFIGURATION

Data Grid provides a Hot Rod Java client configuration API that exposes configuration properties.

3.1. ADDING HOT ROD JAVA CLIENT DEPENDENCIES

Add Hot Rod Java client dependencies to include it in your project.

Prerequisites

- Java 8 or Java 11

Procedure

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

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-client-hotrod</artifactId>
</dependency>
```

Reference

[Data Grid Server Requirements](#)

3.2. CONFIGURING HOT ROD CLIENT CONNECTIONS

Configure Hot Rod Java client connections to Data Grid Server.

Procedure

- Use the **ConfigurationBuilder** class to generate immutable configuration objects that you can pass to **RemoteCacheManager** or use a **hotrod-client.properties** file on the application classpath.

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer()
    .host("127.0.0.1")
    .port(ConfigurationProperties.DEFAULT_HOTROD_PORT)
    .addServer()
    .host("192.0.2.0")
    .port(ConfigurationProperties.DEFAULT_HOTROD_PORT)
    .security().authentication()
    .username("username")
    .password("changeme")
    .realm("default")
    .saslMechanism("SCRAM-SHA-512");
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());
```

hotrod-client.properties

```

infinispan.client.hotrod.server_list = 127.0.0.1:11222,192.0.2.0:11222
infinispan.client.hotrod.auth_username = username
infinispan.client.hotrod.auth_password = changeme
infinispan.client.hotrod.auth_realm = default
infinispan.client.hotrod.sasl_mechanism = SCRAM-SHA-512

```

Configuring Hot Rod URIs

You can also configure Hot Rod client connections with URIs as follows:

ConfigurationBuilder

```

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.uri("hotrod://username:changeme@127.0.0.1:11222,192.0.2.0:11222?
auth_realm=default&sasl_mechanism=SCRAM-SHA-512");
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());

```

hotrod-client.properties

```

infinispan.client.hotrod.uri = hotrod://username:changeme@127.0.0.1:11222,192.0.2.0:11222?
auth_realm=default&sasl_mechanism=SCRAM-SHA-512

```

Adding properties outside the classpath

If the **hotrod-client.properties** file is not on the application classpath then you need to specify the location, as in the following example:

```

ConfigurationBuilder builder = new ConfigurationBuilder();
Properties p = new Properties();
try(Reader r = new FileReader("/path/to/hotrod-client.properties")) {
    p.load(r);
    builder.withProperties(p);
}
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());

```

Additional resources

- [Hot Rod Client Configuration](#)
- [org.infinispan.client.hotrod.configuration.ConfigurationBuilder](#)
- [org.infinispan.client.hotrod.RemoteCacheManager](#)

3.2.1. Defining Data Grid Clusters in Client Configuration

Provide the locations of Data Grid clusters in Hot Rod client configuration.

Procedure

- Provide at least one Data Grid cluster name along with a host name and port for at least one node with the **ClusterConfigurationBuilder** class.
If you want to define a cluster as default, so that clients always attempt to connect to it first, then define a server list with the **addServers("<host_name>:<port>; <host_name>:<port>")** method.

Multiple cluster connections

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addCluster("siteA")
    .addClusterNode("hostA1", 11222)
    .addClusterNode("hostA2", 11222)
    .addCluster("siteB")
    .addClusterNodes("hostB1:11222; hostB2:11222");
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

Default server list with a failover cluster

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServers("hostA1:11222; hostA2:11222")
    .addCluster("siteB")
    .addClusterNodes("hostB1:11222; hostB2:11223");
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

3.2.2. Manually Switching Data Grid Clusters

Manually switch Hot Rod Java client connections between Data Grid clusters.

Procedure

- Call one of the following methods in the **RemoteCacheManager** class:
 - switchToCluster(clusterName)** switches to a specific cluster defined in the client configuration.
 - switchToDefaultCluster()** switches to the default cluster in the client configuration, which is defined as a list of Data Grid servers.

Additional resources

- [RemoteCacheManager](#)

3.2.3. Configuring Connection Pools

Hot Rod Java clients keep pools of persistent connections to Data Grid servers to reuse TCP connections instead of creating them on each request.

Procedure

- Configure Hot Rod client connection pool settings as in the following examples:

ConfigurationBuilder

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1")
    .port(11222)
    .connectionPool()
    .maxActive(10)
```

```

        exhaustedAction(ExhaustedAction.valueOf("WAIT"))
        .maxWait(1)
        .minIdle(20)
        .minEvictableIdleTime(300000)
        .maxPendingRequests(20);
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder.build());

```

hotrod-client.properties

```

infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.connection_pool.max_active = 10
infinispan.client.hotrod.connection_pool.exhausted_action = WAIT
infinispan.client.hotrod.connection_pool.max_wait = 1
infinispan.client.hotrod.connection_pool.min_idle = 20
infinispan.client.hotrod.connection_pool.min_evictable_idle_time = 300000
infinispan.client.hotrod.connection_pool.max_pending_requests = 20

```

3.3. CONFIGURING AUTHENTICATION MECHANISMS FOR HOT ROD CLIENTS

Data Grid Server uses different mechanisms to authenticate Hot Rod client connections.

Procedure

- Specify authentication mechanisms with the **saslMechanism()** method from the **AuthenticationConfigurationBuilder** class or with the **infinispan.client.hotrod.sasl_mechanism** property.

SCRAM

```

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1")
    .port(11222)
    .security()
    .authentication()
    .saslMechanism("SCRAM-SHA-512")
    .username("myuser")
    .password("qwer1234!");

```

DIGEST

```

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1")
    .port(11222)
    .security()
    .authentication()
    .saslMechanism("DIGEST-MD5")
    .username("myuser")
    .password("qwer1234!");

```

PLAIN

```

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1")
    .port(11222)
    .security()
    .authentication()
    .saslMechanism("PLAIN")
    .username("myuser")
    .password("qwer1234!");

```

OAuthbearer

```

String token = "..."; // Obtain the token from your OAuth2 provider
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1")
    .port(11222)
    .security()
    .authentication()
    .saslMechanism("OAuthbearer")
    .token(token);

```

EXTERNAL

```

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
    .host("127.0.0.1")
    .port(11222)
    .security()
    .ssl()
    // TrustStore stores trusted CA certificates for the server.
    .trustStoreFileName("/path/to/truststore")
    .trustStorePassword("truststorepassword".toCharArray())
    .trustStoreType("PKCS12")
    // KeyStore stores valid client certificates.
    .keyStoreFileName("/path/to/keystore")
    .keyStorePassword("keystorepassword".toCharArray())
    .keyStoreType("PKCS12")
    .authentication()
    .saslMechanism("EXTERNAL");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");

```

GSSAPI

```

LoginContext lc = new LoginContext("GssExample", new BasicCallbackHandler("krb_user",
"krb_password".toCharArray()));
lc.login();
Subject clientSubject = lc.getSubject();

```

```

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1")
    .port(11222)
    .security()
    .authentication()
    .saslMechanism("GSSAPI")
    .clientSubject(clientSubject)
    .callbackHandler(new BasicCallbackHandler());

```

Basic Callback Handler

The **BasicCallbackHandler**, as shown in the GSSAPI example, invokes the following callbacks:

- **NameCallback** and **PasswordCallback** construct the client subject.
- **AuthorizeCallback** is called during SASL authentication.

OAuthBearer with Token Callback Handler

Use a **TokenCallbackHandler** to refresh OAuth2 tokens before they expire, as in the following example:

```

String token = "..."; // Obtain the token from your OAuth2 provider
TokenCallbackHandler tokenHandler = new TokenCallbackHandler(token);
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1")
    .port(11222)
    .security()
    .authentication()
    .saslMechanism("OAUTHBEARER")
    .callbackHandler(tokenHandler);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
// Refresh the token
tokenHandler.setToken("newToken");

```

Custom CallbackHandler

Hot Rod clients set up a default **CallbackHandler** to pass credentials to SASL mechanisms. In some cases you might need to provide a custom **CallbackHandler**, as in the following example:

```

public class MyCallbackHandler implements CallbackHandler {
    final private String username;
    final private char[] password;
    final private String realm;

    public MyCallbackHandler(String username, String realm, char[] password) {
        this.username = username;
        this.password = password;
        this.realm = realm;
    }

    @Override
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            if (callback instanceof NameCallback) {
                NameCallback nameCallback = (NameCallback) callback;

```



```

        nameCallback.setName(username);
    } else if (callback instanceof PasswordCallback) {
        PasswordCallback passwordCallback = (PasswordCallback) callback;
        passwordCallback.setPassword(password);
    } else if (callback instanceof AuthorizeCallback) {
        AuthorizeCallback authorizeCallback = (AuthorizeCallback) callback;
        authorizeCallback.setAuthorized(authorizeCallback.getAuthenticationID().equals(
            authorizeCallback.getAuthorizationID()));
    } else if (callback instanceof RealmCallback) {
        RealmCallback realmCallback = (RealmCallback) callback;
        realmCallback.setText(realm);
    } else {
        throw new UnsupportedOperationException(callback);
    }
}
}
}
}

```

```

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1")
    .port(11222)
    .security().authentication()
    .serverName("myhotrodserver")
    .saslmMechanism("DIGEST-MD5")
    .callbackHandler(new MyCallbackHandler("myuser","default","qwer1234!".toCharArray()));

```



NOTE

A custom **CallbackHandler** needs to handle callbacks that are specific to the authentication mechanism that you use. However, it is beyond the scope of this document to provide examples for each possible callback type.

3.3.1. Creating GSSAPI Login Contexts

To use the GSSAPI mechanism, you must create a *LoginContext* so your Hot Rod client can obtain a Ticket Granting Ticket (TGT).

Procedure

1. Define a login module in a login configuration file.

gss.conf

```

GssExample {
    com.sun.security.auth.module.Krb5LoginModule required client=TRUE;
};

```

For the IBM JDK:

gss-ibm.conf

```
GssExample {
    com.ibm.security.auth.module.Krb5LoginModule required client=TRUE;
};
```

- Set the following system properties:

```
java.security.auth.login.config=gss.conf
```

```
java.security.krb5.conf=/etc/krb5.conf
```



NOTE

krb5.conf provides the location of your KDC. Use the *kinit* command to authenticate with Kerberos and verify **krb5.conf**.

3.3.2. SASL authentication mechanisms

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

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

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

3.4. CONFIGURING HOT ROD CLIENT ENCRYPTION

Data Grid Server can enforce SSL/TLS encryption and present Hot Rod clients with certificates to establish trust and negotiate secure connections.

To verify certificates issued to Data Grid Server, Hot Rod clients require either the full certificate chain or a partial chain that starts with the Root CA. You provide server certificates to Hot Rod clients as trust stores.

TIP

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

Prerequisites

- Create a trust store that Hot Rod clients can use to verify Data Grid Server identities.
- If you configure Data Grid Server to validate or authenticate client certificates, create a keystore as appropriate.

Procedure

1. Add the trust store to the client configuration with the **trustStoreFileName()** and **trustStorePassword()** methods or corresponding properties.
2. If you configure client certificate authentication, do the following:
 - a. Add the keystore to the client configuration with the **keyStoreFileName()** and **keyStorePassword()** methods or corresponding properties.
 - b. Configure clients to use the **EXTERNAL** authentication mechanism.

ConfigurationBuilder

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
    .host("127.0.0.1")
    .port(11222)
    .security()
    .ssl()
    // Server SNI hostname.
    .sniHostName("myservername")
    // Keystore that contains the public keys for Data Grid Server.
    // Clients use the trust store to verify Data Grid Server identities.
    .trustStoreFileName("/path/to/server/truststore")
    .trustStorePassword("truststorepassword".toCharArray())
    .trustStoreType("PKCS12")
    // Keystore that contains client certificates.
    // Clients present these certificates to Data Grid Server.
    .keyStoreFileName("/path/to/client/keystore")
    .keyStorePassword("keystorepassword".toCharArray())
    .keyStoreType("PKCS12")
    .authentication()
    // Clients must use the EXTERNAL mechanism for certificate authentication.
    .saslMechanism("EXTERNAL");
```

hotrod-client.properties

```
infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.use_ssl = true
infinispan.client.hotrod.sni_host_name = myservername
# Keystore that contains the public keys for Data Grid Server.
# Clients use the trust store to verify Data Grid Server identities.
infinispan.client.hotrod.trust_store_file_name = server_truststore.pkcs12
infinispan.client.hotrod.trust_store_password = changeme
infinispan.client.hotrod.trust_store_type = PKCS12
# Keystore that contains client certificates.
```

```
# Clients present these certificates to Data Grid Server.
infinispan.client.hotrod.key_store_file_name = client_keystore.pkcs12
infinispan.client.hotrod.key_store_password = changeme
infinispan.client.hotrod.key_store_type = PKCS12
# Clients must use the EXTERNAL mechanism for certificate authentication.
infinispan.client.hotrod.sasl_mechanism = EXTERNAL
```

Next steps

Add a client trust store to the **\$RHDG_HOME/server/conf** directory and configure Data Grid Server to use it, if necessary.

Additional resources

- [Encrypting Data Grid Server Connections](#)
- [SslConfigurationBuilder](#)
- [Hot Rod client configuration properties](#)
- [Using Shared System Certificates](#) (Red Hat Enterprise Linux 7 Security Guide)

3.5. ENABLING HOT ROD CLIENT STATISTICS

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

Procedure

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

Hot Rod Java client statistics

ConfigurationBuilder

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

hotrod-client.properties

```

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

```

3.6. NEAR CACHES

Near caches are local to Hot Rod clients and store recently used data so that every read operation does not need to traverse the network, which significantly increases performance.

Near caches:

- Are populated with read operations, calls to **get()** or **getVersioned()** methods. In the following example the **put()** call does not populate the near cache and only has the effect of invalidating the entry if it already exists:

```

cache.put("k1", "v1");
cache.get("k1");

```

- Register a client listener to invalidate entries when they are updated or removed in remote caches on Data Grid Server. If entries are requested after they are invalidated, clients must retrieve them from the remote caches again.
- Are cleared when clients fail over to different servers.

Bounded near caches

You should always use bounded near caches by specifying the maximum number of entries they can contain. When near caches reach the maximum number of entries, eviction automatically takes place to remove older entries. This means you do not need to manually keep the cache size within the boundaries of the client JVM.



IMPORTANT

Do not use maximum idle expiration with near caches because near-cache reads do not propagate the last access time for entries.

Bloom filters

Bloom filters optimize performance for write operations by reducing the total number of invalidation messages.

Bloom filters:

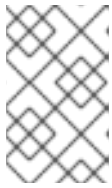
- Reside on Data Grid Server and keep track of the entries that the client has requested.
- Require a connection pool configuration that has a maximum of one active connection per server and uses the **WAIT** exhausted action.
- Cannot be used with unbounded near caches.

3.6.1. Configuring Near Caches

Configure Hot Rod Java clients with near caches to store recently used data locally in the client JVM.

Procedure

1. Open your Hot Rod Java client configuration.
2. Configure each cache to perform near caching with the `nearCacheMode(NearCacheMode.INVALIDATED)` method.



NOTE

Data Grid provides global near cache configuration properties. However, those properties are deprecated and you should not use them but configure near caching on a per-cache basis instead.

3. Specify the maximum number of entries that the near cache can hold before eviction occurs with the `nearCacheMaxEntries()` method.
4. Enable bloom filters for near caches with the `nearCacheUseBloomFilter()` method.

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.configuration.NearCacheMode;
import org.infinispan.client.hotrod.configuration.ExhaustedAction;
```

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer()
    .host("127.0.0.1")
    .port(ConfigurationProperties.DEFAULT_HOTROD_PORT)
    .security().authentication()
        .username("username")
        .password("password")
        .realm("default")
        .saslMechanism("SCRAM-SHA-512")
    // Configure the connection pool for bloom filters.
    .connectionPool()
        .maxActive(1)
        .exhaustedAction(ExhaustedAction.WAIT);
// Configure near caching for specific caches
builder.remoteCache("books")
    .nearCacheMode(NearCacheMode.INVALIDATED)
    .nearCacheMaxEntries(100)
    .nearCacheUseBloomFilter(false);
builder.remoteCache("authors")
    .nearCacheMode(NearCacheMode.INVALIDATED)
    .nearCacheMaxEntries(200)
    .nearCacheUseBloomFilter(true);
```

Additional resources

- [org.infinispan.client.hotrod.configuration.NearCacheConfiguration](https://infinispan.org/docs/client-hotrod/12.0.0.Final/org.infinispan.client.hotrod.configuration.NearCacheConfiguration)
- [org.infinispan.client.hotrod.configuration.ExhaustedAction](https://infinispan.org/docs/client-hotrod/12.0.0.Final/org.infinispan.client.hotrod.configuration.ExhaustedAction)

3.7. FORCING RETURN VALUES

To avoid sending data unnecessarily, write operations on remote caches return **null** instead of previous values.

For example, the following method calls do not return previous values for keys:

```
V remove(Object key);
V put(K key, V value);
```

You can, however, change the default behavior so your invocations return previous values for keys.

Procedure

- Configure Hot Rod clients so method calls return previous values for keys in one of the following ways:

FORCE_RETURN_VALUE flag

```
cache.withFlags(Flag.FORCE_RETURN_VALUE).put("aKey", "newValue")
```

Per-cache

```
ConfigurationBuilder builder = new ConfigurationBuilder();
// Return previous values for keys for invocations for a specific cache.
builder.remoteCache("mycache")
    .forceReturnValues(true);
```

hotrod-client.properties

```
# Use the "*" wildcard in the cache name to return previous values
# for all caches that start with the "somecaches" string.

infinispan.client.hotrod.cache.somecaches*.force_return_values = true
```

Additional resources

- [org.infinispan.client.hotrod.Flag](https://github.com/infinispan/infinispan-client-hotrod)

3.8. CREATING REMOTE CACHES FROM HOT ROD CLIENTS

Use the Data Grid Hot Rod API to create remote caches on Data Grid Server from Java, C++, .NET/C{hash}, JS clients and more.

This procedure shows you how to use Hot Rod Java clients that create remote caches on first access. You can find code examples for other Hot Rod clients in the [Data Grid Tutorials](#).

Prerequisites

- Create a Data Grid user with **admin** permissions.
- Start at least one Data Grid Server instance.
- Have a Data Grid cache configuration.

Procedure

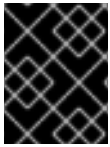
- Invoke the **remoteCache()** method as part of your the **ConfigurationBuilder**.
- Set the **configuration** or **configuration_uri** properties in the **hotrod-client.properties** file on your classpath.

ConfigurationBuilder

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

hotrod-client.properties

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



IMPORTANT

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

Additional resources

- [Hot Rod Client Configuration](#)
- [org.infinispan.client.hotrod.configuration.RemoteCacheConfigurationBuilder](#)

CHAPTER 4. HOT ROD CLIENT API

Data Grid Hot Rod client API provides interfaces for creating caches remotely, manipulating data, monitoring the topology of clustered caches, and more.

4.1. REMOTECACHE API

The collection methods **keySet**, **entrySet** and **values** are backed by the remote cache. That is that every method is called back into the **RemoteCache**. This is useful as it allows for the various keys, entries or values to be retrieved lazily, and not requiring them all be stored in the client memory at once if the user does not want.

These collections adhere to the **Map** specification being that **add** and **addAll** are not supported but all other methods are supported.

One thing to note is the **Iterator.remove** and **Set.remove** or **Collection.remove** methods require more than 1 round trip to the server to operate. You can check out the [RemoteCache](#) Javadoc to see more details about these and the other methods.

Iterator Usage

The iterator method of these collections uses **retrieveEntries** internally, which is described below. If you notice **retrieveEntries** takes an argument for the batch size. There is no way to provide this to the iterator. As such the batch size can be configured via system property **infinispan.client.hotrod.batch_size** or through the [ConfigurationBuilder](#) when configuring the **RemoteCacheManager**.

Also the **retrieveEntries** iterator returned is **Closeable** as such the iterators from **keySet**, **entrySet** and **values** return an **AutoCloseable** variant. Therefore you should always close these `Iterator`s` when you are done with them.

```
try (CloseableIterator<Map.Entry<K, V>> iterator = remoteCache.entrySet().iterator()) {
    }
}
```

What if I want a deep copy and not a backing collection?

Previous version of **RemoteCache** allowed for the retrieval of a deep copy of the **keySet**. This is still possible with the new backing map, you just have to copy the contents yourself. Also you can do this with **entrySet** and **values**, which we didn't support before.

```
Set<K> keysCopy = remoteCache.keySet().stream().collect(Collectors.toSet());
```

4.1.1. Unsupported Methods

The Data Grid **RemoteCache** API does not support all methods available in the **Cache** API and throws **UnsupportedOperationException** when unsupported methods are invoked.

Most of these methods do not make sense on the remote cache (e.g. listener management operations), or correspond to methods that are not supported by local cache as well (e.g. `containsValue`).

Certain atomic operations inherited from **ConcurrentMap** are also not supported with the **RemoteCache** API, for example:

```

boolean remove(Object key, Object value);
boolean replace(Object key, Object value);
boolean replace(Object key, Object oldValue, Object value);

```

However, **RemoteCache** offers alternative versioned methods for these atomic operations that send version identifiers over the network instead of whole value objects.

Reference

- [Cache](#)
- [RemoteCache](#)
- [UnsupportedOperationException](#)
- [ConcurrentMap](#)

4.2. REMOTE ITERATOR API

Data Grid provides a remote iterator API to retrieve entries where memory resources are constrained or if you plan to do server-side filtering or conversion.

```

// Retrieve all entries in batches of 1000
int batchSize = 1000;
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(null,
batchSize)) {
    while(iterator.hasNext()) {
        // Do something
    }
}

// Filter by segment
Set<Integer> segments = ...
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(null, segments,
batchSize)) {
    while(iterator.hasNext()) {
        // Do something
    }
}

// Filter by custom filter
try (CloseableIterator<Entry<Object, Object>> iterator =
remoteCache.retrieveEntries("myFilterConverterFactory", segments, batchSize)) {
    while(iterator.hasNext()) {
        // Do something
    }
}

```

4.2.1. Deploying Custom Filters to Data Grid Server

Deploy custom filters to Data Grid server instances.

Procedure

1. Create a factory that extends **KeyValueFilterConverterFactory**.

```
import java.io.Serializable;

import org.infinispan.filter.AbstractKeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverterFactory;
import org.infinispan.filter.NamedFactory;
import org.infinispan.metadata.Metadata;

//@NamedFactory annotation defines the factory name
@NamedFactory(name = "myFilterConverterFactory")
public class MyKeyValueFilterConverterFactory implements KeyValueFilterConverterFactory
{

    @Override
    public KeyValueFilterConverter<String, SampleEntity1, SampleEntity2>
    getFilterConverter() {
        return new MyKeyValueFilterConverter();
    }

    // Filter implementation. Should be serializable or externalizable for DIST caches
    static class MyKeyValueFilterConverter extends AbstractKeyValueFilterConverter<String,
    SampleEntity1, SampleEntity2> implements Serializable {
        @Override
        public SampleEntity2 filterAndConvert(String key, SampleEntity1 entity, Metadata
        metadata) {
            // returning null will case the entry to be filtered out
            // return SampleEntity2 will convert from the cache type SampleEntity1
        }

        @Override
        public MediaType format() {
            // returns the MediaType that data should be presented to this converter.
            // When omitted, the server will use "application/x-java-object".
            // Returning null will cause the filter/converter to be done in the storage format.
        }
    }
}
```

2. Create a JAR that contains a **META-INF/services/org.infinispan.filter.KeyValueFilterConverterFactory** file. This file should include the fully qualified class name of the filter factory class implementation. If the filter uses custom key/value classes, you must include them in your JAR file so that the filter can correctly unmarshall key and/or value instances.
3. Add the JAR file to the **server/lib** directory of your Data Grid server installation directory.

Reference

- [KeyValueFilterConverterFactory](#)

4.3. METADATAVALUE API

Use the **MetadataValue** interface for versioned operations.

The following example shows a remove operation that occurs only if the version of the value for the entry is unchanged:

```
RemoteCacheManager remoteCacheManager = new RemoteCacheManager();
RemoteCache<String, String> remoteCache = remoteCacheManager.getCache();

remoteCache.put("car", "ferrari");
VersionedValue valueBinary = remoteCache.getWithMetadata("car");

assert remoteCache.remove("car", valueBinary.getVersion());
assert !remoteCache.containsKey("car");
```

Reference

- org.infinispan.client.hotrod.MetadataValue

4.4. STREAMING API

Data Grid provides a Streaming API that implements methods that return instances of **InputStream** and **OutputStream** so you can stream large objects between Hot Rod clients and Data Grid servers.

Consider the following example of a large object:

```
StreamingRemoteCache<String> streamingCache = remoteCache.streaming();
OutputStream os = streamingCache.put("a_large_object");
os.write(...);
os.close();
```

You could read the object through streaming as follows:

```
StreamingRemoteCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
for(int b = is.read(); b >= 0; b = is.read()) {
    // iterate
}
is.close();
```

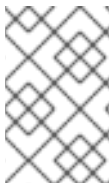


NOTE

The Streaming API does **not** marshal values, which means you cannot access the same entries using both the Streaming and Non-Streaming API at the same time. You can, however, implement a custom marshaller to handle this case.

The **InputStream** returned by the **RemoteStreamingCache.get(K key)** method implements the **VersionedMetadata** interface, so you can retrieve version and expiration information as follows:

```
StreamingRemoteCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
long version = ((VersionedMetadata) is).getVersion();
for(int b = is.read(); b >= 0; b = is.read()) {
    // iterate
}
is.close();
```

**NOTE**

Conditional write methods (**putIfAbsent()**, **replace()**) perform the actual condition check after the value is completely sent to the server. In other words, when the **close()** method is invoked on the **OutputStream**.

Reference

- [org.infinispan.client.hotrod.StreamingRemoteCache](#)

4.5. COUNTER API

The **CounterManager** interface is the entry point to define, retrieve and remove counters.

Hot Rod clients can retrieve the **CounterManager** interface as in the following example:

```
// create or obtain your RemoteCacheManager
RemoteCacheManager manager = ...;

// retrieve the CounterManager
CounterManager counterManager = RemoteCounterManagerFactory.asCounterManager(manager);
```

Reference

- [Clustered Counters](#)

4.6. CREATING EVENT LISTENERS

Java Hot Rod clients can register listeners to receive cache-entry level events. Cache entry created, modified and removed events are supported.

Creating a client listener is very similar to embedded listeners, except that different annotations and event classes are used. Here's an example of a client listener that prints out each event received:

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener(converterFactoryName = "static-converter")
public class EventPrintListener {

    @ClientCacheEntryCreated
    public void handleCreatedEvent(ClientCacheEntryCreatedEvent e) {
        System.out.println(e);
    }

    @ClientCacheEntryModified
    public void handleModifiedEvent(ClientCacheEntryModifiedEvent e) {
        System.out.println(e);
    }

    @ClientCacheEntryRemoved
    public void handleRemovedEvent(ClientCacheEntryRemovedEvent e) {
```

```

    System.out.println(e);
  }
}

```

ClientCacheEntryCreatedEvent and **ClientCacheEntryModifiedEvent** instances provide information on the affected key, and the version of the entry. This version can be used to invoke conditional operations on the server, such as **replaceWithVersion** or **removeWithVersion**.

ClientCacheEntryRemovedEvent events are only sent when the remove operation succeeds. In other words, if a remove operation is invoked but no entry is found or no entry should be removed, no event is generated. Users interested in removed events, even when no entry was removed, can develop event customization logic to generate such events. More information can be found in the [customizing client events section](#).

All **ClientCacheEntryCreatedEvent**, **ClientCacheEntryModifiedEvent** and **ClientCacheEntryRemovedEvent** event instances also provide a **boolean isCommandRetried()** method that will return true if the write command that caused this had to be retried again due to a topology change. This could be a sign that this event has been duplicated or another event was dropped and replaced (eg: ClientCacheEntryModifiedEvent replaced ClientCacheEntryCreatedEvent).

Once the client listener implementation has been created, it needs to be registered with the server. To do so, execute:

```

RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());

```

4.6.1. Removing Event Listeners

When an client event listener is not needed any more, it can be removed:

```

EventPrintListener listener = ...
cache.removeClientListener(listener);

```

4.6.2. Filtering Events

In order to avoid inundating clients with events, users can provide filtering functionality to limit the number of events fired by the server for a particular client listener. To enable filtering, a cache event filter factory needs to be created that produces filter instances:

```

import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-filter")
public static class StaticCacheEventFilterFactory implements CacheEventFilterFactory {

    @Override
    public StaticCacheEventFilter getFilter(Object[] params) {
        return new StaticCacheEventFilter();
    }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers

```

```
// needed when running in a cluster
class StaticCacheEventFilter implements CacheEventFilter<Integer, String>, Serializable {
    @Override
    public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
        String newValue, Metadata newMetadata, EventType eventType) {
        if (key.equals(1)) // static key
            return true;

        return false;
    }
}
```

The cache event filter factory instance defined above creates filter instances which statically filter out all entries except the one whose key is **1**.

To be able to register a listener with this cache event filter factory, the factory has to be given a unique name, and the Hot Rod server needs to be plugged with the name and the cache event filter factory instance.

1. Create a JAR file that contains the filter implementation.
If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the client listener has **useRawData** enabled, this is not necessary since the callback key/value instances will be provided in binary format.
2. Create a **META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory** file within the JAR file and within it, write the fully qualified class name of the filter class implementation.
3. Add the JAR file to the **server/lib** directory of your Data Grid server installation directory.
4. Link the client listener with this cache event filter factory by adding the factory name to the **@ClientListener** annotation:

```
@ClientListener(filterFactoryName = "static-filter")
public class EventPrintListener { ... }
```

5. Register the listener with the server:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());
```

You can also register dynamic filter instances that filter based on parameters provided when the listener is registered are also possible. Filters use the parameters received by the filter factories to enable this option, for example:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventFilter;

class DynamicCacheEventFilterFactory implements CacheEventFilterFactory {
    @Override
    public CacheEventFilter<Integer, String> getFilter(Object[] params) {
        return new DynamicCacheEventFilter(params);
    }
}
```



```

}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class DynamicCacheEventFilter implements CacheEventFilter<Integer, String>, Serializable {
    final Object[] params;

    DynamicCacheEventFilter(Object[] params) {
        this.params = params;
    }

    @Override
    public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
        String newValue, Metadata newMetadata, EventType eventType) {
        if (key.equals(params[0])) // dynamic key
            return true;

        return false;
    }
}

```

The dynamic parameters required to do the filtering are provided when the listener is registered:

```

RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), new Object[]{1}, null);

```



WARNING

Filter instances have to be marshallable when they are deployed in a cluster so that the filtering can happen right where the event is generated, even if the event is generated in a different node to where the listener is registered. To make them marshallable, either make them extend **Serializable**, **Externalizable**, or provide a custom **Externalizer** for them.

4.6.3. Skipping Notifications

Include the **SKIP_LISTENER_NOTIFICATION** flag when calling remote API methods to perform operations without getting event notifications from the server. For example, to prevent listener notifications when creating or modifying values, set the flag as follows:

```

remoteCache.withFlags(Flag.SKIP_LISTENER_NOTIFICATION).put(1, "one");

```

4.6.4. Customizing Events

The events generated by default contain just enough information to make the event relevant but they avoid cramming too much information in order to reduce the cost of sending them. Optionally, the information shipped in the events can be customised in order to contain more information, such as values, or to contain even less information. This customization is done with **CacheEventConverter** instances generated by a **CacheEventConverterFactory**:

```

import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-converter")
class StaticConverterFactory implements CacheEventConverterFactory {
    final CacheEventConverter<Integer, String, CustomEvent> staticConverter = new
StaticCacheEventConverter();
    public CacheEventConverter<Integer, String, CustomEvent> getConverter(final Object[] params) {
        return staticConverter;
    }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventConverter implements CacheEventConverter<Integer, String, CustomEvent>,
Serializable {
    public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata, String newValue,
Metadata newMetadata, EventType eventType) {
        return new CustomEvent(key, newValue);
    }
}

// Needs to be Serializable, Externalizable or marshallable with Infinispan Externalizers
// regardless of cluster or local caches
static class CustomEvent implements Serializable {
    final Integer key;
    final String value;
    CustomEvent(Integer key, String value) {
        this.key = key;
        this.value = value;
    }
}

```

In the example above, the converter generates a new custom event which includes the value as well as the key in the event. This will result in bigger event payloads compared with default events, but if combined with filtering, it can reduce its network bandwidth cost.



WARNING

The target type of the converter must be either **Serializable** or **Externalizable**. In this particular case of converters, providing an Externalizer will not work by default since the default Hot Rod client marshaller does not support them.

Handling custom events requires a slightly different client listener implementation to the one demonstrated previously. To be more precise, it needs to handle **ClientCacheEntryCustomEvent** instances:

```

import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

```

```

@ClientListener
public class CustomEventPrintListener {

    @ClientCacheEntryCreated
    @ClientCacheEntryModified
    @ClientCacheEntryRemoved
    public void handleCustomEvent(ClientCacheEntryCustomEvent<CustomEvent> e) {
        System.out.println(e);
    }
}

```

The **ClientCacheEntryCustomEvent** received in the callback exposes the custom event via **getEventData** method, and the **getType** method provides information on whether the event generated was as a result of cache entry creation, modification or removal.

Similar to filtering, to be able to register a listener with this converter factory, the factory has to be given a unique name, and the Hot Rod server needs to be plugged with the name and the cache event converter factory instance.

1. Create a JAR file with the converter implementation within it.
If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the client listener has **useRawData** enabled, this is not necessary since the callback key/value instances will be provided in binary format.
2. Create a **META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory** file within the JAR file and within it, write the fully qualified class name of the converter class implementation.
3. Add the JAR file to the **server/lib** directory of your Data Grid server installation directory.
4. Link the client listener with this converter factory by adding the factory name to the **@ClientListener** annotation:

```

@ClientListener(converterFactoryName = "static-converter")
public class CustomEventPrintListener { ... }

```

5. Register the listener with the server:

```

RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener());

```

Dynamic converter instances that convert based on parameters provided when the listener is registered are also possible. Converters use the parameters received by the converter factories to enable this option. For example:

```

import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

@NamedFactory(name = "dynamic-converter")
class DynamicCacheEventConverterFactory implements CacheEventConverterFactory {
    public CacheEventConverter<Integer, String, CustomEvent> getConverter(final Object[] params) {

```

```

    return new DynamicCacheEventConverter(params);
  }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed when running in a
// cluster
class DynamicCacheEventConverter implements CacheEventConverter<Integer, String,
CustomEvent>, Serializable {
  final Object[] params;

  DynamicCacheEventConverter(Object[] params) {
    this.params = params;
  }

  public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata,
    String newValue, Metadata newMetadata, EventType eventType) {
    // If the key matches a key given via parameter, only send the key information
    if (params[0].equals(key))
      return new CustomEvent(key, null);

    return new CustomEvent(key, newValue);
  }
}

```

The dynamic parameters required to do the conversion are provided when the listener is registered:

```

RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), null, new Object[]{1});

```



WARNING

Converter instances have to be marshallable when they are deployed in a cluster, so that the conversion can happen right where the event is generated, even if the event is generated in a different node to where the listener is registered. To make them marshallable, either make them extend **Serializable**, **Externalizable**, or provide a custom **Externalizer** for them.

4.6.5. Filter and Custom Events

If you want to do both event filtering and customization, it's easier to implement **org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter** which allows both filter and customization to happen in a single step. For convenience, it's recommended to extend **org.infinispan.notifications.cachelistener.filter.AbstractCacheEventFilterConverter** instead of implementing **org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter** directly. For example:

```

import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

@NamedFactory(name = "dynamic-filter-converter")

```

```

class DynamicCacheEventFilterConverterFactory implements CacheEventFilterConverterFactory {
    public CacheEventFilterConverter<Integer, String, CustomEvent> getFilterConverter(final Object[]
params) {
        return new DynamicCacheEventFilterConverter(params);
    }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed when running in a
// cluster
//
class DynamicCacheEventFilterConverter extends AbstractCacheEventFilterConverter<Integer,
String, CustomEvent>, Serializable {
    final Object[] params;

    DynamicCacheEventFilterConverter(Object[] params) {
        this.params = params;
    }

    public CustomEvent filterAndConvert(Integer key, String oldValue, Metadata oldMetadata,
String newValue, Metadata newMetadata, EventType eventType) {
        // If the key matches a key given via parameter, only send the key information
        if (params[0].equals(key))
            return new CustomEvent(key, null);

        return new CustomEvent(key, newValue);
    }
}

```

Similar to filters and converters, to be able to register a listener with this combined filter/converter factory, the factory has to be given a unique name via the **@NamedFactory** annotation, and the Hot Rod server needs to be plugged with the name and the cache event converter factory instance.

1. Create a JAR file with the converter implementation within it.
If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the client listener has **useRawData** enabled, this is not necessary since the callback key/value instances will be provided in binary format.
2. Create a **META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverterFactory** file within the JAR file and within it, write the fully qualified class name of the converter class implementation.
3. Add the JAR file to the **server/lib** directory of your Data Grid server installation directory.

From a client perspective, to be able to use the combined filter and converter class, the client listener must define the same filter factory and converter factory names, e.g.:

```

@ClientListener(filterFactoryName = "dynamic-filter-converter", converterFactoryName = "dynamic-
filter-converter")
public class CustomEventPrintListener { ... }

```

The dynamic parameters required in the example above are provided when the listener is registered via either filter or converter parameters. If filter parameters are non-empty, those are used, otherwise, the converter parameters:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener(), new Object[]{1}, null);
```

4.6.6. Event Marshalling

Hot Rod servers can store data in different formats, but in spite of that, Java Hot Rod client users can still develop **CacheEventConverter** or **CacheEventFilter** instances that work on typed objects. By default, filters and converter will use data as POJO (application/x-java-object) but it is possible to override the desired format by overriding the method **format()** from the filter/converter. If the format returns **null**, the filter/converter will receive data as it's stored.

Hot Rod Java clients can be configured to use different **org.infinispan.commons.marshall.Marshaller** instances. If doing this and deploying **CacheEventConverter** or **CacheEventFilter** instances, to be able to present filters/converter with Java Objects rather than marshalled content, the server needs to be able to convert between objects and the binary format produced by the marshaller.

To deploy a Marshaller instance server-side, follow a similar method to the one used to deploy **CacheEventConverter** or **CacheEventFilter** instances:

1. Create a JAR file with the converter implementation within it.
2. Create a **META-INF/services/org.infinispan.commons.marshall.Marshaller** file within the JAR file and within it, write the fully qualified class name of the marshaller class implementation.
3. Add the JAR file to the **server/lib** directory of your Data Grid server installation directory.

Note that the Marshaller could be deployed in either a separate jar, or in the same jar as the **CacheEventConverter** and/or **CacheEventFilter** instances.

4.6.6.1. Deploying Protostream Marshallers

If a cache stores Protobuf content, as it happens when using ProtoStream marshaller in the Hot Rod client, it's not necessary to deploy a custom marshaller since the format is already support by the server: there are transcoders from Protobuf format to most common formats like JSON and POJO.

When using filters/converters with those caches, and it's desirable to use filter/converters with Java Objects rather binary Protobuf data, it's necessary to configure the extra ProtoStreammarshallers so that the server can unmarshall the data before filtering/converting. To do so, you must configure the required **SerializationContextInitializer(s)** as part of the Data Grid server configuration.

See [Cache Encoding and Marshalling](#) for more information.

4.6.7. Listener State Handling

Client listener annotation has an optional **includeCurrentState** attribute that specifies whether state will be sent to the client when the listener is added or when there's a failover of the listener.

By default, **includeCurrentState** is false, but if set to true and a client listener is added in a cache already containing data, the server iterates over the cache contents and sends an event for each entry to the client as a **ClientCacheEntryCreated** (or custom event if configured). This allows clients to build some local data structures based on the existing content. Once the content has been iterated over, events are received as normal, as cache updates are received. If the cache is clustered, the entire cluster wide contents are iterated over.

4.6.8. Listener Failure Handling

When a Hot Rod client registers a client listener, it does so in a single node in a cluster. If that node fails, the Java Hot Rod client detects that transparently and fails over all listeners registered in the node that failed to another node.

During this fail over the client might miss some events. To avoid missing these events, the client listener annotation contains an optional parameter called **includeCurrentState** which if set to true, when the failover happens, the cache contents can be iterated over and **ClientCacheEntryCreated** events (or custom events if configured) are generated. By default, **includeCurrentState** is set to false.

Use callbacks to handle failover events:

```
@ClientCacheFailover
public void handleFailover(ClientCacheFailoverEvent e) {
    ...
}
```

This is very useful in use cases where the client has cached some data, and as a result of the fail over, taking in account that some events could be missed, it could decide to clear any locally cached data when the fail over event is received, with the knowledge that after the fail over event, it will receive events for the contents of the entire cache.

4.7. HOT ROD JAVA CLIENT TRANSACTIONS

You can configure and use Hot Rod clients in JTA [Transactions](#).

To participate in a transaction, the Hot Rod client requires the [TransactionManager](#) with which it interacts and whether it participates in the transaction through the [Synchronization](#) or [XAResource](#) interface.



IMPORTANT

Transactions are optimistic in that clients acquire write locks on entries during the prepare phase. To avoid data inconsistency, be sure to read about [Detecting Conflicts with Transactions](#).

4.7.1. Configuring the Server

Caches in the server must also be transactional for clients to participate in JTA [Transactions](#).

The following server configuration is required, otherwise transactions rollback only:

- Isolation level must be **REPEATABLE_READ**.
- **PESSIMISTIC** locking mode is recommended but **OPTIMISTIC** can be used.
- Transaction mode should be **NON_XA** or **NON_DURABLE_XA**. Hot Rod transactions should not use **FULL_XA** because it degrades performance.

For example:

```
<replicated-cache name="hotrodRepITx">
  <locking isolation="REPEATABLE_READ"/>
  <transaction mode="NON_XA" locking="PESSIMISTIC"/>
</replicated-cache>
```

Hot Rod transactions have their own recovery mechanism.

4.7.2. Configuring Hot Rod Clients

Transactional [RemoteCache](#) are configured per-cache basis. The exception is the transaction's **timeout** which is global, because a single transaction can interact with multiple [RemoteCaches](#).

The following example shows how to configure a transactional [RemoteCache](#) for cache **my-cache**:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
//other client configuration parameters
cb.transactionTimeout(1, TimeUnit.MINUTES);
cb.remoteCache("my-cache")
    .transactionManagerLookup(GenericTransactionManagerLookup.getInstance())
    .transactionMode(TransactionMode.NON_XA);
```

See [ConfigurationBuilder](#) and [RemoteCacheConfigurationBuilder](#) Javadoc for documentation on configuration parameters.

You can also configure the Java Hot Rod client with a properties file, as in the following example:

```
infinispan.client.hotrod.cache.my-cache.transaction.transaction_manager_lookup =
org.infinispan.client.hotrod.transaction.lookup.GenericTransactionManagerLookup
infinispan.client.hotrod.cache.my-cache.transaction.transaction_mode = NON_XA
infinispan.client.hotrod.transaction.timeout = 60000
```

4.7.2.1. TransactionManagerLookup Interface

TransactionManagerLookup provides an entry point to fetch a [TransactionManager](#).

Available implementations of **TransactionManagerLookup**:

[GenericTransactionManagerLookup](#)

A lookup class that locates [TransactionManagers](#) running in Java EE application servers. Defaults to the [RemoteTransactionManager](#) if it cannot find a [TransactionManager](#). This is the default for Hot Rod Java clients.

TIP

In most cases, [GenericTransactionManagerLookup](#) is suitable. However, you can implement the **TransactionManagerLookup** interface if you need to integrate a custom [TransactionManager](#).

[RemoteTransactionManagerLookup](#)

A basic, and volatile, [TransactionManager](#) if no other implementation is available. Note that this implementation has significant limitations when handling concurrent transactions and recovery.

4.7.3. Transaction Modes

[TransactionMode](#) controls how a [RemoteCache](#) interacts with the [TransactionManager](#).



IMPORTANT

Configure transaction modes on both the Data Grid server and your client application. If clients attempt to perform transactional operations on non-transactional caches, runtime exceptions can occur.

Transaction modes are the same in both the Data Grid configuration and client settings. Use the following modes with your client, see the Data Grid configuration schema for the server:

NONE

The `RemoteCache` does not interact with the `TransactionManager`. This is the default mode and is non-transactional.

NON_XA

The `RemoteCache` interacts with the `TransactionManager` via `Synchronization`.

NON_DURABLE_XA

The `RemoteCache` interacts with the `TransactionManager` via `XAResource`. Recovery capabilities are disabled.

FULL_XA

The `RemoteCache` interacts with the `TransactionManager` via `XAResource`. Recovery capabilities are enabled. Invoke the `XaResource.recover()` method to retrieve transactions to recover.

4.7.4. Detecting Conflicts with Transactions

Transactions use the initial values of keys to detect conflicts.

For example, "k" has a value of "v" when a transaction begins. During the prepare phase, the transaction fetches "k" from the server to read the value. If the value has changed, the transaction rolls back to avoid a conflict.



NOTE

Transactions use versions to detect changes instead of checking value equality.

The `forceReturnValue` parameter controls write operations to the `RemoteCache` and helps avoid conflicts. It has the following values:

- If **true**, the `TransactionManager` fetches the most recent value from the server before performing write operations. However, the `forceReturnValue` parameter applies only to write operations that access the key for the first time.
- If **false**, the `TransactionManager` does not fetch the most recent value from the server before performing write operations.



NOTE

This parameter does not affect *conditional* write operations such as `replace` or `putIfAbsent` because they require the most recent value.

The following transactions provide an example where the `forceReturnValue` parameter can prevent conflicting write operations:

Transaction 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v1");
tm.commit();
```

Transaction 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v2");
tm.commit();
```

In this example, TX1 and TX2 are executed in parallel. The initial value of "k" is "v".

- If **forceReturnValue = true**, the **cache.put()** operation fetches the value for "k" from the server in both TX1 and TX2. The transaction that acquires the lock for "k" first then commits. The other transaction rolls back during the commit phase because the transaction can detect that "k" has a value other than "v".
- If **forceReturnValue = false**, the **cache.put()** operation does not fetch the value for "k" from the server and returns null. Both TX1 and TX2 can successfully commit, which results in a conflict. This occurs because neither transaction can detect that the initial value of "k" changed.

The following transactions include **cache.get()** operations to read the value for "k" before doing the **cache.put()** operations:

Transaction 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v1");
tm.commit();
```

Transaction 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v2");
tm.commit();
```

In the preceding examples, TX1 and TX2 both read the key so the **forceReturnValue** parameter does not take effect. One transaction commits, the other rolls back. However, the **cache.get()** operation requires an additional server request. If you do not need the return value for the **cache.put()** operation that server request is inefficient.

4.7.5. Using the Configured Transaction Manager and Transaction Mode

The following example shows how to use the **TransactionManager** and **TransactionMode** that you configure in the **RemoteCacheManager**:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.remoteCache("my-cache")
    .transactionManagerLookup(RemoteTransactionManagerLookup.getInstance())
    .transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//The my-cache instance uses the RemoteCacheManager configuration.
RemoteCache<String, String> cache = rcm.getCache("my-cache");

//Return the transaction manager that the cache uses.
TransactionManager tm = cache.getTransactionManager();

//Perform a simple transaction.
tm.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
tm.commit();
```