Hot Rod Java Client Guide

Table of Contents

Chapter 1. Hot Rod Java Clients

Access Infinispan remotely through the Hot Rod Java client API.

1.1. Hot Rod Protocol

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

- Load balancing. Hot Rod clients can send requests across Infinispan clusters using different strategies.
- Failover. Hot Rod clients can monitor Infinispan 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. Configuring the Infinispan Maven Repository

Infinispan Java distributions are available from Maven.

Infinispan artifacts are available from Maven central. See the org.infinispan group for available Infinispan artifacts.

1.2.1. Configuring Your Infinispan 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 Infinispan 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 Infinispan artifact you add as a dependency to your project.

4. Save and close `pom.xml`.

The following example shows the Infinispan version and BOM:

```
<properties>
  <version.infinispan>12.0.0.CR1</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 Infinispan artifacts as dependencies to your `pom.xml` as required.

1.3. Getting the Hot Rod Java Client

Add the Hot Rod Java client to your project.

Prerequisites

Hot Rod Java clients can use 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

Infinispan Server Requirements

Chapter 2. Configuring Hot Rod Java Clients

2.1. Programmatically Configuring Hot Rod Java Clients

Use the `ConfigurationBuilder` class to generate immutable configuration objects that you can pass to `RemoteCacheManager`.

For example, create a client instance with the Java fluent API as follows:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
      = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.marshaller(new org.infinispan.commons.marshall.ProtoStreamMarshaller())
  .statistics()
      .enable()
      .jmxDomain("org.example")
  .addServer()
      .host("127.0.0.1")
      .port(11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

Reference

org.infinispan.client.hotrod.configuration.ConfigurationBuilder

2.2. Configuring Hot Rod Java Client Property Files

Add `hotrod-client.properties` to your classpath so that the client passes configuration to `RemoteCacheManager`.

Example hotrod-client.properties

```
# Hot Rod client configuration
infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.marshaller =
org.infinispan.commons.marshall.ProtoStreamMarshaller
infinispan.client.hotrod.async_executor_factory =
org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory
infinispan.client.hotrod.default_executor_factory.pool_size = 1
infinispan.client.hotrod.hash_function_impl.2 =
org.infinispan.client.hotrod.impl.consistenthash.ConsistentHashV2
infinispan.client.hotrod.tcp_no_delay = true
infinispan.client.hotrod.tcp_keep_alive = false
infinispan.client.hotrod.request_balancing_strategy =
org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy
infinispan.client.hotrod.key_size_estimate = 64
infinispan.client.hotrod.value_size_estimate = 512
infinispan.client.hotrod.force_return_values = false

## Connection pooling configuration
maxActive = -1
maxIdle = -1
whenExhaustedAction = 1
minEvictableIdleTimeMillis=300000
minIdle = 1
```

To use `hotrod-client.properties` somewhere other than your classpath, do:

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

Reference

- Hot Rod Client Configuration
- org.infinispan.client.hotrod.RemoteCacheManager
- Java system properties

2.3. Client Intelligence

Hot Rod client intelligence refers to mechanisms for locating Infinispan servers to efficiently route requests.

Basic intelligence

Clients do not store any information about Infinispan clusters or key hash values.

Topology-aware

Clients receive and store information about Infinispan clusters. Clients maintain an internal mapping of the cluster topology that changes whenever servers join or leave clusters.

To receive a cluster topology, clients need the address (`IP:HOST`) of at least one Hot Rod server at startup. After the client connects to the server, Infinispan transmits the topology to the client. When servers join or leave the cluster, Infinispan transmits an updated topology to the client.

Distribution-aware

Clients are topology-aware and store consistent hash values for keys.

For example, take a `put(k,v)` operation. The client calculates the hash value for the key so it can locate the exact server on which the data resides. Clients can then connect directly to the owner to dispatch the operation.

The benefit of distribution-aware intelligence is that Infinispan servers do not need to look up values based on key hashes, which uses less resources on the server side. Another benefit is that servers respond to client requests more quickly because it skips additional network roundtrips.

2.3.1. Request Balancing

Clients that use topology-aware intelligence use request balancing for all requests. The default balancing strategy is round-robin, so topology-aware clients always send requests to servers in round-robin order.

For example, `s1`, `s2`, `s3` are servers in a Infinispan cluster. Clients perform request balancing as follows:

```
CacheContainer cacheContainer = new RemoteCacheManager();
Cache<String, String> cache = cacheContainer.getCache();

//client sends put request to s1
cache.put("key1", "aValue");
//client sends put request to s2
cache.put("key2", "aValue");
//client sends get request to s3
String value = cache.get("key1");
//client dispatches to s1 again
cache.remove("key2");
//and so on...
```

Clients that use distribution-aware intelligence use request balancing only for failed requests. When requests fail, distribution-aware clients retry the request on the next available server.

Custom balancing policies

You can implement FailoverRequestBalancingStrategy and specify your class in your `hotrod-client.properties` configuration with the following property:

`infinispan.client.hotrod.request_balancing_strategy`

2.3.2. Client Failover

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

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

For example, you have a Infinispan 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.

2.4. Configuring Authentication Mechanisms for Hot Rod Clients

Infinispan servers use different mechanisms to authenticate Hot Rod client connections.

Procedure

- Specify the authentication mechanisms that Infinispan server uses with the `saslMechanism()` method from the `SecurityConfigurationBuilder` class.

SCRAM

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .username("myuser")
         .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

DIGEST

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .saslMechanism("DIGEST-MD5")
         .username("myuser")
         .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

PLAIN

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .saslMechanism("PLAIN")
         .username("myuser")
         .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

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);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

OAUTHBEARER authentication with TokenCallbackHandler

You can configure clients with 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");
```

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())
         // KeyStore stores valid client certificates.
         .keyStoreFileName("/path/to/keystore")
         .keyStorePassword("keystorepassword".toCharArray())
      .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()
         .enable()
         .saslMechanism("GSSAPI")
         .clientSubject(clientSubject)
         .callbackHandler(new BasicCallbackHandler());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

The preceding configuration uses the `BasicCallbackHandler` to retrieve the client subject and handle authentication. However, this actually invokes different callbacks:

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

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`.

Your `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.

```
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 UnsupportedCallbackException(callback);
         }
      }
   }
}

   ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .enable()
         .serverName("myhotrodserver")
         .saslMechanism("DIGEST-MD5")
         .callbackHandler(new MyCallbackHandler("myuser","default","qwer1234!"
.toCharArray()));
remoteCacheManager=new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache=remoteCacheManager.getCache("secured");
```

2.4.1. Hot Rod Endpoint Authentication Mechanisms

Infinispan supports the following SASL authentications mechanisms with the Hot Rod connector:

Authentication mechanism	Description	Related details
`PLAIN`	Uses credentials in plain-text format. You should use `PLAIN` authentication with encrypted connections only.	Similar to the `Basic` HTTP mechanism.
`DIGEST-*`	Uses hashing algorithms and nonce values. Hot Rod connectors support `DIGEST-MD5`, `DIGEST-SHA`, `DIGEST-SHA-256`, `DIGEST-SHA-384`, and `DIGEST-SHA-512` hashing algorithms, in order of strength.	Similar to the `Digest` HTTP mechanism.
`SCRAM-*`	Uses *salt* values in addition to hashing algorithms and nonce values. Hot Rod connectors support `SCRAM-SHA`, `SCRAM-SHA-256`, `SCRAM-SHA-384`, and `SCRAM-SHA-512` hashing algorithms, in order of strength.	Similar to the `Digest` HTTP mechanism.
`GSSAPI`	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding `kerberos` server identity in the realm configuration. In most cases, you also specify an `ldap-realm` to provide user membership information.	Similar to the `SPNEGO` HTTP mechanism.
`GS2-KRB5`	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding `kerberos` server identity in the realm configuration. In most cases, you also specify an `ldap-realm` to provide user membership information.	Similar to the `SPNEGO` HTTP mechanism.
`EXTERNAL`	Uses client certificates.	Similar to the `CLIENT_CERT` HTTP mechanism.
`OAUTHBEARER`	Uses OAuth tokens and requires a `token-realm` configuration.	Similar to the `BEARER_TOKEN` HTTP mechanism.

2.4.2. 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;
   };
   ```

2. Set the following system properties:

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

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

   

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

2.5. Configuring Hot Rod Client Encryption

Infinispan servers that use SSL/TLS encryption present Hot Rod clients with certificates so they can establish trust and negotiate secure connections.

To verify server-issued certificates, Hot Rod clients require part of the TLS certificate chain. For example, the following image shows a certificate authority (CA), named "CA", that has issued a certificate for a server named "HotRodServer":

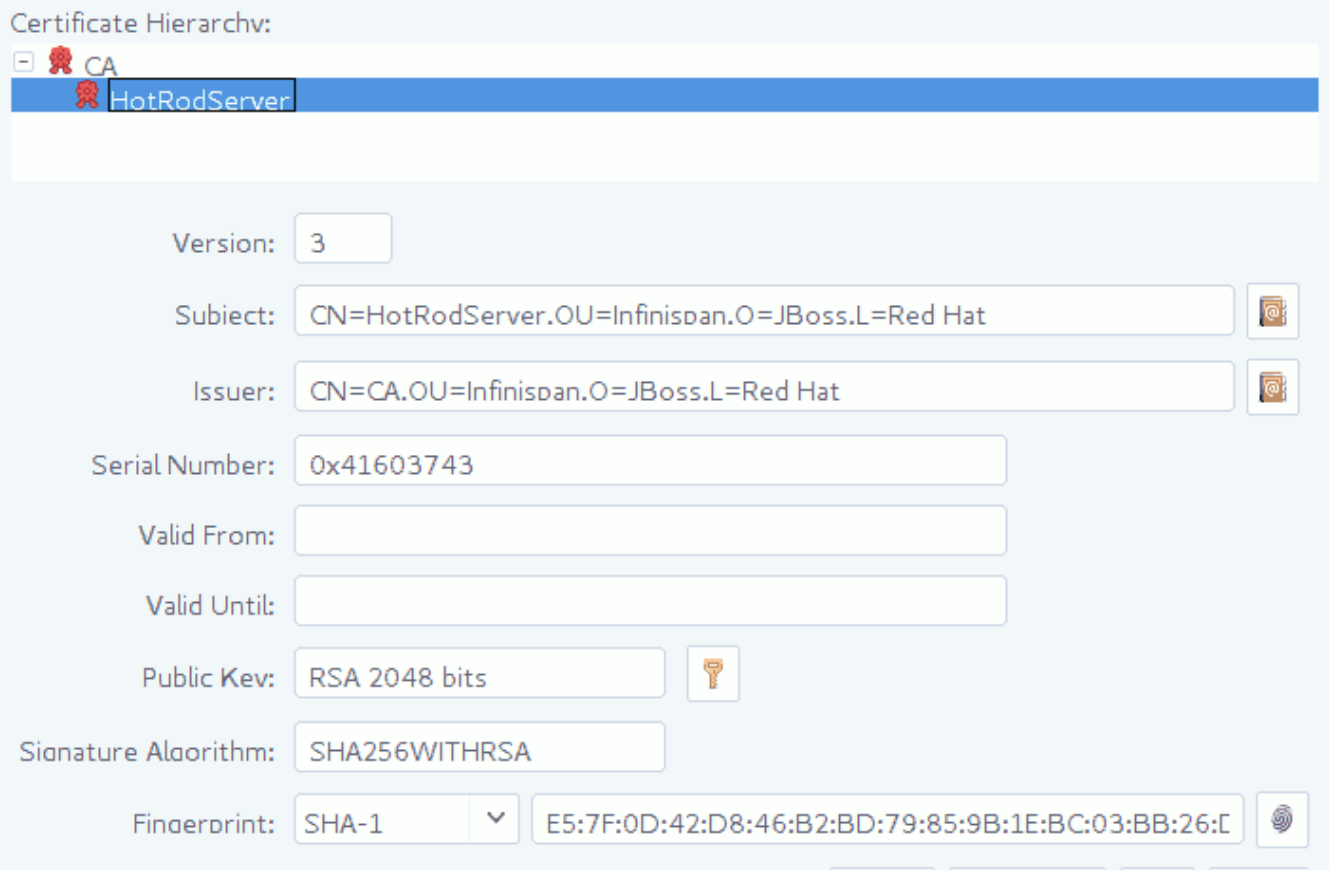

Figure 1. Certificate chain

Procedure

1. Create a Java keystore with part of the server certificate chain. In most cases you should use the public certificate for the CA.
2. Specify the keystore as a *TrustStore* in the client configuration with the `SslConfigurationBuilder` class.

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         // Server SNI hostname.
         .sniHostName("myservername")
         // Server certificate keystore.
         .trustStoreFileName("/path/to/truststore")
         .trustStorePassword("truststorepassword".toCharArray())
         // Client certificate keystore.
         .keyStoreFileName("/path/to/client/keystore")
         .keyStorePassword("keystorepassword".toCharArray());
RemoteCache<String, String> cache=remoteCacheManager.getCache("secured");
```

Specify a path that contains certificates in PEM format and Hot Rod clients automatically generate trust stores.

Use `.trustStorePath("/path/to/certificate")`.

2.6. Monitoring Hot Rod Client Statistics

Enable Hot Rod client statistics that include remote and near-cache hits and misses as well as connection pool usage.

Procedure

- Use the `StatisticsConfigurationBuilder` class to enable and configure Hot Rod client statistics.

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

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .statistics()
    //Enable client statistics.
    .enable()
   //Register JMX MBeans for RemoteCacheManager and each RemoteCache.
   .jmxEnable()
    //Set JMX domain name to which MBeans are exposed.
    .jmxDomain("org.example")
   .addServer()
    .host("127.0.0.1")
    .port(11222);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

2.7. Defining Infinispan Clusters in Client Configuration

Provide the locations of Infinispan clusters in Hot Rod client configuration.

Procedure

- Provide at least one Infinispan cluster name, hostname, and port with the `ClusterConfigurationBuilder` class.

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

Default Cluster

When adding clusters to your Hot Rod client configuration, you can define a list of Infinispan servers in the format of `hostname1:port; hostname2:port`. Infinispan then uses the server list as the default cluster configuration.

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

2.7.1. Manually Switching Infinispan Clusters

Manually switch Hot Rod Java client connections between Infinispan 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 Infinispan servers.

Reference

RemoteCacheManager

2.8. Creating Caches with Hot Rod Clients

Programmatically create caches on Infinispan Server through the `RemoteCacheManager` API.

The following procedure demonstrates programmatic cache creation with the Hot Rod Java client. However Hot Rod clients are available in different languages such as Javascript or C++.

Prerequisites

- Create a user and start at least one Infinispan server instance.
- Get the Hot Rod Java client.

Procedure

1. Configure your client with the `ConfigurationBuilder` class.

```
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.DefaultTemplate;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.commons.configuration.XMLStringConfiguration;
...

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer()
         .host("127.0.0.1")
         .port(11222)
       .security().authentication()
          .enable()
          .username("username")
          .password("password")
          .realm("default")
          .saslMechanism("DIGEST-MD5");

manager = new RemoteCacheManager(builder.build());
```

2. Use the `XMLStringConfiguration` class to add cache definitions in XML format.
3. Call the `getOrCreateCache()` method to add the cache if it already exists or create it if not.

```
private void createCacheWithXMLConfiguration() {
    String cacheName = "CacheWithXMLConfiguration";
    String xml = String.format("<distributed-cache name=\"%s\" mode=\"SYNC\"
                                  statistics=\"true\">" +
                                 "<locking isolation=\"READ_COMMITTED\"/>" +
                                 "<transaction mode=\"NON_XA\"/>" +
                                 "<expiration lifespan=\"60000\"
interval=\"20000\"/>" +
                               "</distributed-cache>" +
                               , cacheName);
    manager.administration().getOrCreateCache(cacheName, new
XMLStringConfiguration(xml));
    System.out.println("Cache created or already exists.");
}
```

4. Create caches with `org.infinispan` templates as in the following example with the `createCache()` invocation:

```
private void createCacheWithTemplate() {
    manager.administration().createCache("myCache", "org.infinispan.DIST_SYNC");
    System.out.println("Cache created.");
}
```

Next Steps

Try some working code examples that show you how to create remote caches with the Hot Rod Java client. Visit the Infinispan Tutorials.

Reference

- `RemoteCacheManager` Javadoc
- Getting the Hot Rod Java Client

2.9. Creating Caches on First Access

When Hot Rod Java clients attempt to access caches that do not exist, they return `null` for `getCache("$cacheName")` invocations.

You can change this default behavior so that clients automatically create caches on first access using default configuration templates or Infinispan cache definitions in XML format.

Programmatic procedure

- Use the `remoteCache()` method to create per-cache configurations in the Hot Rod `ConfigurationBuilder` class as follows:

```
import org.infinispan.client.hotrod.DefaultTemplate;
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
...

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.remoteCache("my-cache") ①
    .templateName(DefaultTemplate.DIST_SYNC)
builder.remoteCache("another-cache") ②
    .configuration("<infinispan><cache-container><distributed-cache name=\"another-
cache\"/></cache-container></infinispan>");
builder.remoteCache("my-other-cache") ③
    .configurationURI(URI.create("file:/path/to/configuration.xml"));
```

① Creates a cache named "my-cache" from the `org.infinispan.DIST_SYNC` template.

② Creates a cache named "another-cache" from an XML definition.

③ Creates a cache named "my-other-cache" from an XML file.

Hot Rod client properties

- Add `infinispan.client.hotrod.cache.<cache-name>` properties to your `hotrod-client.properties` file to create per-cache configurations as follows:

```
infinispan.client.hotrod.cache.my-cache.template_name=org.infinispan.DIST_SYNC ①
infinispan.client.hotrod.cache.another-cache.configuration=<infinispan><cache-
container><distributed-cache name=\"another-cache\"/></cache-container></infinispan>
②
infinispan.client.hotrod.cache.my-other-
cache.configuration_uri=file:/path/to/configuration.xml ③
```

① Creates a cache named "my-cache" from the `org.infinispan.DIST_SYNC` template.

② Creates a cache named "another-cache" from an XML definition.

③ Creates a cache named "my-other-cache" from an XML file.

Reference

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

2.10. Creating Per-Cache Configurations

In addition to creating caches on first access, you can remotely configure certain aspects of individual caches such as:

- Force return values
- Near-caching
- Transaction modes

Procedure

- Enable *force return values* for a cache named `a-cache` as follows:

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

ConfigurationBuilder builder = new ConfigurationBuilder();
builder
  .remoteCache("a-cache")
    .forceReturnValues(true);
```

- Use wildcard globbing in the remote cache name to enable force return values for all caches that start with the string `somecaches`:

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

ConfigurationBuilder builder = new ConfigurationBuilder();
builder
  .remoteCache("somecaches*")
    .forceReturnValues(true);
```

When using declarative configuration and your cache names contain the `.` character, you must enclose the cache name in square brackets, for example `infinispan.client.hotrod.cache.[example.MyCache].template=⋯`

2.11. Configuring Near Caching

Hot Rod Java clients can keep local caches that store recently used data, which significantly increases performance of `get()` and `getVersioned()` operations because the data is local to the client.

When you enable near caching with Hot Rod Java clients, calls to `get()` or `getVersioned()` calls populate the near cache when entries are retrieved from servers. When entries are updated or removed on the server-side, entries in the near cache are invalidated. If keys are requested after they are invalidated, clients must fetch the keys from the server again.

You can also configure the number of entries that near caches can contain. When the maximum is reached, near-cached entries are evicted.

Near cache considerations

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

- Near caches are cleared when clients failover to different servers when using clustered cache modes.
- You should always configure the maximum number of entries that can reside in the near cache. Unbounded near caches require you to keep the size of the near cache within the boundaries of the client JVM.
- Near cache invalidation messages can degrade performance of write operations. The near cache can be configured with a bloom filter to significantly reduce this degredation. However bloom filter can only be enabled with a bounded near cache.

Bloom filter near cache requires the pool configuration to also have a maximum of 1 active connection per server and utilize the WAIT exhausted action. This is required so the server can keep the bloom filter updated on the server side to which listener was registered.

Procedure

1. Set the near cache mode to `INVALIDATED` in the client configuration for the caches you want
2. Define the size of the near cache by specifying the maximum number of entries.

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

// Configure different near cache settings for specific caches
ConfigurationBuilder builder = new ConfigurationBuilder();
builder
  .remoteCache("bounded")
    .nearCacheMode(NearCacheMode.INVALIDATED)
    .nearCacheMaxEntries(100)
    .bloomFilter(false);
  .remoteCache("unbounded").nearCache()
    .nearCacheMode(NearCacheMode.INVALIDATED)
    .nearCacheMaxEntries(-1);

// Following required for bloom filter near cache
// builder
//    .connectionPool()
//       .maxActive(1)
//       .exhaustedAction(ExhaustedAction.WAIT);
```

You should always configure near caching on a per-cache basis. Even though Infinispan provides global near cache configuration properties, you should not use them.

= 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 change this default behavior with the `FORCE_RETURN_VALUE` flag so your invocations return previous values.

Procedure

- Use the `FORCE_RETURN_VALUE` flag to get previous values instead of `null` as in the following example:

```
cache.put("aKey", "initialValue");
assert null == cache.put("aKey", "aValue");
assert "aValue".equals(cache.withFlags(Flag.FORCE_RETURN_VALUE).put("aKey",
   "newValue"));
```

Reference

org.infinispan.client.hotrod.Flag

= Configuring Connection Pools Hot Rod Java clients keep pools of persistent connections to Infinispan servers to reuse TCP connections instead of creating them on each request.

Clients use asynchronous threads that check the validity of connections by iterating over the connection pool and sending pings to Infinispan servers. This improves performance by finding broken connections while they are idle in the pool, rather than on application requests.

Procedure

- Configure Hot Rod client connection pool settings with the `ConnectionPoolConfigurationBuilder` class.

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
    .host("127.0.0.1")
    .port(11222)
    //Configure client connection pools.
    .connectionPool()
    //Set the maximum number of active connections per server.
    .maxActive(10)
    //Set the minimum number of idle connections
    //that must be available per server.
    .minIdle(20);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

= Hot Rod Java Client Marshalling Hot Rod is a binary TCP protocol that requires you to transform Java objects into binary format so they can be transferred over the wire or stored to disk.

By default, Infinispan uses a ProtoStream API to encode and decode Java objects into Protocol Buffers (Protobuf); a language-neutral, backwards compatible format. However, you can also implement and use custom marshallers.

Reference

- Marshalling Java Objects

- Using the ProtoStream Marshaller

= Configuring SerializationContextInitializer Implementations You can add implementations of the ProtoStream `SerializationContextInitializer` interface to Hot Rod client configurations so Infinispan marshalls custom Java objects.

Procedure

- Add your `SerializationContextInitializer` implementations to your Hot Rod client configuration as follows:

`hotrod-client.properties`

```
infinispan.client.hotrod.context-initializers=org.infinispan.example
.LibraryInitializerImpl,org.infinispan.example.AnotherExampleSciImpl
```

Programmatic configuration

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder
      .addServer()
      .host("127.0.0.1")
      .port(11222)
      .addContextInitializers(new LibraryInitializerImpl(), new
AnotherExampleSciImpl());
RemoteCacheManager rcm = new RemoteCacheManager(builder.build());
```

- Using the ProtoStream Marshaller
- ProtoStream Serialization Contexts

= Configuring Custom Marshallers Configure Hot Rod clients to use custom marshallers.

Procedure

1. Implement the `org.infinispan.commons.marshall.Marshaller` interface.
2. Specify the fully qualified name of your class in your Hot Rod client configuration.
3. Add your Java classes to the Infinispan deserialization allow list.

   In the following example, only classes with fully qualified names that contain `Person` or `Employee` are allowed:

   ```
   ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
   clientBuilder.marshaller("org.infinispan.example.marshall.CustomMarshaller")
     .addJavaSerialAllowList(".*Person.*", ".*Employee.*");
     ...
   ```

Reference

- org.infinispan.commons.marshall.Marshaller

- Using Custom Marshallers
- Adding Java Classes to Deserialization Allow Lists

= Configuring Hot Rod Client Data Formats By default, Hot Rod client operations use the configured marshaller when reading and writing from Infinispan servers for both keys and values.

However, the `DataFormat` API lets you decorate remote caches so that all operations can happen with custom data formats.

Using different marshallers for key and values

Marshallers for keys and values can be overridden at run time. For example, to bypass all serialization in the Hot Rod client and read the `byte[]` as they are stored in the server:

```
// Existing RemoteCache instance
RemoteCache<String, Pojo> remoteCache = ...

// IdentityMarshaller is a no-op marshaller
DataFormat rawKeyAndValues =
DataFormat.builder()
          .keyMarshaller(IdentityMarshaller.INSTANCE)
          .valueMarshaller(IdentityMarshaller.INSTANCE)
          .build();

// Creates a new instance of RemoteCache with the supplied DataFormat
RemoteCache<byte[], byte[]> rawResultsCache =
remoteCache.withDataFormat(rawKeyAndValues);
```

Using different marshallers and formats for keys, with `keyMarshaller()` and `keyType()` methods might interfere with the client intelligence routing mechanism and cause extra hops within the Infinispan cluster to perform the operation. If performance is critical, you should use keys in the format stored by the server.

Returning XML Values

```
Object xmlValue = remoteCache
      .withDataFormat(DataFormat.builder()
      .valueType(APPLICATION_XML)
      .valueMarshaller(new UTF8StringMarshaller())
      .build())
      .get(key);
```

The preceding code example returns XML values as follows:

```
<?xml version="1.0" ?><string>Hello!</string>
```

Reading data in different formats

Request and send data in different formats specified by a `org.infinispan.commons.dataconversion.MediaType` as follows:

```
// Existing remote cache using ProtostreamMarshaller
RemoteCache<String, Pojo> protobufCache = ...

// Request values returned as JSON
// Use the UTF8StringMarshaller to convert UTF-8 to String
DataFormat jsonString =
DataFormat.builder()
          .valueType(MediaType.APPLICATION_JSON)
          .valueMarshaller(new UTF8StringMarshaller())
          .build();
RemoteCache<byte[], byte[]> rawResultsCache =
protobufCache.withDataFormat(jsonString);
```

```
// Alternatively, use a custom value marshaller
// that returns `org.codehaus.jackson.JsonNode` objects
DataFormat jsonNode =
DataFormat.builder()
          .valueType(MediaType.APPLICATION_JSON)
          .valueMarshaller(new CustomJacksonMarshaller()
          .build();

RemoteCache<String, JsonNode> jsonNodeCache =
remoteCache.withDataFormat(jsonNode);
```

In the preceding example, data conversion happens in the Infinispan server. Infinispan throws an exception if it does not support conversion to and from a storage format.

Reference

org.infinispan.client.hotrod.DataFormat

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

= Basic API Below is a sample code snippet on how the client API can be used to store or retrieve information from a Infinispan server using the Java Hot Rod client. It assumes that a Infinispan server is running at `localhost:11222`.

```
//API entry point, by default it connects to localhost:11222
CacheContainer cacheContainer = new RemoteCacheManager();

//obtain a handle to the remote default cache
Cache<String, String> cache = cacheContainer.getCache();

//now add something to the cache and make sure it is there
cache.put("car", "ferrari");
assert cache.get("car").equals("ferrari");

//remove the data
cache.remove("car");
assert !cache.containsKey("car") : "Value must have been removed!";
```

The client API maps the local API: RemoteCacheManager corresponds to DefaultCacheManager (both implement CacheContainer). This common API facilitates an easy migration from local calls to remote calls through Hot Rod: all one needs to do is switch between DefaultCacheManager and RemoteCacheManager - which is further simplified by the common CacheContainer interface that both inherit.

= 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());
```

= Unsupported Methods The Infinispan `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

= Remote Iterator API Infinispan 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
     }
}
```

= Deploying Custom Filters to Infinispan Server Deploy custom filters to Infinispan 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 ommitted, 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 Infinispan server installation directory.

Reference

- KeyValueFilterConverterFactory

= 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

= Streaming API Infinispan 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 Infinispan 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();
```

The Streaming API does **not** marshall 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();
```

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.iinfinispan.client.hotrod.StreamingRemoteCache

= 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

= 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
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());
```

== Removing Event Listeners

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

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

== 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 Infinispan 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);
```

Filter instances have to marshallable when they are deployed in a cluster so that the filtering can happen right where the event is generated, even if the even 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.

== 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");
```

== 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.

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 Infinispan 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});
```

Converter instances have to 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.

== 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.CacheEventFilterConverte`

rFactory 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 Infinispan 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);
```

== 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 Infinispan 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.

=== 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 ProtoStream marshallers 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 Infinispan server configuration.

See ProtoStream for more information.

== 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.

== 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 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.

= 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.

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.

== 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`.
- Locking mode must be `PESSIMISTIC`. In a future release, `OPTIMISTIC` locking mode will be supported.
- 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="hotrodReplTx">
  <locking isolation="REPEATABLE_READ"/>
  <transaction mode="NON_XA" locking="PESSIMISTIC"/>
</replicated-cache>
```

Hot Rod transactions have their own recovery mechanism.

== Configuring Hot Rod Clients When you create the RemoteCacheManager, you can set the default TransactionManager and TransactionMode that the RemoteCache uses.

The RemoteCacheManager lets you create only one configuration for transactional caches, as in the following example:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
//other client configuration parameters
cb.transaction().transactionManagerLookup(GenericTransactionManagerLookup.getInsta
nce());
cb.transaction().transactionMode(TransactionMode.NON_XA);
cb.transaction().timeout(1, TimeUnit.MINUTES)
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

The preceding configuration applies to all instances of a remote cache. If you need to apply

different configurations to remote cache instances, you can override the RemoteCache configuration. See Overriding RemoteCacheManager Configuration.

See ConfigurationBuilder 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.transaction.transaction_manager_lookup =
org.infinispan.client.hotrod.transaction.lookup.GenericTransactionManagerLookup
infinispan.client.hotrod.transaction.transaction_mode = NON_XA
infinispan.client.hotrod.transaction.timeout = 60000
```

=== 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.

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.

== Transaction Modes TransactionMode controls how a RemoteCache interacts with the TransactionManager.

Configure transaction modes on both the Infinispan 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 Infinispan configuration and client settings. Use the following modes with your client, see the Infinispan 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.

== Overriding Configuration for Cache Instances Because RemoteCacheManager does not support different configurations for each cache instance. However, RemoteCacheManager includes the `getCache(String)` method that returns the RemoteCache instances and lets you override some configuration parameters, as follows:

`getCache(String cacheName, TransactionMode transactionMode)`
: Returns a RemoteCache and overrides the configured TransactionMode.

`getCache(String cacheName, boolean forceReturnValue, TransactionMode transactionMode)`
: Same as previous, but can also force return values for write operations.

`getCache(String cacheName, TransactionManager transactionManager)`
: Returns a RemoteCache and overrides the configured TransactionManager.

`getCache(String cacheName, boolean forceReturnValue, TransactionManager transactionManager)`
: Same as previous, but can also force return values for write operations.

`getCache(String cacheName, TransactionMode transactionMode, TransactionManager transactionManager)`
: Returns a RemoteCache and overrides the configured TransactionManager and TransactionMode. Uses the configured values, if `transactionManager` or `transactionMode` is null.

`getCache(String cacheName, boolean forceReturnValue, TransactionMode transactionMode, TransactionManager transactionManager)`
: Same as previous, but can also force return values for write operations.

The `getCache(String)` method returns RemoteCache instances regardless of whether they are transaction or not. RemoteCache includes a `getTransactionManager()` method that returns the TransactionManager that the cache uses. If the RemoteCache is not transactional, the method

returns `null`.

== 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.

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.

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.

== 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.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstan
ce());
cb.transaction().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();
```

== Overriding the Transaction Manager

The following example shows how to override `TransactionManager` with the `getCache` method:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstan
ce());
cb.transaction().transactionMode(TransactionMode.NON_XA);

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

//Define a custom TransactionManager.
TransactionManager myCustomTM = ...

//Override the TransactionManager for the my-cache instance. Use the default
configuration if null is returned.
RemoteCache<String, String> cache = rcm.getCache("my-cache", null, myCustomTM);

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

== Overriding the Transaction Mode

The following example shows how to override `TransactionMode` with the `getCache` method:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstan
ce());
cb.transaction().transactionMode(TransactionMode.NON_XA);

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

//Override the transaction mode for the my-cache instance.
RemoteCache<String, String> cache = rcm.getCache("my-cache", TransactionMode
.NON_DURABLE_XA, null);

//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();
```