Pivotal

PRODUCT DOCUMENTATION

# VMware GemFire® C++ Client 10.1

Rev: 10.1.3

© Copyright 2020 VMware Inc. or its affiliates. All Rights Reserved.

# Table of Contents


| | |
|---|---|
| Table of Contents | 2 |
| VMware GemFire® Native Client 10.1 Documentation | 3 |
| GemFire Native Client 10.1 Release Notes | 4 |
| System Requirements | 6 |
| Upgrading a Native Client Application From Version 9 to Version 10 | 11 |
| Installing the Native Library | 19 |
| Getting Started with the Native Library | 20 |
| Put/Get/Remove Example | 23 |
| Configuring a Client Application | 25 |
| System Level Configuration | 27 |
| Configuring the Client Cache | 31 |
| Configuring Regions | 32 |
| Registering Interest for Entries | 36 |
| Region Attributes | 39 |
| Serializing Data | 41 |
| VMware GemFire® PDX Serialization | 42 |
| Using the PdxSerializable Abstract Class | 44 |
| PdxSerializable Example | 46 |
| Remote Queries | 50 |
| Continuous Queries | 53 |
| Security: Authentication and Encryption | 57 |
| Authentication | 58 |
| TLS/SSL Client-Server Communication Encryption | 60 |
| Set Up OpenSSL | 60 |
| Starting and stopping the client and server with SSL in place | 61 |
| Function Execution | 63 |
| Transactions | 69 |
| System Properties | 72 |
| Client Cache XML Reference | 77 |
| Cache Initialization File: XML Essentials | 77 |
| Cache Initialization File Element Descriptions | 78 |


© Copyright Pivotal Software Inc, 2013-2020

# VMware GemFire® Native Client 10.1 Documentation

Published July 10, 2020.

The VMware GemFire® Native Client is a library that provides access for C++ and Microsoft® .NET™ clients to a VMware GemFire® distributed system.

See the Release Notes for new features and support information regarding this release.

Upgrading a Native Client Application From Version 9 to Version 10 provides help with the upgrade from Native Client 9.x to Native Client 10.x.

See the API docs for API details:

- C++ API docs
- .NET API docs

The Apache Geode community has a host of examples based on the latest C++ and .NET APIs ( https://github.com/apache/geode-native/tree/develop/examples ).

See the *VMware GemFire® User Guide* for information regarding the server.

# GemFire Native Client 10.1 Release Notes

## What's New in GemFire Native Client 10.1

VMware GemFire® Native Client 10.1 is based on Apache Geode 1.11.

Version 10.1 includes a number of improvements:

- SSL enhancement - support for certificate chaining, better one-way SSL consistency
- Compatibility issues with various versions of PCC
- Performance improvements
- Bug fixes. See Issues Resolved in Native Client 10.1

The Apache Geode community has a host of examples based on the latest C++ and .NET APIs ( https://github.com/apache/geode-native/tree/develop/examples ).

## Issues Resolved in Native Client 10.1

This section describes issues resolved in VMware GemFire® Native Client version 10.1 and its patch releases, beginning with the most recent release.

### Issues Resolved in Native Client 10.1.3

**GEODE-8297, GEMNC-472:** Increased default timeout value for authorization to accommodate networks with higher latency.

### Issues Resolved in Native Client 10.1.2

**GEODE-7930**: Endpoint names are no longer truncated to 99 characters. The Native Client now supports endpoint names that meet the RFC 2181 standard of 255 characters for fully-qualified domain names. This solution also corrects a spurious "Failed to add endpoint" error that was issued when, in fact, no error had occurred.

### Issues Resolved in Native Client 10.1.1

**GEODE-8015, GEMNC-470**: Added debugging symbols to the released libraries. The Native Client release

for Windows now includes a .pdb symbol file. On Linux, the symbols are now embedded in the shared library (.so file).

## Issues Resolved in Native Client 10.1.0

**GEODE-3415:** Added support for certificate chain files in SSL configuration.

**GEODE-7437:** Enforced recognition of trust store in one-way SSL.

**GEODE-5708, GEMNC-465:** Fixed an issue with an overly-aggressive memory free-up operation in partitioned regions that caused the putAll() operation to fail when called a second time due to a closed server connection.

**GEODE-6576:** Improved handling of stale connections to partitioned regions.

**GEODE-6624, GEMNC-438:** Improved handling of data serialization error reporting by fixing a problem caused by nested exceptions.

**GEODE-6800, GEMNC-448:** Fixed a gcc compilation error related to CacheableFileName objects.

**GEODE-6835, GEMNC-442:** Added retry logic to prevent spurious server-side SecurityManager errors.

**GEODE-7019:** Fix closing of idle connections in native client.

**GEODE-7061:** Reduced the number of connections created during high load conditions with many threads.

**GEODE-7299:** Fixed a memory leak associated with PDX data serialization.

**GEODE-7316:** Fixed a race condition that could cause a client app to crash on shutdown.

**GEODE-7418, GEMNC-464:** Fixed an issue with PDX serialization/deserialization of JSON objects.

**GEODE-7476, GEODE-7509, GEMNC-436:** Fixed a memory leak that appeared during repeated queries.

**GEODE-7783:** Optimized connection handling to improve performance.

# System Requirements

**In this topic**

GemFire Compatibility

Application Compatibility

.NET Compatibility

Host Machine Requirements

Windows Support

Linux Support

PCF Support

Software Requirements for Using SSL

The VMware GemFire® native client provides access for C++ and Microsoft® .NET™ clients to the VMware GemFire® distributed system. It operates on platforms running Microsoft Windows, Linux (Intel), and Pivotal Cloud Foundry.

## GemFire Compatibility

The GemFire Native Client supports applications that communicate with GemFire servers. Native Client version 10.1 works with Pivotal GemFire versions 9.0.0 and later.

The following table shows which versions of the Native Client are compatible with the various versions of the GemFire server.

| GemFire Native Client Version | GemFire Server Version |
| --- | --- |
| GemFire Native Client 10 | GemFire Server 10 |
| GemFire Native Client 10<br>GemFire Native Client 9.1, 9.2 | GemFire Server 9.x |
| GemFire Native Client 9.1<br>GemFire Native Client 8.2 | GemFire Server 8.2 |

## Application Compatibility

GemFire Native Client is compiled using 64-bit architectures for all operating systems. Linking with 32-bit

© Copyright Pivotal Software Inc, 2013-2020

applications is not supported.

## Supported Platforms: C++ Client

The GemFire Native Client supports applications that run in the following client environments:

| Platform | Version |
| --- | --- |
| Linux | Red Hat Enterprise Linux (RHEL) 7 |
| Linux | Ubuntu 16 (Xenial) |
| Windows Desktop | 10 |
| Windows Server | 2016 |
| Pivotal Cloud Foundry | PCF 2.3.2+ |

## Supported Platforms: .NET Client

| Platform | Version |
| --- | --- |
| Windows Desktop | 10 |
| Windows Server | 2016 |
| Pivotal Cloud Foundry | PCF 2.3.2+ |

## .NET Compatibility

For Windows applications, a Microsoft .NET Framework must be installed to support the C++/CLI (Common Language Infrastructure) library for the native client.

The client supports .NET 4.5.2 (and newer) and Visual Studio 2017 (and newer) for compiling .NET applications on Windows. It does not support .NET Core. For more information on the features of .NET and Visual Studio Community Edition 2017, see the Visual Studio 2017 web page.

## Host Machine Requirements

Each machine that runs a native client must meet the following requirements:

- A system clock set to the correct time and a time synchronization service such as Network Time Protocol (NTP). Correct time stamps permit the following activities:

- Logs that are useful for troubleshooting. Synchronized time stamps ensure that log messages from different hosts can be merged to reproduce an accurate chronological history of a distributed run.
  - Aggregate product-level and application-level time statistics.
  - Accurate monitoring of the system with scripts and other tools that read the system statistics and log files.
- The host name and host files are properly configured for the machine.

## Windows Support

For Windows C++ applications, the GemFire Native Client library, `pivotal-gemfire.dll`, requires the **Microsoft Visual C++ 2017 Redistributable Package**, which you can find on the Visual Studio 2017 web page. Scroll down to “Redistributables and Build Tools” and select “Microsoft Visual C++ Redistributable for Visual Studio 2017”, and be sure to select the “x64” version. Install it on all machines that will run your C++ application.

## Linux Support

For Linux, you can verify that you meet the native client dependencies at the library level by using the `ldd` tool and entering this command:

```
$ ldd $client-installdir/lib/libpivotal-gemfire.so
```

where *client-installdir* is the location in which you have installed the client.

The following libraries are external dependencies of the native library, `libpivotal-gemfire.so`. Verify that the `ldd` tool output includes all of these:

- libdl.so.2
- libm.so.6
- libpthread.so.0
- libc.so.6
- libz.so.1

### Disabling Syn Cookies on Linux

Many default Linux installations use SYN cookies to protect the system against malicious attacks that

flood TCP SYN packets. The use of SYN cookies dramatically reduces network bandwidth, and can be triggered by a running VMware GemFire® distributed system.

To disable SYN cookies permanently:

1. Edit the `/etc/sysctl.conf` file to include the following line:

   ```
   net.ipv4.tcp_syncookies = 0
   ```

   Setting this value to zero disables SYN cookies.

2. Reload `sysctl.conf`:

   ```
   $ sysctl -p
   ```

# PCF Support

Pivotal Cloud Foundry supports .NET and C++ native client applications.

PCF versions 2.3.2 and higher include the Microsoft VS 2017 C++ Redistributable DLLs.

## PCF .NET Requirements

- PCF 2.3.2 or newer
- Windows Server 2016
- .NET 4.5.2 or newer

To run your cloud native .NET application on PCF:

1. The `Pivotal.GemFire.dll` must be in the `output` folder of your .NET project.
2. Rebuild your application.
3. From Visual Studio, publish your application to a filesystem.
4. From within the published filesystem, use `cf push` to deploy your application to PCF as you would other .NET applications.

## PCF C++ Requirements

- PCF 2.3.2 or newer
- Ubuntu or Windows 2017 stem cells

To run your cloud native C++ application on PCF:

1. The runtime libraries `pivotal-gemfire.dll` and `cryptoImpl.dll` must be in the path of your C++ application.
2. Use `cf push` to deploy your application to PCF as you would other C++ applications.

## Software Requirements for Using SSL

If you plan on using SSL in your VMware GemFire® native client and server deployment, you will need to download and install OpenSSL. The VMware GemFire® native client requires OpenSSL version 1.1.1.

For Windows platforms, you can use either the regular or the OpenSSL "Light" version.

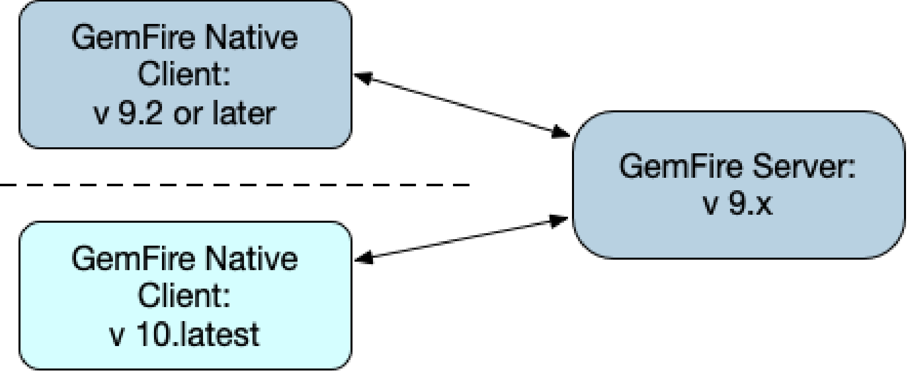
In addition, make sure that your system environment variables have been configured to include OpenSSL.

# Upgrading a Native Client Application From Version 9 to Version 10

GemFire Native Client Version 10 introduces breaking changes for Version 9 applications. Updating your client applications will require more intervention than merely recompiling.

In general, you will have best performance and reliability if clients and servers both run the latest versions of their respective software.

GemFire server and client software releases follow similar numbering schemes, but they are not released in lockstep. The following diagram illustrates the interoperability between recent versions of GemFire server software and GemFire Native Client software.




## Overview of Changes

VMware GemFire® Native Client improvements and new features include:

- A modernized C++ API that constitutes a big step forward to fully supporting C++ 11.
- Local memory management has been greatly improved, as well as the adoption of a new cache model that allows for multiple cache objects to exist in a given process space.
- The .NET interface benefits from all the enhancements made in the C++ interface.
- The Native Client now supports IIS application domains and Pivotal Cloud Foundry.
- A new architecture that allows for more flexible client-side data models
- Improvements to the reflection-based AutoSerializer

The Apache Geode community has a host of examples based on the latest C++ and .NET APIs ( https://github.com/apache/geode-native/tree/develop/examples ).

For examples of source changes see the Native Client 9 to Native Client 10 Upgrade Sample .

© Copyright Pivotal Software Inc, 2013-2020

These examples show both the original and new API usage, and may be helpful as starting points for upgrading your application to Native Client 10. To see details for upgrading your particular API usage refer to the Native Client 10 API documentation:

- C++ API docs
- .NET API docs

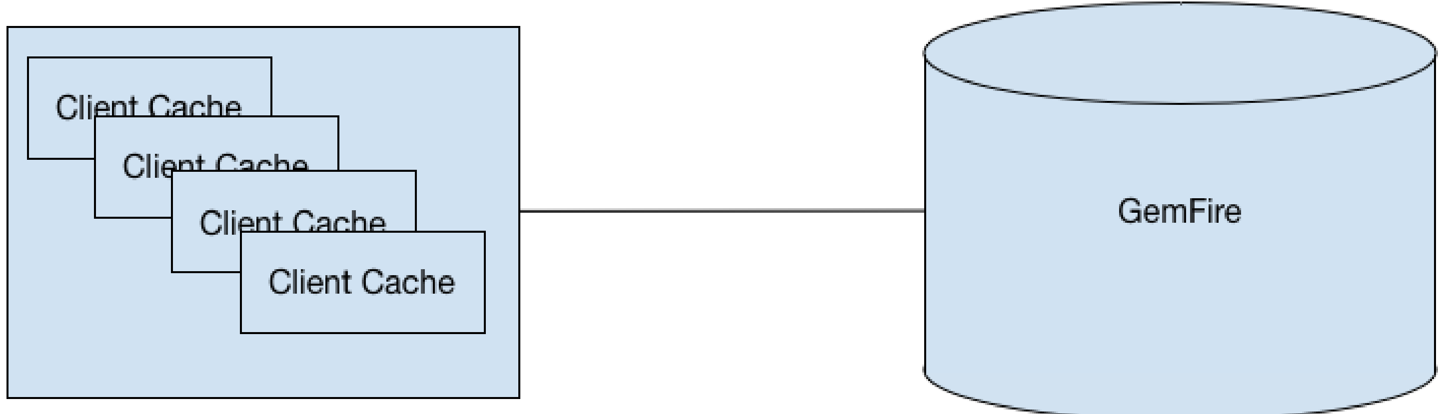
## Compiler Upgrade

Using Version 10 of the Native Client with your application requires a C++11-compatible compiler.

## Removal of Cache Singleton

A major change in Native Client 10 is the change from a singleton-based design to an instance-based design. This gives developers greater programming flexibility, as client cache instances can have completely independent access to the GemFire data grid.

Multiple client-side instances of Cache do not automatically share objects with one another.




The cache creation process in Native Client 10 follows a different pattern and now returns an object (see examples). Each also requires a pool. Native Client 10 further simplifies the cache creation and system architecture with the removal of `DistributedSystem`. An instance of `DistributedSystem` is no longer needed to manage an application's "connecting" into the VMware GemFire® Java server. This is now managed through a Cache instance.

A note to .NET users of the Native Client: users can pass in an instance of their authorization class into the CacheFactory ( `CacheFactory.SetAuthInitialize(app_auth);` ).

# Serialization Interface Changes

The Native Client serialization APIs for both C++ and .NET have been changed to more closely resemble the behavior of the GemFire Java client. The serializable API has been refactored into data serializable and PDX serializable interfaces. In addition, to be consistent with the Java Server, the new data serializable interface does not have fixed `ClassId` properties. `ClassId` is now a parameter passed in to register a given type.

# C++ Standardization

In Native Client 10, many version 9 utility classes, such as shared pointers, have been replaced by their equivalents from the C++11 standard library.

One of the biggest changes made in Native Client 10 is the replacement of custom `apache::geode::client::SharedPtr` with `std::shared_ptr`. The custom base object `apache::geode::client::SharedBase` has been removed and is no longer a required derivation to make library objects reference counted; instead objects may simply be wrapped by a `std::shared_ptr<>`. Upgrading to Native Client 10 requires replacing all *Ptr types with their C++11 replacements.

For example, replace

```
RegionPtr regionPtr;
```

with

```
std::shared_ptr<Region> regionPtr;
```

Other adopted C++11 standards include:

- All time values now use std::chrono. For example, `std::chrono` replaces `CacheableDate`
- Longs and ints are now replaced with language primitives of explicit size, such as `int32_t` and `int16_t`.
- `std:string` replaces `char *`
- std container classes
- `PDXSerializable::objectSize()` and `DataSerializable::objectSize()` return `size_t`
- Properties use `std::unordered` map

# Enum Classes

The following Version 9 Enums are now defined as C++11 Enum classes in the Version 10 client:

- `CqOperation`
- `CqState`
- `ExpirationAction`
- `PdxFieldTypes`

# Exceptions

GemFire Native Client Exceptions, which were implemented as macros in v9, are now classes that inherit from `std::exception`.

# Object Oriented Design Patterns

Native Client 10 has adopted many more object oriented design patterns. For example, the `CacheFactory` now uses a builder pattern and returns a value rather than a pointer.

Other examples of pattern-oriented changes:

- Replace `apache::geode::client::PoolPtr` with `std::shared_ptr<apache::geode::client::Pool>`
- Replace `apache::geode::client::RegionPtr` with `std::shared_ptr<apache::geode::client::Region>`
- Replace `apache::geode::client::EntryEventPtr` with `std::shared_ptr<apache::geode::client::EntryEvent>`
- Replace `apache::geode::client::CachePtr` with `std::unique_ptr<apache::geode::client::Cache>`
- PdxSerializable toData/fromData are now passed to PdxWriter/PdxReader as references
- Execution factory returns value type
- `Cache::createPdxInstanceFactory` returns object
- `CqQuery::getCqAttributesMutator` returns value
- `Cache::createDataInput/Output` returns value

# Initialization Files

The best practice for most applications is to set properties and parameters programmatically. For clients that use the older, file-based scheme, the following changes apply to the system initialization files, `geode.properties` and `cache.xml`:

- In both files, parameters specifying times should include units (`s`, `m`, `h`, etc.).
- For the `cache.xml` file, the schema name space and location have changed. Use

```
<client-cache
  xmlns="http://geode.apache.org/schema/cpp-cache"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://geode.apache.org/schema/cpp-cache
    http://geode.apache.org/schema/cpp-cache/cpp-cache-1.0.xsd"
  version="1.0">
```

# Other Changes

- The `$GFCPP` environment variable is no longer needed
- `enable-chunk-handler-thread` now defaults to `false` and replaces `disable-chunk-handler-thread`
- Native Client 10 now supports OpenSSL
- Statistics and StatisticsFactory are no longer available

# .NET API Changes

These .NET API classes have changed as follows:

**CacheFactory**

- Creation was via static method `CreateCacheFactory`, now created via `new`
- Authorization implementation now is a setter on factory called `SetAuthInitialize`
- `Appdomain` property is no longer a supported property
- `GetAnyInstance()` is no longer supported (there is no more global singleton). Make method calls on the specific instance you are working with

**PoolFactory**

© Copyright Pivotal Software Inc, 2013-2020

- Creation was obtained via static method `PoolManager.CreateFactory`, now via `GetPoolFactory` method on `Cache`
- `SetEntryTimeToLive` - was `int`, now uses `TimeSpan`
- SetEntryIdleTimeout - was `int`, now uses `TimeSpan`
- SetRegionTimeToLive - was `int`, now uses `TimeSpan`
- SetRegionIdleTimeout - was `int`, now uses `TimeSpan`

#### RegionFactory

- `SetEntryTimeToLive` - was `int`, now uses `TimeSpan`.

#### IGeodeSerializable

- The `IGeodeSerializable` interface has been renamed to `IDataSerializable`.

## .NET Session State Provider

The Native Client 10 version of the Session State Provider (SSP) only requires configuration to be set in `Web.Config` and the deployment of server-side functions.

## C++ API Changes

The following classes have changed or are no longer present in the current release.

| Version 9: Removed Class | Version 10: Recommended Action |
|---|---|
| Assert | N/A |
| AttributesFactory | Replace with RegionAttributesFactory |
| CacheableArrayType, CacheableContainerType, CacheableKeyType | Converted to templates. See INSTALL_DIR/include/geode/CacheableBuiltins.hpp |
| DistributedSystem | Used internally |
| EqualToSB | No longer needed; use std types |
| GeodeTypeIds | Removed from public API |
| HashMapOfCacheable | Replace with std::hash |
| HashMapOfSharedBase | Replace with std::hash |

© Copyright Pivotal Software Inc, 2013-2020

| Version 9: Removed Class | Version 10: Recommended Action |
|---|---|
| HashSB | Replace with std::hash |
| HashSetOfCacheableKey | Replace with std::hash |
| HashSetOfSharedBase | Replace with std::shared_ptr<T> |
| HashSetT | Replace with std::hash |
| InternalCacheTransactionManager2PC | Removed from public API |
| Log | Use LogLevel at cache creation |
| LogFn | Use LogLevel at cache creation |
| LogVarargs | Use LogLevel at cache creation |
| NullSharedBase | Replace with nullptr |
| SelectResultsIterator | Replace with ResultsCollector |
| SharedArrayPtr | Replace with std::shared_ptr<T> |
| SharedBase | Abstract base class no longer needed. Replace with std::shared_ptr<T> |
| SharedPtr | Replace with std::shared_ptr<T> |
| SPEHelper | Exception helper no longer needed after move from SharedPtr to std::shared_ptr |
| VectorOfCacheable | std::vector<std::shared_ptr<T>> |
| VectorOfCacheableKey | std::vector<std::shared_ptr<T>> |
| VectorOfSharedBase | std::vector<std::shared_ptr<T>> |
| VectorT | std::vector<T> |

The following classes have changed or new in the current release.

| New or Renamed Class | Summary of Changes |
|---|---|
| AuthenticatedView | Replaces AuthenticatedCache in v9 API. Used for multi-user authentication. |
| DataSerializable | An interface for objects whose state can be written/read as primitive types. Supersedes Serializable, which is now the superclass of all user objects in the cache that can be serialized. |
| DefaultResultCollector | Default class that gathers results from function execution. The ResultCollector interface also changed. |
| LogLevel | Method returns log level. |
| RegionAttributesFactory | Replaces AttributesFactory |

| New or Renamed Class | Summary of Changes |
| --- | --- |
| RegionShortcut | Enum class holding all region types (PROXY, CACHING_PROXY, CACHING_PROXY_ENTRY_LRU, LOCAL_ENTRY_LRU) |
| TypeRegistry | Class for registering a custom serializable type. |

# Installing the Native Library

Install the native client by extracting the contents of the distribution archive and setting up the environment.

## Installation Prerequisites

Before installing the GemFire native client, confirm that your system meets the hardware and software requirements described in GemFire Native Client System Requirements.

## Copy and Uncompress the Distribution Archive

1. In a browser, navigate to the Pivotal GemFire download page.
2. From the **Releases:** pull-down menu, select the most recent version of VMware GemFire® Native Client.
3. Expand the entry in the **Release Download Files** dialog box, select the version that best suits your development platform, and download it.
4. Move the downloaded archive to the local directory or folder in which you wish to install the Native Client libraries. For ease of use, choose a well-known location:
   - On Linux, /usr/local
   - On Windows, C:\Program Files
5. Uncompress the distribution archive, which may be a ZIP archive or a compressed tar file (.tar.gz or .tgz). For example:

   ```
   $ unzip pivotal-gemfire-nativeclient-windows-64bit-10.x.y.zip
   ```

   or

   ```
   $ tar xvzf pivotal-gemfire-nativeclient-linux-64bit-10.x.y.tar.gz
   ```

6. For ease of use, rename the resulting directory to `nativeclient`.

# Getting Started with the Native Library

**In this topic**


Set Up Your Development Environment

Establish Access to a VMware GemFire® Cluster

  Connecting to the Server

  Application Development Walkthrough

Programming Examples


To use the VMware GemFire® Native Library for developing VMware GemFire® client applications:

- Obtain a distribution of the Native library and install it on your development platform.
- Set up your development environment with the tools you need, such as a compiler and an OpenSSL security library.
- Establish access to a new or existing VMware GemFire® cluster.
- Write your client application using the VMware GemFire® native library to interact with the VMware GemFire® server.

## Set Up Your Development Environment

You will need some essential tools, such as a compiler and a linker. Your compiler must have access to the Native Client header files, and the linker must have access to the Native Client libraries. The header files and libraries are located in the Native Client installation directory.

## Establish Access to a VMware GemFire® Cluster

As you develop your application, you will need access to a VMware GemFire® cluster. Your client application connects to a VMware GemFire® cluster by specifying the address (host name or IP address) and port number of one or more locators, and the name of a region that also exists on the cluster. The client API establishes a pool of these network connections for your client application to use.

You can choose whether to use a large, remote, production-quality cluster; a small, local, development cluster; or something in-between, such as a testing or experimental lab installation.

In the *VMware GemFire® User's Guide*, see Configuring and Running a Cluster and Client/Server Configuration for instructions on setting up and starting the cluster for a client/server configuration.

### Connecting to the Server

To connect to a server, your application must follow these steps:

1. Instantiate a `CacheFactory`, setting characteristics of interest (for example, `log-level`).

© Copyright Pivotal Software Inc, 2013-2020

2. Create a cache and use it to instantiate a `PoolFactory`, specifying the hostname and port for the server locator.

3. Create a named pool of network connections.

4. Instantiate a region of the desired type (usually CACHING_PROXY or PROXY) and connect it by name to its counterpart on the server.

Once the connection pool and the shared region are in place, your client application is ready to share data with the server.

**Server Connection: C++ Example**

This example of connecting to the server is taken from the C++ `put-get-remove` example.

Instantiate a `CacheFactory` and set its characteristics:

```
auto cacheFactory = CacheFactory();          // instantiate cache factory
cacheFactory.set("log-level", "none");        // set cache log-level characteristics
```

Create a cache and use it to instantiate a `PoolFactory`:

```
auto cache = cacheFactory.create();                       // create cache
auto poolFactory = cache.getPoolManager().createFactory();  // instantiate pool factory

poolFactory.addLocator("localhost", 10334);                // add locator to pool factory
```

Create a named pool of network connections, and instantiate a region of the desired type:

```
auto pool = poolFactory.create("pool");                  // create a pool called "pool" that knows where the server is
auto regionFactory = cache.createRegionFactory(RegionShortcut::PROXY); // instantiate region factory with PROXY characteristics
auto region = regionFactory.setPoolName("pool").create("example_userinfo"); // create a connection to the region "example_userinfo" on the server
```

See the *VMware GemFire® User Guide* section Configuring a Client/Server System for more details.

## Application Development Walkthrough

The C++ App Development Walkthrough describes how to set up a native client development environment using CMake.

# Programming Examples

The VMware GemFire® Client build provides a set of programming examples to help you understand the client API. The `examples` directory contains CMake files and a `cpp` subdirectory containing C++ examples. The Windows build also includes a `dotnet` subdirectory containing C# examples.

CMake files are located at each level of the directory structure to allow examples to be built individually or in groups.

The directory structure resembles this hierarchy (some entries are omitted for clarity):

```
MyProject/
  cmake/
  CMakeLists.txt
  examples/
    BUILD-EXAMPLES.md
    CMakeLists.txt
    CMakeLists.txt.in
    cmake/
    cpp/
      authinitialize/
      continuousquery/
      dataserializable/
      functionexecution/
      pdxserializable/
      pdxserializer/
      putgetremove/
      remotequery/
      sslputget/
      transaction/
    dotnet/
      authinitialize/
      continuousquery/
      dataserializable/
      functionexecution/
      pdxautoserializer/
      pdxserializable/
      putgetremove/
      remotequery/
      sslputget/
      transaction/
```

See the `BUILD-EXAMPLES.md` file for detailed instructions on building and executing the examples, and read the source code to understand how the examples are constructed.

See Put/Get/Remove Example for sample code showing the basics of how a client application connects to a VMware GemFire® cluster and performs basic operations on a remote server.

# Put/Get/Remove Example

**In this topic**

Put/Get/Remove Example Code

The native client release contains an example written for C++ showing how a client application can establish a connection to a cluster and then use that connection to perform basic operations on a remote server. The examples are located in `examples/cpp/putgetremove`.

The example performs a sequence of operations, displaying simple log entries as they run.

- To run the example, follow the instructions in the `README.md` file in the example directory.
- Review the source code in the example directory to see exactly how it operates.
- Begin by running a script that sets up the server-side environment by invoking `gfsh` commands to create a region, simply called “example_userinfo.”
- Run the example client application, which performs the following steps:
  - Connects to the server
  - Performs region put operations using key/value pairs
  - Uses region get to retrieve the values
  - Uses region remove to remove the values

## Put/Get/Remove Example Code

This section contains code snippets showing highlights of the C++ put/get/remove example. They are not intended for cut-and-paste execution. For the complete source, see the example source directory.

The C++ example creates a cache, then uses it to create a connection pool and a region object (of class `Region`).

```
auto cacheFactory = CacheFactory();
cacheFactory.set("log-level", "none");
auto cache = cacheFactory.create();
auto poolFactory = cache.getPoolManager().createFactory();

poolFactory.addLocator("localhost", 10334);
auto pool = poolFactory.create("pool");
auto regionFactory = cache.createRegionFactory(RegionShortcut::PROXY);
auto region = regionFactory.setPoolName("pool").create("example_userinfo");
```

The client then populates the data store with two key/value pairs.

```
region->put("rtimmons", "Robert Timmons");
region->put("scharles", "Sylvia Charles");
```

Next, the application retrieves the stored values using `Get` operations.

```
auto user1 = region->get("rtimmons");
auto user2 = region->get("scharles");
```

Finally, the application deletes one of the stored values using the `Remove` method.

```
if (region->existsValue("rtimmons")) {
  std::cout << "rtimmons's info not deleted" << std::endl;
} else {
  std::cout << "rtimmons's info successfully deleted" << std::endl;
}
```

# Configuring a Client Application

**In this topic**

Programmatic Configuration vs XML Configuration

High Availability with Server Redundancy

You can configure your native client application:

- Programmatically in your app code
- Via XML files and properties files (see Client Cache XML Reference)
- Through a combination of programmatic and file-based approaches

This section describes configuration on two levels, the system level and the cache level. System property settings describe your application's behavior, while cache configuration describes data.

## Programmatic Configuration vs XML Configuration

Programmatic configuration enables your client application to dynamically adapt to changing runtime conditions.

In contrast, XML configuration externalizes properties, such as locator addresses and pool connection details, so they can be changed without requiring that you recompile your application.

### C++ RegionFactory Example

The following examples illustrate how to set a region's expiration timeout attribute programmatically and through XML.

Setting a property programmatically:

```
auto regionFactory = cache.createRegionFactory(RegionShortcut::CACHING_PROXY);
auto region = regionFactory.setRegionTimeToLive(ExpirationAction::INVALIDATE,
              std::chrono::seconds(120))
    .create("exampleRegion0");
```

XML equivalent:

```
<region name="exampleRegion0" refid="CACHING_PROXY">
  <region-attributes pool-name="default">
    <region-time-to-live>
      <expiration-attributes timeout="120s" action="invalidate"/>
    </region-time-to-live>
  </region-attributes>
</region>
```

## Tables of properties

See System Properties for a list of system properties that can be configured programmatically or in the `geode.properties` file.

## High Availability with Server Redundancy

When redundancy is enabled, secondary servers maintain queue backups while the primary server pushes events to the client. If the primary server fails, one of the secondary servers steps in as primary to provide uninterrupted event messaging to the client. To configure high availability, set the `subscription-redundancy` in the client's pool configuration. This setting indicates the number of secondary servers to use. See the *VMware GemFire® User Guide* section Configuring Highly Available Servers for more details.

# System Level Configuration

**In this topic**


Attribute Definition Priority

Search Path for Multiple Properties Files

Defining Properties Programmatically

About the geode.properties Configuration File

Configuration File Locations

Using the Default Sample File

Configuring System Properties for the Client

Running a Client Out of the Box


## Attribute Definition Priority

You can specify attributes in different ways, which can cause conflicting definitions. Applications can be configured programmatically, and that has priority over other settings.

In case an attribute is defined in more than one place, the first source in this list is used:

- Programmatic configuration
- Properties set at the command line
- `current-working-directory/geode.properties` file
- `native-client-installation-directory/defaultSystem/geode.properties` file
- defaults

The `geode.properties` files and programmatic configuration are optional. If they are not present, no warnings or errors occur. For details on programmatic configuration through the `Properties` object, see Defining Properties Programmatically.

## Search Path for Multiple Properties Files

The client and cache server processes first look for their properties file in the `native-client-installation-directory/defaultSystem` directory, then in the working directory.

Any properties set in the working directory override settings in the `native-client-installation-directory/defaultSystem/geode.properties` file.

The `geode.properties` file provides information to the client regarding the expected server configuration. Properties set in this file (in the client environment) do not have any effect on the server itself. Its main purpose is to inform the client application as to how to communicate with the server.

## Defining Properties Programmatically

You can pass in specific properties programmatically by using a `Properties` object to define the non-default properties.

Example:

```
auto systemProps = Properties::create();
systemProps->insert("statistic-archive-file", "stats.gfs");
systemProps->insert("cache-xml-file", "./myapp-cache.xml");
systemProps->insert("stacktrace-enabled", "true");
auto cache = CacheFactory(systemProps).create();
```

## About the geode.properties Configuration File

The `geode.properties` file provides local settings required to connect a client to a distributed system, along with settings for licensing, logging, and statistics. See System Properties.

## Configuration File Locations

A client looks for a `geode.properties` file first in the working directory where the process runs, then in `native-client-installation-directory/defaultSystem`. Use the `defaultSystem` directory to group configuration files or to share them among processes for more convenient administration. If `geode.properties` is not found, the process starts up with the default settings.

For the `cache.xml` cache configuration file, a client looks for the path specified by the `cache-xml-file` attribute in `geode.properties` (see System Properties). If the `cache.xml` is not found, the process starts with an unconfigured cache.

## Using the Default Sample File

A sample `geode.properties` file is included with the VMware GemFire® native client installation in the `*native-client-installation-directory*/defaultSystem` directory.

To use this file:

1. Copy the file to the directory where you start the application.
2. Uncomment the lines you need and edit the settings as shown in this example:

   ```
   cache-xml-file=test.xml
   ```

3. Start the application.

**Default geode.properties File**

```
# Default C++ distributed system properties
# Copy to current directory and uncomment to override defaults.
#
## Debugging support, enables stacktraces in  apache::geode::client::Exception.
#
# The default is false, uncomment to enable stacktraces in exceptions.
#stacktrace-enabled=true
#crash-dump-enabled=true
#
#
## Cache region configuration
#
#cache-xml-file=cache.xml
#
## Log file config
#
#log-file=gemfire_cpp.log
#log-level=config
# zero indicates use no limit.
#log-file-size-limit=0
# zero indicates use no limit.
#log-disk-space-limit=0
...
```

# Configuring System Properties for the Client

The typical configuration procedure for a client includes the high-level steps listed below.

1. Place the `geode.properties` file for the application in the working directory or in

`native-client-installation-directory/defaultSystem`.

2. Place the `cache.xml` file for the application in the desired location and specify its path using the `cache-xml-file` property in the `geode.properties` file.

3. Add other attributes to the `geode.properties` file as needed for the local system architecture.

## Running a Client Out of the Box

If you start a client without any configuration, it uses any attributes set programmatically plus any hard-coded defaults (listed in System Properties). Running with the defaults is a convenient way to learn the operation of the distributed system and to test which attributes need to be reconfigured for your environment.

Running based on defaults is not recommended for production systems, as important components, such as security, might be overlooked.

# Configuring the Client Cache

Client caches provide the framework for clients to store, manage, and distribute application data.

A cache is an entry point for access to VMware GemFire®. Through the cache, clients gain access to the VMware GemFire® caching framework for data loading, distribution, and maintenance.

A `Cache` instance allows your client to set general parameters for communication between a cache and other caches in the distributed system, and to create and access any region in the cache.

Regions are created from `Cache` instances. Regions provide the entry points to the interfaces for instances of `Region` and `RegionEntry`.

For more information specific to your client programming language, see the C++ Client API.

# Configuring Regions

**In this topic**

Programmatic Region Creation

Declarative Region Creation

Invalidating and Destroying Regions

Region Access

Getting the Region Size

The region is the core building block of the VMware GemFire® distributed system. All cached data is organized into data regions and you do all of your data puts, gets, and querying activities against them.

In order to connect to a VMware GemFire® server, a client application must define a region that corresponds to a region on the server, at least in name. See Data Regions in the *VMware GemFire® User Guide* for details regarding server regions, and Region Attributes in this guide for client region configuration parameters.

You can create regions either programmatically or through declarative statements in a `cache.xml` file. Programmatic configuration is recommended, as it keeps the configuration close at hand and eliminates an external dependency. Region creation is subject to attribute consistency checks.

## Programmatic Region Creation

To create a region:

1. Instantiate a `CacheFactory` and use it to create a cache.
2. The cache includes an instance of `PoolManager` —use it to create a connection pool.
3. Use cache to instantiate a `RegionFactory` and use it to create a region, specifying any desired attributes and an association with the connection pool.

### C++ Region Creation Example

The following example illustrates how to create two regions using C++.

```
auto cache = CacheFactory().create();

auto examplePool = cache.getPoolManager()
    .createFactory()
    .addLocator("localhost", 40404)
    .setSubscriptionEnabled(true)
    .create("examplePool");

auto clientRegion1 = cache.createRegionFactory(RegionShortcut::PROXY)
  .setPoolName("examplePool")
  .create("clientRegion1");
```

## Declarative Region Creation

Declarative region creation involves placing the region's XML declaration, with the appropriate attribute settings, in a `cache.xml` file that is loaded at cache creation.

Like the programmatic examples above, the following example creates two regions with attributes and a connection pool:

```
<?xml version="1.0" encoding="UTF-8"?>
<client-cache
    xmlns="http://geode.apache.org/schema/cpp-cache"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://geode.apache.org/schema/cpp-cache
      http://geode.apache.org/schema/cpp-cache/cpp-cache-1.0.xsd"
    version="1.0">
    <pool name="examplePool" subscription-enabled="true">
        <server host="localhost" port="40404" />
    </pool>
    <region name="clientRegion1" refid="PROXY">
        <region-attributes pool-name="examplePool"/>
    </region>
    <region name="clientRegion2" refid="CACHING_PROXY">
        <region-attributes pool-name="examplePool">
            <region-time-to-live>
                <expiration-attributes timeout="120s" action="invalidate"/>
            </region-time-to-live>
        </region-attributes>
    </region>
</client-cache>
```

The `cache.xml` file contents must conform to the XML described in the `cpp-cache-1.0.xsd` file provided in your distribution's `xsds` subdirectory and available online at https://geode.apache.org/schema/cpp-

cache/cpp-cache-1.0.xsd.

## Invalidating and Destroying Regions

Invalidation marks all entries contained in the region as invalid (with null values). Destruction removes the region and all of its contents from the cache.

You can execute these operations explicitly in the local cache in the following ways:

- Through direct API calls from the client using `apache::geode::client::Region:invalidateRegion()`
- Through expiration activities based on the region's statistics and attribute settings.

In either case, you can perform invalidation and destruction as a local or a distributed operation.

- A local operation affects the region only in the local cache.
- A distributed operation works first on the region in the local cache and then distributes the operation to all other caches where the region is defined. This is the proper choice when the region is no longer needed, or valid, for any application in the distributed system.
- If the region on the server is configured as a partitioned region, it cannot be cleared using API calls from the client.

A user-defined cache writer can abort a region destroy operation. Cache writers are synchronous listeners with the ability to abort operations. If a cache writer is defined for the region anywhere in the distributed system, it is invoked before the region is explicitly destroyed.

Whether carried out explicitly or through expiration activities, invalidation and destruction cause event notification.

## Region Access

You can use `Cache::getRegion` to retrieve a reference to a specified region.

`Cache::getRegion` returns `nullptr` if the region is not already present in the application's cache. A server region must already exist.

A region name *cannot* contain these characters:

| Ineligible Character description | Ineligible Character |
| --- | --- |

| Ineligible Character description | Ineligible Character |
| --- | --- |
| whitespace | *space or tab* |
| angle brackets | < > |
| colon | : |
| quote | `"` |
| forward slash and back slash | / \ |
| pipe (vertical bar) | \| |
| question mark | ? |
| asterisk | * |

## Getting the Region Size

The `Region` API provides a `size` method that gets the size of a region. For client regions, this gives the number of entries in the local cache, not on the servers. See the `Region` API documentation for details.

# Registering Interest for Entries

For client regions, you can programmatically register interest in entry keys stored on a cache server region. A client region receives update notifications from the cache server for the keys of interest.

You can register interest for specific entry keys or for all keys. Regular expressions can be used to register interest for keys whose strings match the expression. You can also unregister interest for specific keys, groups of keys based on regular expressions, or for all keys.

**Note:** Interest registration and unregistration are symmetrical operations. Consequently, you cannot register interest in all keys and then unregister interest in a specific set of keys. Also, if you first register interest in specific keys with `registerKeys`, then call `registerAllKeys`, you must call `unregisterAllKeys` before specifying interest in specific keys again.

## Client API for Registering Interest

You register client interest through the C++ API. The C++ API provides the `registerKeys`, `registerAllKeys`, and `registerRegex` methods, with corresponding unregistration accomplished using the `unregisterKeys`, `unregisterAllKeys`, and `unregisterRegex` methods.

The `registerKeys`, `registerRegex` and `registerAllKeys` methods have the option to populate the cache with the registration results from the server. The `registerRegex` and `registerAllKeys` methods can also optionally return the current list of keys registered on the server.

## Setting Up Client Notification

In addition to the programmatic function calls, to register interest for a server region and receive updated entries you need to configure the region with the `PROXY` or `CACHING_PROXY RegionShortcut` setting. The region's pool should have `subscription-enabled=true` set either in the client XML or programmatically via a `CacheFactory::setSubscriptionEnabled(true)` API call. Otherwise, when you register interest, you will get an `UnsupportedOperationException`.

```
<region name="listenerWriterLoader" refid="CACHING_PROXY">
  ...
```

All clients that have subscriptions enabled track and drop (ignore) any duplicate notifications received. To reduce resource usage, a client expires tracked sources for which new notifications have not been received for a configurable amount of time.

© Copyright Pivotal Software Inc, 2013-2020

### Notification Sequence

Notifications invoke `CacheListeners` of cacheless clients in all cases for keys that have been registered on the server. Similarly, invalidates received from the server invoke `CacheListeners` of cacheless clients.

If you register to receive notifications, listener callbacks are invoked irrespective of whether the key is in the client cache when a `destroy` or `invalidate` event is received.

## Registering Interest for Specific Keys

You register and unregister interest for specific keys through the `registerKeys` and `unregisterKeys` functions. You register interest in a key or set of keys by specifying the key name using the programmatic syntax shown in the following example:

```
keys0.push_back(keyPtr1);
keys1.push_back(keyPtr3);
regPtr0->registerKeys(keys0);
regPtr1->registerKeys(keys1);
```

The programmatic code snippet in the next example shows how to unregister interest in specific keys:

```
regPtr0->unregisterKeys(keys0);
regPtr1->unregisterKeys(keys1);
```

## Registering Interest for All Keys

If the client registers interest in all keys, the server provides notifications for all updates to all keys in the region. The next example shows how to register interest in all keys:

```
regPtr0->registerAllKeys();
regPtr1->registerAllKeys();
```

The following example shows a code sample for unregistering interest in all keys.

```
regPtr0->unregisterAllKeys();
regPtr1->unregisterAllKeys();
```

© Copyright Pivotal Software Inc, 2013-2020

# Registering Interest Using Regular Expressions

The `registerRegex` function registers interest in a regular expression pattern. The server automatically sends the client changes for entries whose keys match the specified pattern.

*Keys must be strings in order to register interest using regular expressions.*

The following example shows interest registration for all keys whose first four characters are `Key-`, followed by any string of characters. The characters `.*` represent a wildcard that matches any string.

```
regPtr1->registerRegex("Key-.*");
```

To unregister interest using regular expressions, you use the `unregisterRegex` function. The next example shows how to unregister interest in all keys whose first four characters are `Key-`, followed by any string (represented by the `.*` wildcard).

```
regPtr1->unregisterRegex("Key-.*");
```

# Register Interest Scenario

In this register interest scenario, a cache listener is used with a cacheless region that has `subscription-enabled` set to `true`. The client region is configured with caching disabled; client notification is enabled; and a cache listener is established. The client has not registered interest in any keys.

When a value changes in another client, it sends the event to the server. The server will not send the event to the cacheless client, even though `client-notification` is set to `true`.

To activate the cache listener so the cacheless region receives updates, the client should explicitly register interest in some or all keys by using one of the API calls for registering interest. This way, the client receives all events for the keys to which it has registered interest. This applies to Java-based clients as well as non-Java clients.

# Region Attributes

Region attributes govern the automated management of a region and its entries.

Region attribute settings determine where the data resides, how the region is managed in memory, and the automatic loading, distribution, and expiration of region entries.

## Specifying Region Attributes

Specify region attributes before creating the region. You can do this either through the API or through the declarative XML file. The API includes classes for defining a region's attributes before creation and for modifying some attributes after creation. For details, see the API for `RegionShortcut`, `RegionAttributes`, `RegionAttributesFactory`, and `AttributesMutator`.

## Region Shortcuts

VMware GemFire® provides predefined, shortcut region attributes settings for your use in `RegionShortcut`. The shortcuts are:

`PROXY`

> does not store data in the client cache, but connects the region to the servers.

`CACHING_PROXY`

> stores data in the client cache and connects the region to the servers.

`CACHING_PROXY_ENTRY_LRU`

> stores data in the client cache and connects the region to the servers. Limits the amount of data stored locally in the client to a default limit of 100,000 entries by ejecting the least recently used (LRU) entries.

`LOCAL`

> stores data in the client cache and does not connect the region to the servers. This is a client-side-only region.

`LOCAL_ENTRY_LRU`

stores data in the client cache and does not connect the region to the servers. This is a client-side-only region. Limits the amount of data stored locally in the client to a default limit of 100,000 entries by ejecting the least recently used (LRU) entries.

# Serializing Data

Data in your client application's VMware GemFire® cache must be serializable to be shared with VMware GemFire® servers and other VMware GemFire® clients. VMware GemFire® provides multiple data serialization options for storage and transmittal between processes, of which **VMware GemFire® Portable Data eXchange (PDX) serialization**) offers the best combination of versatility and ease-of-use for most applications.

To learn more about other serialization options, see the Data Serialization section in the *VMware GemFire® User Guide*.

# VMware GemFire® PDX Serialization

**In this topic**


Portability of PDX Serializable Objects

Reduced Deserialization of Serialized Objects

Delta Propagation with PDX Serialization

PDX Serialization Details


VMware GemFire®'s Portable Data eXchange (PDX) is a cross-language data format that can reduce the cost of distributing and serializing your objects.

VMware GemFire® C++ PDX serialization:

- Is interoperable with other languages by VMware GemFire® – no need to program a Java-side implementation
- Reduces deserialization overhead by providing direct field access on servers of serialized data, without full deserialization. Stores data in named fields that you can access individually, to avoid the cost of deserializing the entire data object
- Works with VMware GemFire® delta propagation

For greater control, you can specify individual treatment for domain objects using the `PdxSerializable` interface.

## Portability of PDX Serializable Objects

When you create a `PdxSerializable` object, VMware GemFire® stores the object's type information in a central registry. The information is passed between peers, between clients and servers, and between distributed systems.

When using PDX serialization, clients automatically pass registry information to servers when they store a `PdxSerializable` object. Clients can run queries and functions against the data in the servers without the servers needing to know anything about the stored objects. One client can store data on the server to be retrieved by another client, with the server never needing to know the object type. This means you can code your C++ clients to manage data using Java servers without having to create Java implementations of your C++ domain objects.

# Reduced Deserialization of Serialized Objects

The access methods for `PdxSerializable` objects allow you to examine specific fields of your domain object without deserializing the entire object. This can reduce deserialization costs significantly. Client C++ apps can run queries and execute functions against the objects in the server caches without deserializing the entire object on the server side. The query engine automatically recognizes PDX objects and uses only the fields it needs.

Clients can execute Java functions on server data that only access parts of the domain objects by using `PdxInstance.`

Likewise, peers can access just the fields needed from the serialized object, keeping the object stored in the cache in serialized form.

# Delta Propagation with PDX Serialization

You can use VMware GemFire® delta propagation with PDX serialization.

# PDX Serialization Details

See the following sections for details on implementing PDX serialization:

- Using the PdxSerializable Abstract Class
- PdxSerializable Example

# Using the PdxSerializable Abstract Class

When you write objects using PDX serialization, they are distributed to the server tier in PDX serialized form. Domain classes need to inherit the `PdxSerializable` abstract class to serialize and de-serialize the object.

When you run queries against the objects on the servers, only the fields you specify are deserialized. A domain class should serialize and de-serialize all its member fields in the same order in its `toData` and `fromData` functions.

Use this procedure to program your domain object for PDX serialization using the `PdxSerializable` abstract class.

1. In your domain class, implement `PdxSerializable`. For example:

   ```
   class Order : public PdxSerializable {
   ```

2. Program the `toData` function to serialize your object as required by your application. (See `markIdentityField` in a later step for an optimization that you can apply to this code sample.)

   ```
   void Order::toData(PdxWriter& pdxWriter) const {
     pdxWriter.writeInt(ORDER_ID_KEY_, order_id_);
     pdxWriter.writeString(NAME_KEY_, name_);
     pdxWriter.writeShort(QUANTITY_KEY_, quantity_);
   }
   ```

   If you also use PDX serialization in Java or .NET for the object, serialize the object in the same way for each language. Serialize the same fields in the same order and mark the same identity fields.

3. Program the `fromData` function to read your data fields from the serialized form into the object's fields.

   ```
   void Order::fromData(PdxReader& pdxReader) {
     order_id_ = pdxReader.readInt(ORDER_ID_KEY_);
     name_ = pdxReader.readString(NAME_KEY_);
     quantity_ = pdxReader.readShort(QUANTITY_KEY_);
   }
   ```

   In your `fromData` implementation, use the same name as you did in `toData` and call the read operations in the same order as you called the write operations in your `toData` implementation.

4. Optionally, program your domain object's `hashCode` and equality functions. When you do so, you can optimize those functions by specifying the *identity fields* to be used in comparisons.

    - Marked identity fields are used to generate the `hashCode` and equality functions of PdxInstance, so the identity fields should themselves either be primitives, or implement `hashCode` and `equals`.
    - The `markIdentityField` function indicates that the given field name should be included in `hashcode` and equality checks of this object on a server.
    - Invoke the `markIdentityField` function directly after the identity field's `write*` function.
    - If no fields are set as identity fields, then all fields will be used in `hashCode` and equality checks, so marking identity fields improves the efficiency of hashing and equality operations.
    - It is important that the fields used by your equality function and `hashCode` implementations are the same fields that you mark as identity fields.

    This code sample expands the sample from the description of the `toData` function, above, to illustrate the use of `markIdentityField`:

```
void Order::toData(PdxWriter& pdxWriter) const {
  pdxWriter.writeInt(ORDER_ID_KEY_, order_id_);
  pdxWriter.markIdentityField(ORDER_ID_KEY_);

  pdxWriter.writeString(NAME_KEY_, name_);
  pdxWriter.markIdentityField(NAME_KEY_);

  pdxWriter.writeShort(QUANTITY_KEY_, quantity_);
  pdxWriter.markIdentityField(QUANTITY_KEY_);
}
```

# PdxSerializable Example

The native client release contains an example showing how a client application can register for serialization of custom objects using the C++ PdxSerializable abstract class.

The example is located in `examples/cpp/pdxserializable`.

The example defines the serializable class, `Orders`, including its serialization and deserialization methods and its factory method. Once these pieces are in place, execution is simple: the main routine of the example registers the serializable class then performs some put and get operations.

## Execution

The example performs a sequence of operations, displaying simple log entries as they run.

- To run the example, follow the instructions in the README.md file in the example directory.
- Review the source code in the example directory to see exactly how it operates.
- Begin by running a script that sets up the server-side environment by invoking `gfsh` commands to create a region, a locator, and a server.
- Run the example client application, which performs the following steps:
  - Connects to the server
  - Registers the PdxSerializable class
  - Creates orders
  - Stores orders
  - Retrieves orders

## C++ Example

This section contains code snippets showing highlights of the C++ PdxSerialiable example. They are not intended for cut-and-paste execution. For the complete source, see the example source directory.

The C++ example defines a PdxSerializable class called `Order` that inherits from the `PdxSerializable` abstract class. An `Order` object contains three fields:

© Copyright Pivotal Software Inc, 2013-2020

- an integer `order_id`
- a string `name`
- a short-int `quantity`

From Order.hpp:

```
class Order : public PdxSerializable {
 public:
 ...

 private:
  int32_t order_id_;
  std::string name_;
  int16_t quantity_;
};
```

Using the `PdxSerializable` read and write methods, the `Order` class defines `fromData()` and `toData()` methods that perform the deserialization and serialization operations, respectively, and the `createDeserializable()` factory method:

From Order.cpp:

```
void Order::fromData(PdxReader& pdxReader) {
  order_id_ = pdxReader.readInt(ORDER_ID_KEY_);
  name_ = pdxReader.readString(NAME_KEY_);
  quantity_ = pdxReader.readShort(QUANTITY_KEY_);
}

void Order::toData(PdxWriter& pdxWriter) const {
  pdxWriter.writeInt(ORDER_ID_KEY_, order_id_);
  pdxWriter.markIdentityField(ORDER_ID_KEY_);

  pdxWriter.writeString(NAME_KEY_, name_);
  pdxWriter.markIdentityField(NAME_KEY_);

  pdxWriter.writeShort(QUANTITY_KEY_, quantity_);
  pdxWriter.markIdentityField(QUANTITY_KEY_);
}

...

std::shared_ptr<PdxSerializable> Order::createDeserializable() {
  return std::make_shared<Order>();
}
```

The C++ example mainline creates a cache, then uses it to create a connection pool and a region object (of class `Region`).

```
auto cacheFactory = CacheFactory();
cacheFactory.set("log-level", "none");
auto cache = cacheFactory.create();
auto poolFactory = cache.getPoolManager().createFactory();

poolFactory.addLocator("localhost", 10334);
auto pool = poolFactory.create("pool");
auto regionFactory = cache.createRegionFactory(RegionShortcut::PROXY);
auto region = regionFactory.setPoolName("pool").create("custom_orders");
```

The client registers the PdxSerializable class that was created in Orders.cpp:

```
cache.getTypeRegistry().registerPdxType(Order::createDeserializable);
```

The client then instantiates and stores two `Order` objects:

```
auto order1 = std::make_shared<Order>(1, "product x", 23);
auto order2 = std::make_shared<Order>(2, "product y", 37);

region->put("Customer1", order1);
region->put("Customer2", order2);
```

Next, the application retrieves the stored values, in one case extracting the fields defined in the serialization code:

```
if (auto order1retrieved =
        std::dynamic_pointer_cast<Order>(region->get("Customer1"))) {
  std::cout << "OrderID: " << order1retrieved->getOrderId() << std::endl;
  std::cout << "Product Name: " << order1retrieved->getName() << std::endl;
  std::cout << "Quantity: " << order1retrieved->getQuantity() << std::endl;
} else {
  std::cout << "Order 1 not found." << std::endl;
}
```

The application retrieves the second object and displays it without extracting the separate fields:

```
if (region->existsValue("rtimmons")) {
  std::cout << "rtimmons's info not deleted" << std::endl;
} else {
  std::cout << "rtimmons's info successfully deleted" << std::endl;
}
```

Finally, the application closes the cache:

```
cache.close();
```

# Remote Queries

**In this topic**

Remote Query Basics

- Query language: OQL
- Creating Indexes

Remote Query API

- Query
- Executing a Query from the Client
- C++ Query Example

Use the remote query API to query your cached data stored on a cache server.

## Remote Query Basics

Queries are evaluated and executed on the cache server, and the results are returned to the client. You can optimize your queries by defining indexes on the cache server.

Querying and indexing operate only on remote cache server contents.

### Query language: OQL

VMware GemFire® provides a SQL-like querying language called OQL that allows you to access data stored in VMware GemFire® regions. OQL is very similar to SQL, but OQL allows you to query complex objects, object attributes, and methods.

In the context of a query, specify the name of a region by its full path, starting with a slash (`/`).

The query language supports drilling down into nested object structures. Nested data collections can be explicitly referenced in the FROM clause of a query.

A query execution returns its results as either a `ResultSet` or a `StructSet`.

Query language features and grammar are described in the VMware GemFire® manual at Querying.

### Creating Indexes

Indexes can provide significant performance gains for query execution. You create and maintain indexes on the cache server. For detailed information about working with indexes configured on a cache server, see Working with Indexes in the server's documentation.

# Remote Query API

This section gives a general overview of the interfaces and classes that are provided by the query package API.

## Query

You must create a `Query` object for each new query. The `Query` interface provides methods for managing the compilation and execution of queries, and for retrieving an existing query string.

A `Query` is obtained from a `QueryService`, which is obtained from one of two sources:

- To create a `Query` that operates on the VMware GemFire® server, use `apache::geode::client::Pool::getQueryService()` to instantiate a `QueryService` obtained from a `Pool`.
- To create a `Query` that operates on your application's local cache, use `apache::geode::client::Cache::getQueryService()` to instantiate a `QueryService` obtained from a `Cache`.

## Executing a Query from the Client

The essential steps to create and execute a query are:

1. Create an instance of the `QueryService` class. If you are using the pool API (recommended), you should obtain the `QueryService` from the pool.
2. Create a `Query` instance that is compatible with the OQL specification.
3. Use the `Query.execute()` method to submit the query string to the cache server. The server remotely evaluates the query string and returns the results to the client.
4. Iterate through the returned objects.

## C++ Query Example

These C++ code excerpts are from the `examples/cpp/remotequery` example included in your client distribution. See the example for full context.

© Copyright Pivotal Software Inc, 2013-2020

Following the steps listed above,

1. Obtain a `QueryService` object from the connection pool:

```
std::shared_ptr<QueryService> queryService = nullptr;
queryService = pool->getQueryService();
```

2. Create a `Query` object by calling `QueryService.newQuery()`, specifying your OQL query as a string parameter:

```
auto query = queryService->newQuery("SELECT * FROM /custom_orders WHERE quantity > 30");
```

3. Execute the query. Collect the query output, returned as either a `ResultSet` or a `StructSet`, and iterate through the results:

```
auto queryResults = query->execute();

for (auto&& value : *queryResults) {
  auto&& order = std::dynamic_pointer_cast<Order>(value);
  std::cout << order->toString() << std::endl;
}
```

# Continuous Queries

**In this topic**

Continuous Query Basics

Typical Continuous Query Lifecycle

Executing a Continuous Query from the Client

C++ Continuous Query Example

The C++ and .NET clients can initiate queries that run on the VMware GemFire® cache server and notify the client when the query results have changed. For details on the server-side setup for continuous queries, see How Continuous Querying Works in the *VMware GemFire® User Guide*.

## Continuous Query Basics

Continuous querying provides the following features:

- **Standard VMware GemFire® native client query syntax and semantics**. Continuous queries are expressed in the same language used for other native client queries. See Remote Queries.
- **Standard VMware GemFire® events-based management of CQ events.** The event handling used to process CQ events is based on the standard VMware GemFire® event handling framework.
- **Complete integration with the client/server architecture**. CQ functionality uses existing server-to-client messaging mechanisms to send events. All tuning of your server-to-client messaging also tunes the messaging of your CQ events. If your system is configured for high availability then your CQs are highly available, with seamless failover provided in case of server failure (see High Availability for Client-to-Server Communication ). If your clients are durable, you can also define any of your CQs as durable (see Durable Client Messaging ).
- **Interest criteria based on data values**. Continuous queries are run against the region's entry values. Compare this to register interest by reviewing Registering Interest for Entries.
- **Active query execution**. Once initialized, the queries operate on new events. Events that change the query result are sent to the client immediately.

## Typical Continuous Query Lifecycle

1. The client creates the CQ. This sets up everything for running the query and provides the client with a `CqQuery` object, but does not execute the CQ. At this point, the query is in a `STOPPED` state,

© Copyright Pivotal Software Inc, 2013-2020 

ready to be closed or run.

2. The client initiates the CQ with an API call to one of the `CqQuery execute*` methods. This puts the query into a `RUNNING` state on the client and on the server. The server remotely evaluates the query string, and optionally returns the results to the client. `CqQuery execute*` methods include:
    - `CqQuery.execute()`
    - `CqQuery.executeWithInitialResults()`
3. A CQ Listener waits for events. When it receives events, it takes action accordingly with the data in the CqEvent.
4. The CQ is closed by a client call to `CqQuery.close`. This de-allocates all resources in use for the CQ on the client and server. At this point, the cycle could begin again with the creation of a new `CqQuery` instance.

## Executing a Continuous Query from the Client

The essential steps to create and execute a continuous query are:

1. Create an instance of the `QueryService` class. If you are using the pool API (recommended), you should obtain the `QueryService` from the pool.
2. Define a CQ Listener (a `CqListener`) to field events sent from the server.
3. Use one of the `CqQuery execute*` methods to submit the query string to the cache server.
4. The server remotely evaluates the query string, then monitors those results and notifies the client if they change.
5. The client listens for changes that match the query predicate.
6. Iterate through the returned objects.
7. When finished, close down the continuous query.

### C++ Continuous Query Example

These C++ code excerpts are from the `examples/cpp/continuousquery` example included in your client distribution. See the example for full context.

© Copyright Pivotal Software Inc, 2013-2020

Following the steps listed above,

1. Create a query service:

```
auto queryService = pool->getQueryService();
```

2. Define a CqListener:

```
class MyCqListener : public CqListener {
```

3. Create an instance of your CqListener and insert it into a CQ attributes object:

```
CqAttributesFactory cqFactory;

auto cqListener = std::make_shared<MyCqListener>();

cqFactory.addCqListener(cqListener);
auto cqAttributes = cqFactory.create();
```

4. Create a Continuous Query using the query service and the CQ attributes:

```
auto query = queryService->newCq(
"MyCq", "SELECT * FROM /custom_orders c WHERE c.quantity > 30",
cqAttributes);
```

5. Execute the query:

```
query->execute();
```

6. Wait for events and do something with them.

```
/* Excerpt from the CqListener */

/* Determine Operation Type */
switch (cqEvent.getQueryOperation()) {
case CqOperation::OP_TYPE_CREATE:
  opStr = "CREATE";
  break;
case CqOperation::OP_TYPE_UPDATE:
  opStr = "UPDATE";
  break;
case CqOperation::OP_TYPE_DESTROY:
  opStr = "DESTROY";
  break;
default:
  break;
}

...

/* Take action based on OP Type */
```

7. When finished, close up shop.

```
query->execute();

... (respond to events as they arrive)

query->stop();
query->close();

cache.close();
```

# Security: Authentication and Encryption

Most security configuration takes place on the VMware GemFire® server. The server's security framework authenticates clients as they connect to a cache server and authorizes client cache operations using developer-provided implementations for authentication and authorization.

For an explanation of the server-side implementation of security features, see Security in the *VMware GemFire® User Guide*.

A Native Client application must address two security concerns when connecting to a VMware GemFire® server:

- **Authentication**
  The Client must submit its authentication credentials to the server using the developer-provided authentication implementation expected by the server.
- **TLS/SSL Client/Server Communication Encryption**
  Communication between client and server should be encrypted so authentication credentials and other transmissions cannot be viewed by third-parties.

# Authentication

A client is authenticated when it connects with valid credentials to a VMware GemFire® cache server that is configured with the client authentication callback. For details on the server's role in authentication and what it expects from the client, see Implementing Authentication in the *VMware GemFire® User Guide*.

In your application, authentication credentials must be set when creating the cache. In practice, this means setting the authentication credentials when you create the CacheFactory.

## C++ Authentication Example

In this C++ authentication example, the `CacheFactory` creation process sets the authentication callback:

```
auto cacheFactory = CacheFactory(config);
auto authInitialize = std::make_shared<UserPasswordAuthInit>();
cacheFactory.set("log-level", "none");
cacheFactory.setAuthInitialize(authInitialize);
```

Credentials are implemented in the `getCredentials` member function of the `AuthInitialize` abstract class.

© Copyright Pivotal Software Inc, 2013-2020 

```
class UserPasswordAuthInit : public AuthInitialize {
public:
  UserPasswordAuthInit() = default;

  ~UserPasswordAuthInit() noexcept override = default;

  std::shared_ptr<Properties> getCredentials(
    const std::shared_ptr<Properties> &securityprops,
    const std::string &) override {
    std::shared_ptr<Cacheable> userName;
    if (securityprops == nullptr ||
      (userName = securityprops->find(SECURITY_USERNAME)) == nullptr) {
      throw AuthenticationFailedException(
      "UserPasswordAuthInit: user name "
      "property [SECURITY_USERNAME] not set.");
    }

    auto credentials = Properties::create();
    credentials->insert(SECURITY_USERNAME, userName->toString().c_str());
    auto passwd = securityprops->find(SECURITY_PASSWORD);
    if (passwd == nullptr) {
      passwd = CacheableString::create("");
    }
    credentials->insert(SECURITY_PASSWORD, passwd->value().c_str());
    return credentials;
  }

  void close() override { return; }
};
```

# TLS/SSL Client-Server Communication Encryption

This section describes how to implement TLS-based communication between your clients and servers using the OpenSSL encryption utility. When configuring TLS/SSL security for your client, you may find it helpful to refer to The SSL section of the VMware GemFire® User Guide.

## Set Up OpenSSL

The open-source OpenSSL toolkit provides a full-strength general purpose cryptography library for encrypting client-server communications.

Download and install OpenSSL 1.1.1 for your specific operating system.

**Notes for Windows users:**

- For Windows platforms, you can use either the regular or the “Light” version of SSL.
- Use a 64-bit implementation of OpenSSL.
- If you use Cygwin, do not use the OpenSSL library that comes with Cygwin, which is built with `cygwin.dll` as a dependency. Instead, download a fresh copy from OpenSSL.
- For many Windows applications, the most convenient way to install OpenSSL is to use `choco` (see chocolatey.org) to install the “Light” version of OpenSSL.

## Step 1. Create keystores

The VMware GemFire® server requires keys and keystores in the Java Key Store (JKS) format while the native client requires them in the clear PEM format. Thus you need to be able to generate private/public keypairs in either format and convert between the two using the `keytool` utility and the `openssl` command.

## Step 2. Enable SSL on the server and on the client

1. On the server, enable SSL for the `locator` and `server` components, as the SSL-enabled client must be able to communicate with both locator and server components.

2. On the client, set `ssl-enabled` to `true`.

3. On the client, set `ssl-keystore` and `ssl-truststore` to point to your keystore files. Paths to the keystore and truststore are local to the client. See Security-Related System Properties for a description of

these properties.

## Starting and stopping the client and server with SSL in place

Before you start and stop the client and server, make sure you configure the native client with the SSL properties as described and with the servers or locators specified as usual.

Specifically, ensure that:

- The OpenSSL and VMware GemFire® DLLs are in the right environment variables for your system: `PATH` for Windows, and `LD_LIBRARY_PATH` for Unix.
- You have generated the keys and keystores.
- You have set the system properties.

For details on stopping and starting locators and cache servers with SSL, see Starting Up and Shutting Down Your System.

The VMware GemFire® Native's libcryptoImpl found in /lib must be linked at compile time. This binary is used to interact with OpenSSL. Link libcryptoImpl, native client, and your application code. We highly recommend using cmake.

### Example locator start command

Ensure that all required SSL properties are configured in your server's `geode.properties` file. Then start your locator as follows:

```
gfsh>start locator --name=my_locator --port=12345 --dir=. \
--security-properties-file=/path/to/your/geode.properties
```

### Example locator stop command

```
gfsh>stop locator --port=12345 \
--security-properties-file=/path/to/your/geode.properties
```

### Example server start command

Again, ensure that all required SSL properties are configured in `geode.properties`. Then start the server with:

© Copyright Pivotal Software Inc, 2013-2020

```
gfsh>start server --name=my_server --locators=hostname[12345] \
--cache-xml-file=server.xml --log-level=fine \
--security-properties-file=/path/to/your/geode.properties
```

### Example server stop command

```
gfsh>stop server --name=my_server
```

# Function Execution

**In this topic**


Server-side Requirements

Client-side Requirements

How Functions Execute

Processing Function Results

Function Execution Example

C++ Example


A client can invoke a server-resident function, with parameters, and can collect and operate on the returned results.

## Server-side Requirements

To be callable from your client, a function must be

- resident on the server, and
- registered as available for client access.

See Executing a Function in VMware GemFire® in the VMware GemFire® User Guide for details on how to write and register server-resident functions.

## Client-side Requirements

The client must connect to the server through a connection pool in order to invoke a server-side function.

## How Functions Execute

1. The calling client application runs the `execute` method on the `Execution` object. The function must already be registered on the servers.
2. The function is invoked on the servers where it needs to run. The servers are determined by the `FunctionService on*` method calls, region configuration, and any filters.
3. If the function has results, the result is returned in a `ResultCollector` object.

© Copyright Pivotal Software Inc, 2013-2020

4. The client collects results using the `ResultCollector.getResult()` method.

In every client where you want to execute the function and process the results:

- Use one of the `FunctionService on*` methods to create an `Execution` object. The `on*` methods, `onRegion`, `onServer` and `onServers`, define the highest level where the function is run.
- If you use `onRegion` you can further narrow your run scope by setting key filters.
- A function run using `onRegion` is a *data dependent* function – others are *data-independent* functions.
- You can run a data dependent function against partitioned and colocated partitioned regions. From the client, provide the appropriate key sets to the function call.
- The `Execution` object allows you to customize the invocation by:
  - Providing a set of data keys to `withFilter` to narrow the execution scope. This works only for `onRegion` Execution objects (data-dependent functions).
  - Providing function arguments to `withArgs`.
  - Defining a custom `ResultCollector` for `withCollector`.
- Call the `Execution.execute()` method to run the function.

## Processing Function Results

To get the results from the function in the client app, use the result collector returned from the function execution. The `getResult` methods of the default result collector block until all results are received, then return the full result set.

The client can use the default result collector. If the client needs special results handling, code a custom `ResultsCollector` implementation to replace the default. Use the `Execution::withCollector` method to specify the custom collector. To handle the results in a custom manner:

1. Write a class that implements the `ResultCollector` interface to handle the results in a custom manner. The methods are of two types: one handles data and information from VMware GemFire® and populates the results set, while the other returns the compiled results to the calling application:
   - `addResult` is called when results arrive from the `Function` methods. Use `addResult` to add a single result to the ResultCollector.
   - `endResults` is called to signal the end of all results from the function execution.
   - `getResult` is available to your executing application (the one that calls `Execution.execute`) to retrieve the results. This may block until all results are available.
   - `clearResults` is called to clear partial results from the results collector. This is used only for

highly available `onRegion` functions where the calling application waits for the results. If the call fails, before VMware GemFire® retries the execution, it calls `clearResults` to ready the instance for a clean set of results.

2. Use the `Execution` object in your executing member to call `withCollector`, passing your custom collector.

## Function Execution Example

The native client release contains examples of function execution in `../examples/cpp/functionexecution`.

- The example begins with a server-side script that runs `gfsh` commands to create a region, simply called “partition_region”.
- The function is preloaded with a JAR file containing the server-side Java function code.
- The function, called “ExampleMultiGetFunction”, is defined in the `examples/utilities` directory of your distribution. As its input parameter, the function takes an array of keys, then performs a `get` on each key and returns an array containing the results.
- The function does not load values into the data store. That is a separate operation, performed in these examples by the client, and does not involve the server-side function.

As prerequisites, the client code must be aware of the connection to the server, the name of the function, and the expected type/format of the input parameter and return value.

The client:

- creates an execution object
- provides the execution object with a populated input parameter array
- invokes the object’s execute method to invoke the server-side function

If the client expects results, it must create a result object. The .NET example uses a built-in result collector (`IResultCollector.getResults()`) to retrieve the function results.

The example creates a result variable to hold the results from the collector.

### C++ Example

This section contains code snippets showing highlights of the C++ function execution example. They are not intended for cut-and-paste execution. For the complete source, see the example source directory.

The C++ example creates a cache.

```
Cache setupCache() {
  return CacheFactory()
      .set("log-level", "none")
      .create();
}
```

The example client uses the cache to create a connection pool,

```
void createPool(const Cache& cache) {
  auto pool = cache.getPoolManager()
      .createFactory()
      .addServer("localhost", EXAMPLE_SERVER_PORT)
      .create("pool");
}
```

Then, using that pool, the client creates a region with the same characteristics and name as the server-side region ( `partition_region` ).

```
std::shared_ptr<Region> createRegion(Cache& cache) {
  auto regionFactory = cache.createRegionFactory(RegionShortcut::PROXY);
  auto region = regionFactory.setPoolName("pool").create("partition_region");

  return region;
}
```

The sample client populates the server's datastore with values, using the API and some sample key-value pairs.

```
void populateRegion(const std::shared_ptr<Region>& region) {
  for (int i = 0; i < keys.size(); i++) {
    region->put(keys[i], values[i]);
  }
}
```

As confirmation that the data has been stored, the sample client uses the API to retrieve the values and write them to the console. This is done without reference to the server-side example function.

```
std::shared_ptr<CacheableVector> populateArguments() {
  auto arguments = CacheableVector::create();
  for (int i = 0; i < keys.size(); i++) {
    arguments->push_back(CacheableKey::create(keys[i]));
  }
  return arguments;
}
```

Next, the client retrieves those same values using the server-side example function. The client code creates the input parameter, an array of keys whose values are to be retrieved.

```
std::vector<std::string> executeFunctionOnServer(const std::shared_ptr<Region> region,
    const std::shared_ptr<CacheableVector> arguments) {
  std::vector<std::string> resultList;
```

The client creates an execution object using `Client.FunctionService.OnRegion` and specifying the region.

```
  auto functionService = FunctionService::onServer(region->getRegionService());
```

The client then calls the server side function with its input arguments and stores the results in a vector.

```
  if(auto executeFunctionResult = functionService.withArgs(arguments).execute(getFuncIName)->getResult()) {
    for (auto &arrayList: *executeFunctionResult) {
      for (auto &cachedString: *std::dynamic_pointer_cast<CacheableArrayList>(arrayList)) {
        resultList.push_back(std::dynamic_pointer_cast<CacheableString>(cachedString)->value());
      }
    }
  } else {
    std::cout << "get executeFunctionResult is NULL\n";
  }

  return resultList;
}
```

It then loops through the results vector and prints the retrieved values.

```
void printResults(const std::vector<std::string>& resultList) {
  std::cout << "Result count = " << resultList.size() << std::endl << std::endl;
  int i = 0;
  for (auto &cachedString: resultList) {
    std::cout << "\tResult[" << i << "]=" << cachedString << std::endl;
    ++i;
  }
```

© Copyright Pivotal Software Inc, 2013-2020

# Transactions

**In this topic**


Native Client Transaction APIs

Running Native Client Transactions

Client Transaction Examples

- C++ Example


The Native Client API runs transactions on the server as if they were local to the client application. Thus, the key to running client transactions lies in making sure the server is properly configured and programmed. For complete information about how transactions are conducted on the VMware GemFire® server, see the Transactions section of the VMware GemFire® User Guide.

## Native Client Transaction APIs

The API for distributed transactions has the familiar relational database methods, `begin`, `commit`, and `rollback`. There are also APIs available to suspend and resume transactions.

The C++ classes for executing transactions are:

- `apache.geode.client.CacheTransactionManager`
- `apache.geode.client.TransactionId`

## Running Native Client Transactions

The syntax for writing client transactions is the same as with server or peer transactions, but when a client performs a transaction, the transaction is delegated to a server that brokers the transaction.

Start each transaction with a `begin` operation, and end the transaction with a `commit` or a `rollback`.

To maintain cache consistency, the local client cache is not used during a transaction. When the transaction completes or is suspended, local cache usage is reinstated.

If the transaction runs on server regions that are a mix of partitioned and replicated regions, perform the first transaction operation on a partitioned region. This sets the server data host for the entire transaction. If you are using PR single-hop, single-hop will be applied as usual to this first operation.

© Copyright Pivotal Software Inc, 2013-2020

In addition to the failure conditions common to all transactions, client transactions can also fail if the transaction delegate fails. If the delegate performing the transaction fails, the transaction code throws a `TransactionException`.

## Client Transaction Examples

The native client release contains a transaction example in `../examples/cpp/transaction`.

The example performs a sequence of operations, displaying simple log entries as they run.

- To run an example, follow the instructions in the `README.md` file in the example directory.
- Review the source code in the example directory to see exactly how it operates.
- You begin by running a script that sets up the server-side environment by invoking `gfsh` commands to create a region, simply called “exampleRegion.”
- You run the example client application, which performs the following steps:
  - Connects to the server
  - Begins a transaction
  - Performs some `put` operations
  - Commits the transaction
- For this example, the transaction code has these characteristics:
  - To introduce the possibility of failure, values are randomized from 0 to 9, and the 0 values are treated as unsuccessful. The transaction is retried until it succeeds.
  - In case the transaction repeatedly fails, the retry loop uses a counter to set a limit of 5 retries.

### C++ Example

This section contains code snippets showing highlights of the C++ transaction example. They are not intended for cut-and-paste execution. For the complete source, see the example source directory.

The C++ example creates a cache, then uses it to create a connection pool.

```
auto cache = CacheFactory().set("log-level", "none").create();
auto poolFactory = cache.getPoolManager().createFactory();

poolFactory.addLocator("localhost", 10334);
auto pool = poolFactory.create("pool");
auto regionFactory = cache.createRegionFactory(RegionShortcut::PROXY);
auto region = regionFactory.setPoolName("pool").create("exampleRegion");
```

© Copyright Pivotal Software Inc, 2013-2020

The example application gets a transaction manager from the cache and begins a transaction.

```
auto transactionManager = cache.getCacheTransactionManager();

transactionManager->begin();
```

Within the transaction, the client populates data store with 10 values associated with Key1 - Key10.

```
for (auto& key : keys) {
  auto value = getValueFromExternalSystem();
  region->put(key, value);
}
```

If all `put` operations succeed, the application commits the transaction. Otherwise, it retries up to 5 times if necessary.

```
auto retries = 5;
while (retries--) {
  try {
    transactionManager->begin();
     ... // PUT OPERATIONS ...
    transactionManager->commit();
    std::cout << "Committed transaction - exiting" << std::endl;
    break;
  } catch ( ... ) {
    transactionManager->rollback();
    std::cout << "Rolled back transaction - retrying(" << retries << ")" << std::endl;
  }
}
```

# System Properties

A variety of system properties can be specified when a client connects to a distributed system, either programmatically or in a `geode.properties` file. See `apache::geode::client::SystemProperties` in the C++ API docs
.

The following settings can be configured:

- General Properties
  Basic information for the process, such as cache creation parameters.
- Logging Properties
  How and where to log system messages.
- Statistics Archiving Properties
  How to collect and archive statistics information.
- Durable Client Properties
  Information about the durable clients connected to the system.
- System Properties for Client Authentication and Authorization
  Information about various security parameters.
- System Properties for High Availability
  System properties to configure periodic acknowledgment (ack).

The following tables list attributes that can be specified programmatically or stored in the `geode.properties` file to be read by a client.

## General Properties

| | | |
|---|---|---|
| cache-xml-file | Name and path of the file whose contents are used by default to configure a cache if one is created. If not specified, the client starts with an empty cache, which is populated at run time. | no default |
| heap-lru-delta | The percentage of entries the system will evict each time it detects that it has exceeded the heap-lru-limit. This property is used only if `heap-lru-limit` is greater than 0. | 10 % |
| heap-lru-limit | Maximum amount of memory, in megabytes, used by the cache for all regions. If this limit is exceeded by `heap-lru-delta` percent, LRU reduces the memory footprint as necessary. If not specified, or set to 0, memory usage is governed by each region's LRU entries limit, if any. | 0 |

| | | |
|---|---|---|
| conflate-events | Client side conflation setting, which is sent to the server. | server |
| connect-timeout | Amount of time (in seconds) to wait for a response after a socket connection attempt. | 59 |
| connection-pool-size | Number of connections per endpoint | 5 |
| enable-chunk-handler-thread | If the chunk-handler-thread is operative (enable-chunk-handler=true), it processes the response for each application thread. When the chunk handler is not operative (enable-chunk-handler=false), each application thread processes its own response. | false |
| disable-shuffling-of-endpoints | If true, prevents server endpoints that are configured in pools from being shuffled before use. | false |
| max-fe-threads | Thread pool size for parallel function execution. An example of this is the GetAll operations. | 2 * number of logical processors |
| max-socket-buffer-size | Maximum size of the socket buffers, in bytes, that the client will try to set for client-server connections. | 65 * 1024 |
| notify-ack-interval | Interval, in seconds, in which client sends acknowledgments for subscription notifications. | 1 |
| notify-dupcheck-life | Amount of time, in seconds, the client tracks subscription notifications before dropping the duplicates. | 300 |
| ping-interval | Interval, in seconds, between communication attempts with the server to show the client is alive. Pings are only sent when the `ping-interval` elapses between normal client messages. This must be set lower than the server's `maximum-time-between-pings`. | 10 |
| redundancy-monitor-interval | Interval, in seconds, at which the subscription HA maintenance thread checks for the configured redundancy of subscription servers. | 10 |
| tombstone-timeout | Time in milliseconds used to timeout tombstone entries when region consistency checking is enabled. | 480000 |

## Logging Properties

| | | |
|---|---|---|
| log- | | |

| | | |
|---|---|---|
| disk-space-limit | Maximum amount of disk space, in megabytes, allowed for all log files, current, and rolled. If set to 0, the space is unlimited. | 0 |
| log-file | Name and full path of the file where a running client writes log messages. If not specified, logging goes to `stdout`. | no default file |
| log-file-size-limit | Maximum size, in megabytes, of a single log file. Once this limit is exceeded, a new log file is created and the current log file becomes inactive. If set to 0, the file size is unlimited. | 0 |
| log-level | Controls the types of messages that are written to the application's log. These are the levels, in descending order of severity and the types of message they provide:<br>• **Error** (highest severity) is a serious failure that will probably prevent program execution.<br>• **Warning** is a potential problem in the system.<br>• **Info** is an informational message of interest to the end user and system administrator.<br>• **Config** is a static configuration message, often used to debug problems with particular configurations.<br>• **Fine, Finer, Finest, and Debug** provide tracing information. Only use these with guidance from technical support.<br><br>Enabling logging at any level enables logging for all higher levels. | config |

## Statistics Archiving Properties

| | | |
|---|---|---|
| statistic-sampling-enabled | Controls whether the process creates a statistic archive file. | true |
| statistic-archive-file | Name and full path of the file where a running system member writes archives statistics. If `archive-disk-space-limit` is not set, the client appends the process ID to the configured file name, like `statArchive-PID.gfs`. If the space limit is set, the process ID is not appended but each rolled file name is renamed to statArchive-ID.gfs, where ID is the rolled number of the file. | ./statArchive.gfs |
| archive-disk- | Maximum amount of disk space, in gigabytes, allowed for all archive | |

| | | |
|---|---|---|
| space-limit | files, current, and rolled. If set to 0, the space is unlimited. | 0 |
| archive-file-size-limit | Maximum size, in megabytes, of a single statistic archive file. Once this limit is exceeded, a new statistic archive file is created and the current archive file becomes inactive. If set to 0, the file size is unlimited. | 0 |
| statistic-sample-rate | Rate, in seconds, that statistics are sampled. Operating system statistics are updated only when a sample is taken. If statistic archival is enabled, then these samples are written to the archive.<br><br>Lowering the sample rate for statistics reduces system resource use while still providing some statistics for system tuning and failure analysis. | 1 |
| enable-time-statistics | Enables time-based statistics for the distributed system and caching. For performance reasons, time-based statistics are disabled by default. See System Statistics. | false |

## Durable Client Properties

| Attribute | Description | Default |
|---|---|---|
| auto-ready-for-events | Whether client subscriptions automatically receive events when declaratively configured via XML. If set to `false`, event startup is not automatic and you need to call the `Cache.ReadyForEvents()` method API after subscriptions for the server to start delivering events. | true |
| durable-client-id | Identifier to specify if you want the client to be durable. | empty |
| durable-timeout | Time, in seconds, a durable client’s subscription is maintained when it is not connected to the server before being dropped. | 300 |

## Security Properties

The table describes the security-related system properties for native client authentication and authorization.

See SSL Client/Server Communication.

### System Properties for Client Authentication and Authorization

| | | |
|---|---|---|
| `security-client-auth-factory` | Sets the key for the `AuthInitialize` factory function. | empty |
| `security-client-auth-library` | Registers the path to the `securityImpl.dll` library. | empty |
| `security-client-dhalgo` | Diffie-Hellman based credentials encryption is not supported. | null |
| `security-client-kspath` | Path to a .PEM file, which contains the public certificates for all Geode cache servers to which the client can connect through specified endpoints. | null |
| `ssl-enabled` | True if SSL connection support is enabled. | empty |
| `ssl-keystore` | Name of the .PEM keystore file, containing the client's private key. Not set by default. Required if `ssl-enabled` is true. | |
| `ssl-keystore-password` | Sets the password for the private key .PEM file for SSL. | null |
| `ssl-truststore` | Name of the .PEM truststore file, containing the servers' public certificate. Not set by default. Required if `ssl-enabled` is true. | |

## High Availability Properties

| | | |
|---|---|---|
| `notify-ack-interval` | Minimum period, in seconds, between two consecutive acknowledgment messages sent from the client to the server. | 10 |
| `notify-dupcheck-life` | Minimum time, in seconds, a client continues to track a notification source for duplicates when no new notifications arrive before expiring it. | 300 |

# Client Cache XML Reference

This section documents the XML elements you can use to configure your VMware GemFire® native client application.

To define a configuration using XML:

1. Set cache configuration parameters in a declarative XML file. By convention, this user guide refers to the file as `cache.xml`, but you can choose any name.

2. Specify the filename and path to your XML configuration file by setting the `cache-xml-file` property in the `geode.properties` file. If you do not specify path, the application will search for the file in its runtime startup directory.

For example:

```
cache-xml-file=cache.xml
```

When you run your application, the native client runtime library reads and applies the configuration specified in the XML file.

The declarative XML file is used to externalize the configuration of the client cache. The contents of the XML file correspond to APIs found in the `apache::geode::client` package for C++ applications, and the `Apache::Geode::Client` package for .NET applications.

Elements are defined in the Client Cache XSD file, named `cpp-cache-1.0.xsd`, which you can find in your native client distribution in the `xsds` directory, and online at `https://geode.apache.org/schema/cpp-cache/cpp-cache-1.0.xsd`.

## Cache Initialization File: XML Essentials

This section assumes you are familiar with XML. When creating a cache initialization file for your native client application, keep these basics in mind:

- Place an XML prolog at the top of the file. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

- Quote all parameter values, including numbers and booleans. For example:

```
concurrency-level="10"
caching-enabled="true"
```

Some types are specific to the VMware GemFire® cache initialization file:

- **Duration:** Time specified as a non-negative integer and a unit, with no intervening space. The recognized units are `h`, `min`, `s`, `ms`, `us`, and `ns`. For example:

```
idle-timeout = "5555ms"
statistic-interval = "10s"
update-locator-list-interval="5min"
```

- **Expiration:** Complex type consisting of a duration (integer + unit) and an action, where the action is one of `invalidate`, `destroy`, `local-invalidate`, or `local-destroy`. For example:

```
<expiration-attributes timeout="20s" action="destroy"/>
<expiration-attributes timeout="10s" action="invalidate"/>
```

- **Library:** Complex type consisting of a library name and a function name. Used by the client to invoke functions. For example:

```
<persistence-manager library-name="SqLiteImpl"
 library-function-name="createSqLiteInstance">
```

## Cache Initialization File Element Descriptions

This section shows the hierarchy of `<client-cache>` sub-elements that you use to configure VMware GemFire® caches and clients. The top-level element in this syntax is `<client-cache>`.

- `<client-cache>`
    - `<pool>`
        - `<locator>`
        - `<server>`
    - `<region>`
        - `<region-attributes>`
            - `<region-time-to-live>`
            - `<region-idle-time>`
            - `<entry-time-to-live>`
            - `<entry-idle-time>`
            - `<partition-resolver>`

- <cache-loader>
- <cache-listener>
- <cache-writer>
- <persistence-manager>

<pdx>

In the descriptions, elements and attributes not designated “required” are optional.

## <client-cache> Element

The <client-cache> element is the top-level element of the XSD file.

Your declarative cache file must include a schema of the following form:

```
<client-cache
 xmlns="http://geode.apache.org/schema/cpp-cache"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://geode.apache.org/schema/cpp-cache
   http://geode.apache.org/schema/cpp-cache/cpp-cache-1.0.xsd"
 version="1.0">
  ...
</client-cache>
```

### Attributes of <client-cache>

| Attribute | Description |
|---|---|
| version | Required. Must be “1.0” |

### Sub-elements of <client-cache>

<client-cache> must contain at least one of these sub-elements:

| Element | Minimum Occurrences | Maximum Occurrences |
|---|---|---|
| <pool> | 0 | unbounded |
| <region> | 0 | unbounded |
| <pdx> | 0 | 1 |

# <pool> Element

The <pool> element is a collection of the connections by which your client application communicates with the VMware GemFire® server.

- A connection can specify either a locator or a server.
- A `<pool>` must contain at least one connection, locator or server, and can contain multiples of either or both.

## Sub-elements of <pool>

A `<pool>` must contain at least one sub-element that specifies a connection, which can be either a server or a locator. Multiples of either or both types are permitted.

| Element | Minimum Occurrences | Maximum Occurrences |
|---|---|---|
| <locator> | 0 | unbounded |
| <server> | 0 | unbounded |

## Attributes of <pool>

| Attribute | Description | Default |
|---|---|---|
| name | String. Required. Name of the pool, used when connecting regions to this pool. | |
| free-connection-timeout | Duration. The amount of time to wait for a free connection if max-connections is set and all of the connections are in use. | 10s |
| load-conditioning-interval | Duration. The interval at which the pool checks to see if a connection to a given server should be moved to a different server to improve the load balance. | 5min |
| min-connections | Non-negative integer. The minimum number of connections to keep available at all times. When the pool is created, it will create this many connections. If 0 (zero), then connections are not made until an operation is performed that requires client-to-server communication. | 1 |
| | Integer >= -1. The maximum number of connections to be created. If all | |

| Attribute | Description | Default |
| --- | --- | --- |
| max-connections | of the connections are in use, an operation requiring a client to server connection blocks until a connection is available. A value of -1 means no maximum. | -1 |
| retry-attempts | Integer >= -1. The number of times to retry an operation after a timeout or exception. A value of -1 indicates that a request should be tried against every available server before failing. | -1 |
| idle-timeout | Duration. Sets the amount of time a connection can be idle before it expires. A value of 0 (zero) indicates that connections should never expire. | 5s |
| ping-interval | Duration. The interval at which the pool pings servers. | 10s |
| read-timeout | Duration. The amount of time to wait for a response from a server before timing out and trying the operation on another server (if any are available). | 10s |
| server-group | String. Specifies the name of the server group to which this pool should connect. If the value is null or "" then the pool connects to all servers. | "" |
| socket-buffer-size | String. The size in bytes of the socket buffer on each connection established. | 32768 |
| subscription-enabled | Boolean. When `true`, establish a server to client subscription. | false |
| subscription-message-tracking-timeout | String. The amount of time that messages sent from a server to a client will be tracked. The tracking is done to minimize duplicate events. Entries that have not been modified for this amount of time are expired from the list. | 900s |
| subscription-ack-interval | String. The amount of time to wait before sending an acknowledgement to the server for events received from server subscriptions. | 100ms |
| subscription-redundancy | String. Sets the redundancy level for this pool's server-to-client subscriptions. An effort is made to maintain the requested number of copies (one copy per server) of the server-to-client subscriptions. At most, one copy per server is made up to the requested level. If 0 then no redundant copies are kept on the servers. | 0 |
| statistic-interval | Duration. The interval at which client statistics are sent to the server. A value of 0 (zero) means do not send statistics. | 0ms (disabled) |
| pr-single-hop-enabled | String. When `true`, enable single hop optimizations for partitioned regions. | true |
| | Boolean. Sets the thread local connections policy for this pool. When `true` then any time a thread goes to use a connection from this pool it | |

| Attribute | Description | Default |
| --- | --- | --- |
| thread-local-connections | will check a thread local cache and see if it already has a connection in it. If so it will use it. If not it will get one from this pool and cache it in the thread local. This gets rid of thread contention for the connections but increases the number of connections the servers see. When `false` then connections are returned to the pool as soon as the operation being done with the connection completes. This allows connections to be shared among multiple threads keeping the number of connections down. | false |
| multiuser-authentication | Boolean. Sets the pool to use multi-user secure mode. If in multiuser mode, then app needs to get `RegionService` instance of `Cache`. | false |
| update-locator-list-interval | Duration. The frequency with which client updates the locator list. To disable this set its value to `std::chrono::milliseconds::zero()`. | |

## <locator>

`<locator>` is a sub-element of `<pool>` that defines a connection to a VMware GemFire® locator, specified by a host and port combination.

### Attributes of <locator>

| Attribute | Description |
| --- | --- |
| host | String. Locator host name. |
| port | Integer in the range 0 - 65535, inclusive. Locator port number. |

For example:

```
<locator host="stax01" port="1001" />
```

## <server>

`<server>` is a sub-element of `<pool>` that defines a connection to a VMware GemFire® server, specified by a host and port combination.

## Attributes of <server>

| Attribute | Description |
| --- | --- |
| host | String. Server host name. |
| port | Integer in the range 0 - 65535, inclusive. Server port number. |

For example:

```
<server host="motown01" port="2001" />
```

# <region>

You can specify 0 or more regions per `<client-cache>`. There is no maximum limit on the number of regions a `<client-cache>` can contain.

In order to connect to a VMware GemFire® server, a client application must define at least one region whose name corresponds to that of a region on the server.

Regions can be nested.

### Sub-elements of <region>

Use the `<region-attributes>` sub-element to specify most of the characteristics of a region. Regions may be nested.

| Element | Minimum Occurrences | Maximum Occurrences |
| --- | --- | --- |
| <region-attributes> | 0 | 1 |
| <region> | 0 | unbounded |

### Attributes of <region>

You can specify many attributes to configure a region, but most of these attributes are encapsulated in the `<region-attributes>` sub-element. The `<region>` element itself has only two attributes: a required name and an optional reference identifier.

| Attribute | Description |
| --- | --- |
| name | String. Required. |
| refid | String. |

## <region-attributes>

Specify 0 or 1 `<region-attributes>` element for each `<region>` you define.

If you specify a `<region-attributes>` element, you must specify at least one of these sub-elements. When more than one sub-element is specified, they must be defined in this order:

| Element | Type | Minimum Occurrences | Maximum Occurrences |
| --- | --- | --- | --- |
| <region-time-to-live> | expiration | 0 | 1 |
| <region-idle-time> | expiration | 0 | 1 |
| <entry-time-to-live> | expiration | 0 | 1 |
| <entry-idle-time> | expiration | 0 | 1 |
| <partition-resolver> | library | 0 | 1 |
| <cache-loader> | library | 0 | 1 |
| <cache-listener> | library | 0 | 1 |
| <cache-writer> | library | 0 | 1 |
| <persistence-manager> | list of properties | 0 | 1 |

### Attributes of <region-attributes>

| Attribute | Description | Default |
| --- | --- | --- |
| caching-enabled | Boolean. If true, cache data for this region in this process. If false, then no data is stored in the local process, but events and distributions will still occur, and the region can still be used to put and remove, etc. | true |
| cloning-enabled | Boolean. Sets cloning on region. | false |
| scope | Enumeration: `local`, `distributed-no-ack`, `distributed-ack` | |
| initial- | String. Sets the initial entry capacity for the region. | 10000 |

© Copyright Pivotal Software Inc, 2013-2020

capacity

| Attribute | Description | Default |
| --- | --- | --- |
| load-factor | String. Sets the entry load factor for the next `RegionAttributes` to be created. | 0.75 |
| concurrency-level | String. Sets the concurrency level of the next `RegionAttributes` to be created. | 16 |
| lru-entries-limit | String. Sets the maximum number of entries this cache will hold before using LRU eviction. A return value of zero, 0, indicates no limit. If disk-policy is `overflows`, must be greater than zero. | |
| disk-policy | Enumeration: `none`, `overflows`, `persist`. Sets the disk policy for this region. | none |
| endpoints | String. A list of `servername:port-number` pairs separated by commas. | |
| client-notification | Boolean true/false (on/off) | false |
| pool-name | String. The name of the pool to attach to this region. The pool with the specified name must already exist. | |
| concurrency-checks-enabled | Boolean: true/false. Enables concurrent modification checks. | true |
| id | String. | |
| refid | String. | |

## <region-time-to-live>

<region-time-to-live> specifies how long this region remains in the cache after the last create or update, and the expiration action to invoke when time runs out. A create or update operation on any entry in the region resets the region’s counter, as well. Get (read) operations do not reset the counter.

Use the `<expiration-attributes>` sub-element to specify duration and expiration action.  The attributes of <expiration-attributes> must be defined in this order:

| Attribute | Description |
| --- | --- |
| timeout | Duration, specified as an integer and units. Required. |
| action | Enumeration. One of: `invalidate`, `destroy`, `local-invalidate`, `local-destroy` |

## <region-idle-time>

<region-idle-time> specifies how long this region remains in the cache after the last access, and the expiration action to invoke when time runs out. The counter is reset after any access, including create, put, and get operations. Access to any entry in the region resets the region's counter, as well.

Use the `<expiration-attributes>` sub-element to specify duration and expiration action. The attributes of <expiration-attributes> must be defined in this order:

| Attribute | Description |
|---|---|
| timeout | Duration, specified as an integer and units. Required. |
| action | Enumeration. One of: `invalidate`, `destroy`, `local-invalidate`, `local-destroy` |

## <entry-time-to-live>

<entry-time-to-live> specifies how long this entry remains in the cache after the last create or update, and the expiration action to invoke when time runs out. Get (read) operations do not reset the counter.

Use the `<expiration-attributes>` sub-element to specify duration and expiration action. The attributes of <expiration-attributes> must be defined in this order:

| Attribute | Description |
|---|---|
| timeout | Duration, specified as an integer and units. Required. |
| action | Enumeration. One of: `invalidate`, `destroy`, `local-invalidate`, `local-destroy` |

## <entry-idle-time>

<entry-idle-time> specifies how long this entry remains in the cache after the last access, and the expiration action to invoke when time runs out. The counter is reset after any access, including create, put, and get operations.

Use the `<expiration-attributes>` sub-element to specify duration and expiration action. The attributes of

<expiration-attributes> must be defined in this order:

| Attribute | Description |
|---|---|
| timeout | Duration, specified as an integer and units. Required. |
| action | Enumeration. One of: `invalidate`, `destroy`, `local-invalidate`, `local-destroy` |

## <partition-resolver>

<partition-resolver> identifies a function by specifying `library-name` and `library-function-name`.

A partition resolver is used for single-hop access to partitioned region entries on the server side. This resolver implementation must match that of the `PartitionResolver` on the server side. See the API Class Reference for the **PartitionResolver** class.

For example:

```
<partition-resolver library-name="appl-lib"
 library-function-name="createTradeKeyResolver"/>
```

## <cache-loader>

<cache-loader> identifies a cache loader function by specifying `library-name` and `library-function-name`. See the API Class Reference for the **CacheLoader** class.

## <cache-listener>

<cache-listener> identifies a cache listener function by specifying `library-name` and `library-function-name`. See the API Class Reference for the **CacheListener** class.

© Copyright Pivotal Software Inc, 2013-2020 

## <cache-writer>

<cache-writer> identifies a cache writer function by specifying `library-name` and `library-function-name`. See the API Class Reference for the **CacheWriter** class.

## <persistence-manager>

For each region, if the disk-policy attribute is set to `overflows`, a persistence-manager plug-in must perform cache-to-disk and disk-to-cache operations. See the API Class Reference for the **PersistenceManager** class.

<persistence-manager> identifies a persistence manager function by specifying `library-name` and `library-function-name`. You can also specify a set of properties to be passed to the function as parameters.

The sub-element `<properties>` is a sequence of 0 or more `<property>` elements.

Each `<property>` is a name-value pair. Where the attributes must be specified in this order:

- `name`
- `value`

For example:

```
<region-attributes>
  <persistence-manager library-name="libSqLiteImpl.so" library-function-name="createSqLiteInstance">
    <properties>
      <property name="PersistenceDirectory" value="/xyz"/>
      <property name="PageSize" value="65536"/>
      <property name="MaxPageCount" value="1073741823"/>
    </properties>
  </persistence-manager>
</region-attributes>
```

## <pdx>

Specifies the configuration for the Portable Data eXchange (PDX) method of serialization.

Attributes of <pdx>

| Attribute | Description |
| --- | --- |
| ignore-unread-fields | Boolean. When `true`, do not preserve unread PDX fields during deserialization. You can use this option to save memory. Set this attribute to `true` only in members that are only reading data from the cache. |
| read-serialized | Boolean. When `true`, PDX deserialization produces a `PdxInstance` instead of an instance of the domain class. |