Advanced Gemfire

Tim Dalsing

# Introduction

The purpose of this document is to describe tips, tricks, techniques, and best practices that go beyond a basic introduction to Gemfire. This document expands on the Gemfire Fundamentals document. It is intended as a cookbook and reference to guide the developer in making optimal design and coding choices, and to provide samples that are beyond the simple examples provided in the Gemfire distribution.

Code samples and utility components can be found in the git repo at:

https://github.com/Pivotal-Field-Engineering/gemfire-fe.git

Many of the components and samples solve specific problems described in this document.

# JVM

The latest JDK should be used. As of Gemfire 8 JDK 1.8 is supported. There is a bug fix in JDK 8 that is critical for Gemfire (related to file descriptor leaks in the NIO module). The full JDK, rather than the JRE, should be used.

The jps command that comes with the JDK is very useful. It is similar to the \*nix ps command, except it only displays Java processes. Depending on how the JDK was installed this command may not be in the PATH.

# Getting Started

The easiest way to get going is to start a locator and a server on the desktop:

1. Unzip or untar the Gemfire distribution to some directory, or use brew on Mac as described in the official documentation
2. Set a GEMFIRE\_HOME variable that points to the distribution
3. Add the $GEMFIRE\_HOME/bin directory to the PATH
4. Create a directory when the locator and server will be started
5. Change to the directory
6. Start a locator via gfsh start locator --name=locator
7. Start a server via gfsh start server --name=server --locators=localhost[10334]
8. Use GFSH to configure and manage the single node cluster

To start multiple servers on a desktop, add a different name and port for each server:

gfsh start server --name=server1 --locators=localhost[10334] --port=40401

gfsh start server --name=server2 --locators=localhost[10334] --port=40402

…

A GFSH run script is very useful for setting up a local cluster, including PDX, disk store, and region configuration. An example file called start.txt:

start locator --name=locator

start server --name=server1 --locators=localhost[10334] --port=40401

start server --name=server2 --locators=localhost[10334] --port=40402

connect

create disk-store --name=pdx --dir=./

create disk-store --name=data --dir=./

sleep

configure pdx --persistent=true --disk-store=pdx --read-serialized=true

shutdown

sleep

start server --name=server1 --locators=localhost[10334] --port=40401

start server --name=server2 --locators=localhost[10334] --port=40402

create region --name=myregion --disk-store=data

Which is then executed via:

gfsh run --file=start.txt

# IDs and ID Generation

Selecting a data type for representing unique IDs for a distributed system is special challenge. A common strategy is to use values generated by the UUID class in the JDK. However, the resulting IDs have several problems:

The hashCode tends to be very non-uniform for values generated by UUID. This results in non-uniform distribution of data across partition regions.

Since the String value is fairly long, the equals method is somewhat slow. Since the key uses the hashCode and equals to determine identity, the equals method is called many times. This can affect performance in some situations.

A numeric value, such as int or long, is a better choice. If the ID is generated by simply incrementing a value, the distribution is perfectly uniform. Also, hashCode and equals are very fast and efficient. If the smaller range of values afforded by an int are sufficient, it is the best choice since the hashCode is simply the value and there won’t be any hashtable collisions. A long is required for most situations if the range of IDs cannot be determined in advance or controlled in some way.

Creating unique IDs for the keys to a region is a common problem. On a single server node this problem is relatively easy to solve. However, on a distributed system, where the IDs must be unique across the entire cluster, the problem is more difficult to solve.

There are a number of ways to generate unique IDs:

1. The next ID is stored in a partition region that is persistent. The key to the ID region is the domain of the ID (e.g., the name of the region where the ID is actually used). A Function is used to get the next ID and update the current ID. Since the Function can be configured to write only to the primary the server node the call is routed to be predictable. The update of the next ID must be synchronized, so this technique is not appropriate for high volume applications.
2. Each server node has a prefix that is the high byte for the next ID. Each server gets it’s own prefix. The first value for the ID is the prefix shifted to the highest byte. Then the ID is incremented from the initial value. A local region with persistence is used to store the next ID. A Function that randomly picks a server node to get the next ID is used. This option is very fast and efficient since the generation of IDs is distributed across cluster. Making sure the prefix is different on each server node is the most difficult aspect of this technique. This technique also works on applications that use the Gemfire client.

If the source system for the data supplies its own ID, it can be mapped to an internal ID using an xref region.

The Gemfire Utilities include a key framework with ID generation classes.

# Key Classes

A common design strategy in Gemfire applications is to use standard Java wrappers, such as Integer or Long, or String as the key class. A better solution is to use a different key class for each domain class. For example, an Account class should have an accompanying AccountKey class. This has several advantages:

1. The key-constraint attribute of the region ensures the right value is used when putting data into a region. If a String or Long is used it can easily be the wrong value.
2. If ID is changed (such as from String to Long) all of the method signatures that have the key class do not have to be changed.
3. Compound key structures that are used with co-location can be created.

The Gemfire Utilities key framework includes a base class for key classes.

If PDX is used in a client/server configuration, and the server is configured for read-serialized=true, a domain class cannot be used for the key on the client since the hashCode calculation will be performed on the PdxInstance in the server and on the domain class in the client. In the PdxInstance the hashCode calculation is performed on the raw bytes since it is not deserialized, so it would be nearly impossible to ensure the hashCode calculation in the domain class matches the calculation in the PdxInstance. When PDX is used and the server is configured for read-serialized=true a String, Integer, or Long should be used for the key.

# Xrefs

There are a number of situations where a query may not be the most efficient way to find objects in the cache. One important use case is the need to perform operations on a subset of the data in a region based on some criteria. A query will require returning a result set, then operating on the results. A more efficient option is to use xrefs that allow direct access to the keys and data without the need to build, parse, and execute a query.

An xref region has a key that represents some value that is “indexed,” that is, the lookup value. This could be an account number, social security number, etc. The value in the xref region is a list of keys for objects that match the indexed value. The xref region is normally a local region without persistence (since the data is in other regions it can be easily rebuilt). A CacheListener is used to maintain the xrefs.

If the regions involved in the xref are partition, a PartitionListener is used to rebuild the xrefs when a bucket is moved or a secondary becomes primary. This is due to the fact that the CacheListener only fires on the primary, so the xrefs won’t exist on the secondaries.

# Partition vs. Replicate Regions

The simplistic analysis that drives the decision to use partition or replicate regions mostly involves whether the data will fit in a single node. If the data comfortable fits in a single node a replicate region should be used. If the data does not fit in a single node use a partition region. However, there are a few subtleties that are missing from this analysis:

1. Replicate regions do not perform as well for write-heavy use cases. Partition regions are better suited for this even if the amount of data is very small. This is particularly true for large clusters with many nodes.
2. Writing the same key in a replicate region on different servers simultaneously can result in locking problems and poor performance. A Function that is called via onServers should never be used to write into a replicate region for this reason.
3. A partition region can be used to distribute logic execution on many servers, and the partition region doesn’t even need to have any data.
4. Writes to a partition region always go the primary if a Function is used with the optimizeForWrite option. This is useful for counters and accumulators that would normally require a distributed lock. Since every call for the same key goes to the same node a local lock can be used instead. The region may have very little data.
5. A smaller heap size is generally desirable since it lessens the workload on the garbage collector. This is especially true if the write volume is high, or there is significant amount of data subject to expiration. In some situations it may be better to use a partition region for the reference data, even at the expense of some addition network traffic.

# Partition Regions and Co-location

One of most powerful features of Gemfire is the ability to force the same bucket of multiple regions into the same member server. This allows operations on many related objects to occur all in memory in the same JVM. This feature can be used to make function calls that act on significant number of objects never leave the JVM or require network calls to retrieve data.

For this to work 2 things must happen:

1. The Gemfire configuration must specify that a region is co-located with another region
2. The keys of the two regions that participate in co-location must have some value in common

The first requirement is met simply by specifying co-location in the configuration. For regions A and B, where B is co-located with A, the configuration for B includes --colocated-with=A (if GFSH is used to create the region). This tells Gemfire that bucket 1 for B is in the same JVM as bucket 1 for A. Without this Gemfire is free to place buckets anywhere, so bucket 1 for A is probably in a different server than bucket 1 for B. A typical scenario is Account and Order. Ideally the Account and Orders associated with an Account would be in the same JVM.

The second requirement is related to how Gemfire determines which bucket to place an object on put and where to find it on get or remove. This is determined by the PartitionResolver plugin. If no PartitionResolver plugin is configured the default is to use the key as the “routing object”. The calculation is as follows:

bucketNumber = Math.abs(routingObject.hashCode % numberOfBuckets)

If the default PartitionResolver is used for A and B, then both A and B must have the same key values. This is usually not desirable or even possible. For the Account/Order example the Account and Order will almost certainly have different key values (especially if there are more than one Order for a particular Account).

One commonly used solution is to place the key for A in the key for B, then write a custom PartitionResolver implementation for B that returns the key for A from within the key for B. In other words, AKey has an idA field and BKey has both idA and idB fields. The hashCode and equals methods in AKey uses the idA field, and the hashCode and equals methods in BKey use the idB field. The PartitionResolver implementation for B just returns the idA field from BKey.

The difficulty with this technique is that only the identity of B is known – a REST api would only pass idB since it would not know idA (e.g., only the Order ID is known, not the Account ID, since that is what the customer would select or enter). There are 2 options to circumvent this:

1. Include both idA and idB in the api. This may be cumbersome in some situations.
2. Use a single string field that is formatted for the identity of B, such as <idB>:<idA>. BKey would just have one field called id that would contain this string, and the custom PartitionResolver would parse the string to extract idA, which is the routing object.

The gemfire-util component provides a class called EncodedStringPartitionResolver that provides a generic solution to option 2.

The PartitionResolver is used for put, get, and remove. Note that the PartitionResolver has access to both the key and the value, but the value is null for get and remove. Therefore, it is not possible to use the value or a field in the value for the routing object. The value is deprecated in the PartitionResolver.

# Local Regions

To some degree local regions in Gemfire don’t seem particularly important, but they have many uses. Some examples include:

1. Store xrefs as described above.
2. Store generated objects that cannot be serialized and distributed. For example, a local region can contain a generated ML algorithm. A CacheListener on the region that is the source of the data is used to maintain the generated objects in the local region. An example of this technique is included in the Gemfire Examples.
3. Hold temporary workflow or work-in-progress objects. A local region would hold the temporary data in conjunction with a partition region that holds the non-temporary data. Since writes to partition regions always go to the primary the temporary data will reside in the same node as the primary bucket in the partition region.
4. The next value in an ID sequence can be stored in a local region that is persistent, as described above.

# Server-side Code Deployment

## Traditional Strategies

The traditional techniques for deploying code in a Gemfire server node are:

1. Deploy JARs via GFSH.
2. Include code in JARs on the Gemfire classpath and reference the classes in cache.xml.
3. Embed the Gemfire server inside a WAR using Spring and deploy the WAR to Tomcat.
4. Embed the Gemfire server inside a Spring application, especially Spring Boot.

All of these techniques have advantages and disadvantages.

### Deploy via GFSH

This technique allows code to be deployed to a running server without restarting it, which is especially important if the node is managing a large amount of data. The disadvantages of this technique are:

1. Deploying code is coarse-grained. An entire JAR must be deployed each time.
2. The object lifecycle provided by Gemfire is relatively simplistic and doesn’t support dependency injection or other advanced lifecycle features.
3. The dependencies between code and region definitions are a bit tricky to manage. For example, a new CacheListener must be uploaded via a JAR before the region definition can be changed. If a CacheListener is removed the region definition must be changed before the JAR without the CacheListener is uploaded.

The advantage of this technique is that it is built into Gemfire and is fully supported by Pivotal.

When using the Gemfire Tile from Pivotal Cloud Foundry (PCF) this is the preferred technique. The WAR and Spring techniques cannot be used, and it is difficult to deploy a JAR to the classpath on a server when using the Tile.

If the server-side code has dependencies, they must be deployed prior to deploying the code. If the dependent JARs have dependencies within them they must be deployed in the correct order.

Note that deployed JARs must not contain the version in the JAR name. When deploying a new version of a JAR Gemfire will think that the JAR is a new JAR rather than an updated version of an existing JAR. The version must be removed from the JAR name. This can be avoided by undeploying the old version before deploying the new version, but this will affect any code running in the server between the time the old JAR is undeployed and the new JAR is deployed. Dependencies between JARs may also restrict undeploying a JAR, so all the JARs may need to be redeployed.

Gemfire scans each JAR for functions, so every class is loaded into the classloader. This is different than the way the JDK works, where only the classes that are used are loaded. Therefore large numbers of dependencies and JARs with large numbers of classes will tend to load slowly and consume significant off-heap memory. Care should be taken to only deploy the JARs that are absolutely necessary.

Gemfire will only load classes found in the JAR files – resources, such as properties files are ignored. The JAR itself is not added to the classpath. Configuration and other resources cannot be deployed via GFSH.

JARs that are provided with Gemfire and are already on the classpath should never be deployed using GFSH. This includes the Gemfire JAR itself and Log4J2. There are some libraries that are embedded in the Gemfire JAR which are specified in the Maven POM for Gemfire. The easiest way to be sure which JARs should not be deployed is to use Maven and examine the Gemfire transitive dependencies. Any transitive dependencies should be excluded from the JARs that are deployed via GFSH.

An easy way to collect all the dependencies in one place is to use the Maven Dependency Plugin (<https://maven.apache.org/plugins/maven-dependency-plugin/>). The dependency:copy-dependencies phase copies all dependencies in the target/dependency directory. Then the dependencies can be deployed to the server using gfsh:

deploy --dir=target/dependency

By mapping the dependency:copy-dependencies phase to the package goal this step can be automated with the build. Any dependencies can be skipped by setting those dependencies to provided and adding <excludeScope>provided</excludeScope> to the plugin configuration in the POM. Note also that the version number in the JAR names may need to be removed.

One caveat with deploying CacheLoader, CacheListener, Function, and other plugins that use the Declarable.init method for initialization: there currently is no means of supplying the properties to pass to this method. In the cache.xml they can be provided via <property> elements, but this is not possible via GFSH. If the properties are global they can be provided via System.getProperty with -D<name>=<value>. If more than one instance of the plugin must be used, the easiest technique is to make the plugin abstract, pass the properties via the constructor, and extend the class for each instance that must be configured.

### Code on Gemfire Classpath

Managing dependencies between code and region definitions is easy, but the server must be restarted each time the code is updated. Rolling restarts can be carefully used in some situations, but any data managed by Gemfire must either be reloaded from disk or replicated from other nodes. Rebalance must be used for partition regions.

A hybrid approach is to add commonly used and rarely changing dependencies, such as Spring, to the classpath and use GFSH to deploy application code. Dependency JARs must be directly uploaded to the servers.

### Embedded in WAR

This technique allows deployments to be uploaded while Tomcat is still running, but since a new ClassLoader is created for each new deployment a new instance of Gemfire Cache is created each time. The data must be reloaded as if the server was restarted, and existing data must be garbage collected.

### Embedded in Spring

The considerable power of Spring is available to the application code, especially dependency injection. All of the Spring modules can be utilized. The disadvantage is that the servers must be restarted with all the problems of data reloading, replication, and rebalancing.

If the code is complex and has many components that must be wired together, and the amount of data is relatively small, this is a good technique. When Gemfire is used as a logic execution platform, as opposed to simple caching, this is an effective technique.

## Other Strategies

Several other options are available for deploying server-side code. These options provide an alternative to the traditional options and have a number of benefits.

When writing code for the server, try to design the code such that it can be used either via a gfsh deploy or embedding via a WAR or Spring. One of the most important aspects is how to gain access to a Region in a CacheListener, CacheLoader, Function, etc. In some cases the Region is provided by the callback methods in the interface (e.g., in the CacheLoader.load(LoaderHelper) method the LoaderHelper class provides access to the Region that triggered the call to the CacheLoader). Where the Region is not provided by the plugin interface it must be obtained some other way. If the plugin implementation is created by Spring the Region may be injected via a setter method. If the plugin implementation is configured via gfsh

### Rules Engine

Since the code running in the server often is implementing business rules, a rules engine is a natural choice for executing business logic. Most rule engines allow rules to be dynamically modified, and some even provide repositories to store the rules. They also can have Java code embedded inside the rule, so actual code can be deployed to server. Rule engines commonly compile the rule to Java so the execution is almost as fast as plain code.

Gemfire Examples include a component that demonstrates the use of Drools (http://drools.org/) inside of a Gemfire server node.

### Scripting

Scripting engines, such as Groovy (http://www.groovy-lang.org), JRuby, and Jython, allow for dynamic modification of scripts. Groovy is a Java dialect, so the learning curve for Java developers is relatively short. Groovy provides an API, GroovyScriptEngine (http://www.groovy-lang.org/integrating.html), which is specifically designed for embedded scripting.

Gemfire Examples includes a component that demonstrates the use of Groovy scripts inside Gemfire.

## Debugging Server-side Code

This topic describes a technique for Eclipse, but it should be similar for other tools.

Server-side code in Gemfire can easily be debugged since Gemfire is written in Java and can be started via a standard debug configuration in Eclipse. When the gfsh start server command is executed the script eventually calls a main method in the com.gemstone.gemfire.distributed.ServerLauncher class. An Eclipse run/debug configuration can be created that uses this class.

The configuration is created as follows:

1. In the Java Application section, select New
2. Enter a name for the configuration
3. Ensure the correct project is selected and the product includes Gemfire as a dependency
4. Enter or search for com.gemstone.gemfire.distributed.ServerLauncher in the Main class field
5. In the Arguments tab enter start server in the Program arguments field
6. In the VM arguments field provide the locator information via –Dgemfire.locators=<locator> (note: any Gemfire property may be added this way)
7. If GFSH will be used to run commands on the server, add the gfsh-dependencies.jar file to the classpath in the Classpath tab

This configuration can be used to either run or debug the server.

One caveat is that functions on the classpath will not be automatically registered like they are when they are in a JAR that is deployed with GFSH. The easiest solution is to create a cache.xml file and configure Gemfire to use it via –Dgemfire.cache-xml-file=path/to/cache.xml. Note that it should NOT go in src/main/resources since it will end up in the JAR file, which will eventually get deployed to the server. Using a cache.xml is also easier than GFSH for development since the GFSH commands don’t have to be executed to configure the server. Note that the cache.xml must contain the <cache-server> element if a test is run via the ClientCache.

Another option is to start the server directly using the Java API. Starting a Gemfire server is as simple as:

Cache cache = new CacheFactory().create();

Configuration via the Java API, cache.xml, or GFSH will work when starting Gemfire this way.

# Logging

Gemfire now uses Log4J2 for logging. That’s the good news. Unfortunately the default configuration leaves a bit to be desired. The normal logging configuration is to print the class name as part of the output, and the default log level is INFO. The default Log4J2 configuration file is embedded in the Gemfire JAR and runs counter to these conventions.

For server-side code there are a couple of ways to get around this problem:

## Custom Log4J2 Configuration

Create your own Log4J2 configuration file and explicitly specify its location with the –Dlog4j.configurationFile=/path/to/log4j2.xml system property. Care must be taken to ensure that any logger in the com.gemstone.gemfire package is logging at WARN since the output is extensive at INFO.

## Use Slf4J/Logback for Application Code

Use Slf4J api and the Logback implementation, with a separate logback.xml configuration file that uses a file logger and writes to a separate log file. Note that there is a Log4J2 bridge component, log4j-slf4j-impl. This allows the code to use the Slf4J api but still use Log4J2 as the implementation. This provides some flexibility since the logging implementation can change without changing any of the application code.

Unfortunately neither solution is optimal since configuration files cannot be deployed in JARs by GFSH. They must be copied directly to the server and included in the classpath.

# GFSH

GFSH is the preferred way to start locators and servers, configure Gemfire, deploy JARs and manage the running cluster.

## GFSH Run Scripts

Generally GFSH commands that affect the cluster configuration should not be typed directly into GFSH on a production cluster. A better technique is to put the commands into run scripts that are checked into source code control. The run scripts contain the same commands that would be entered directly into GFSH.

For most commands a connection to a locator is required, so the first command is connect. For example:

connect --locator=mylocator[10123]

create region --name=myregion --type=REPLICATE

If these commands are in a file called create-region.txt, the file can be run via:

gfsh run --file=create-region.txt

Note that the preceding line is not interactive, so it could be run as part of a build, via a cron job, or any other facility that can execute shell commands.

## Configuring PDX in the Server

One caveat of configuring PDX on the server using GFSH is that the configuration change only affects servers started after PDX was configured. The effect is that the servers must be restarted after making a PDX configuration change. In almost all cases PDX must be persistent and there should be a separate disk store for PDX, so the initial setup of a cluster would generally be:

1. Create a disk store called “pdx” using the create disk-store command
2. Configure PDX using the configure pdx command
3. Restart the servers (the locators do not have to be restarted)

Note that if the PDX configuration must be changed all of the options must be specified on the command, not just the one that changed. Once PDX is configured for a disk store and persistence, it must not be configured without persistence later on. Generally the only thing that should change is the --auto-serializable-classes option. The best way to minimize the chance of incorrectly configuring PDX is to use a run script that is checked into source code control, rather than typing commands directly into GFSH.

Note that GFSH should not be used to shutdown the servers when using the Gemfire Tile in PCF. PCF provides commands to start, stop, and restart the Gemfire cluster.

## Command Order

Care must be taken when ordering commands. If a command creates a region that uses a disk store, the disk store must be created first. If a command creates or alters a region so that a CacheLoader or CacheListener is configured on the region, the code must be deployed to the server before the region can be created or altered.

## Disk Store Creation

Note that there’s an issue with disk store creation in GFSH. The disk store is not fully created on the cluster when the command returns in GFSH. If the following command uses the disk store an error may be returned saying the disk store doesn’t exist. A sleep command can be used to allow time for the disk store to be available prior to referencing it.

# Serialization

Java serialization should not be used on any class that is stored in a Gemfire region, even if PDX is configured. If the PDX configuration has a typo in it, or if a package or class is missing from the configuration, Java serialization will silently be used. If the data is stored on disk it can be orphaned if the PDX configuration is fixed or if the class is modified.

# PDX

If no code is written for the server, PDX with read-serialized=true is the best option since the PdxInstance on the server is never deserialized. Also, if domain classes are used on the client they do not have to be deployed to the server.

It also possible to use PdxInstance directly on both the client and the server, in which read-serialized can be true on both the client and the server. The code must be written to access fields via name, which can be somewhat cumbersome and error prone if not done carefully.

If PdxInstance is used on both the client and the server the key can be PdxInstance rather than String, Integer, or Long, since the problem with hashCode calculation (described in the Keys section above) is not a factor since it is the same everywhere.

The JSON features of Gemfire can be very useful for REST-based micro-services since the PdxInstance can be converted to and from a JSON string very easily (using the JSONFormatter helper class in Gemfire). Gemfire will use a special PDX type for a PdxInstance converted from JSON. However, if domain classes are also used to access the same data it is not possible to use JSONFormatter since the PDX type is not the domain class. The gemfire-util component includes a solution, in the form of the JSONConverter. It allows a fully qualified class name to be specified when converting from JSON to PDX.

# Client-side Caching

Client-side caching in Gemfire seems like a very useful feature, since it reduces the trips to the server, but it has a number of disadvantages. It is useful but it must be utilized carefully. Some of the issues are:

1. Garbage collection and heap issues that are common for the server nodes must also be considered for the client nodes.
2. If the client is deployed in a cloud app, such as in CloudFoundry, the amount of heap is fairly limited.
3. Data can get very stale if subscriptions are not used.
4. If subscriptions are used the connection from the client to the server can get very chatty, especially if there are a large number of clients.