# Skype recording

The following Skype recording is a high level walkthrough of the caching system as of 2016-7-13. It doesn’t start from the very beginning, but should still be useful:

[https://activenetwork.box.com/s/vx86gglsmq6xscmtd2jd975jtz9329ko](https://owawest.activenetwork.com/owa/redir.aspx?SURL=pH386TvCCvoFNSTqe5CioI3yrLdAEYlextIEC835arBZFiVeQqvTCGgAdAB0AHAAcwA6AC8ALwBhAGMAdABpAHYAZQBuAGUAdAB3AG8AcgBrAC4AYgBvAHgALgBjAG8AbQAvAHMALwB2AHgAOAA2AGcAZwBsAHMAbQBxADYAeABzAGMAbQB0AGQAMgBqAGQAOQA3ADUAagB0AHoAOQAzADIAOQBrAG8A&URL=https%3a%2f%2factivenetwork.box.com%2fs%2fvx86gglsmq6xscmtd2jd975jtz9329ko)

# Initial investigation

## Overview

From the very beginning, ActiveNet (ANet) was designed with a very aggressive home-grown caching system for database data:

* **Full table caches of “administrative” tables:** Roughly, the types of data Anet works with can be considered either administrative or transactional. Transactional tables including the population system (customers, companies, etc), and all associated financial and inventory data; these are not basically cached at all. Administrative tables configure the operation of the system, and are comparatively small (although a large org might have tens of thousands of activities or facilities). At startup, the class associated with an administrative table reads the entire table, creating objects for each record, in some type of cache structure (ArrayList, HashMap, etc.), so the full table is available. There are some variations of this model to reduce the amount of data cached for some tables, but this should be considered the baseline design. The code relies extensively on the fact it can enumerate through this data in memory for many purposes, such as displaying select boxes, lists of records for editing, and some in-memory searching. There are approximately 240 such classes.
* **Cache synchronization between servers:** For redundancy, each org has at least two servlet instances on different servers. In order to keep the caches on these servers in sync as administrative tables are edited, ANet has a simple home-grown cache coherence system. There is a separate application called CacheControl, which is a very simple messaging application. On each insert, update or delete into an admin table, the base classes of the administrative table system send a message with a serialized Java object to all other servers for that org registered with CacheControl; upon receiving those messages, they update their copy of the cache. A simple softlock system (the System\_Locks table) is used to prevent concurrent updates of admin objects.

## Problems with current design

* **Memory-bound architecture**: This cache design requires significant memory. Memory has been the limiting factor in how many servlets could run in the current single-tenant (ST) architecture; ANet app servers are memory-bound, not CPU-bound. The multi-tenant (MT) servlet now under development should allow significantly more sites to run on a server, but it will still be necessary to assign specific sites to specific servers, and the system is still expected to have plenty of CPU headroom. Reducing the memory footprint of caching will allow us to use servers more efficiently. The ideal would be if any server can take requests from any site, so the entire set of servers can serve as a “rush pool” for peak loads/
* **Duplication of cache data:** Each server running an org has the entire cache present in its memory. The more servers assigned to a site, the more memory impact
* **Startup time:** When a servlet starts up, before it can process any request, it loads its full cache. For large orgs, this can take a few minutes or more. This causes additional downtime if servers need to be restarted. It also significantly effects deployment time, because thousands of servlets are loading their caches during this time.
* **Cache coherence reliability:** The CacheControl service is a simple home-grown design, and has had reliability problems from time to time. This is one reason each ANet servlet has a scheduled daily reload of the cache.

## General approach

* Move the cache outboard into a separate third-party caching service. This should:
  + Reduce the memory footprint of the ANet servers
  + Eliminate duplication of the cache, except to the degree the cache has internal redundancy.
  + Eliminate the need for the CacheControl system.
* Coding model which reduces code changes in the implementing classes as much as possible; ideally, hiding it within the OrgContext.getCacheWhatever() methods (e.g., OrgContext.getActivities() ), to reduce coding time and increase reliability of conversion.
* Coding model must allow classes using the new caching model to co-exist with older classes, so this conversion can be done incrementally over a number of sprints.
* Focus on classes with highest memory footprint first.

## Caching models

* **Full table caching:** As described above, all records are cached in objects of the corresponding class. This is the most common case. However, there are a few modifications to this, to reduce memory and/or make a better access method.
* **Full table caching but only of active data:** This is implemented in the Activity class (not sure of others yet). Activity objects are fairly large, and are a large organization, there could be tens of thousands of activities. However, since typically new activities are created each season, and the old ones retired, no longer available for enrollment, what we do is only cache the active activities. However, users can do special searches to include retired activities. In this case, they are added to the cache; the daily cache reload throws them away again.
* **On-demand caching of child records:** A very common case is child records of a parent. For example, an Activity has one ActivityDate for each session of that activity. However, ActivityDates searches are never through the full set; we only need the ActivityDates for a specific activity at a time. So from a data access model, the activity dates for each activity are stored in a List; those lists are stored in a Hashtable keyed by the activity primary key, so we have this: Hashtable<Integer, List<ActivityDate>>. We don’t actually need the ActivityDates until certain processing is done, so the code only creates the List on first access, and adds it to the HashTable. Again, the daily reload throws these away, and we start afresh. (This might be better implemented as a SoftReference).

## Cache implementations

There are several collection types used for cache implementations”

|  |  |
| --- | --- |
| Vector Vector<AgeCategoryBO> | Original model, still used in many classes. In most cases the Vector is sorted to match typical display sequence. Any lookup, even by primary key, requires a linear search. |
| ArrayList<ActivityCategory> | Newer implementations of above.  At least some of these, like ActivityCategory, have an associated ConcurrentHashMap, which maps from PK to object. |
| Hashtable<Integer, Activity> | No sorting; provides fast access by primary key (Anet’s find(id) method). |
| HashMap<Integer, ArrayList<FiscalPeriod> Hashtable<Integer, List<ActivityDate>> | Access to all the child records for a parent record by parent record PK |

# POC Implementation

## Caching service implementation

### Design aims

* Significantly reduce the average memory of an org’s data in the servlet
* Require minimal changes to the existing classes; no structural rewrite required on initial implementation.

### Two types of caches

There are two different caching models in ActiveNet:

* **“Simple cache”:** This is a single collection of records from the table, implemented as a Vector, ArrayList, HashTable or HashMap.
* **“Cache of lists”:** This is used for collections of child records, like the ActivityDates for an Activity. We don’t have a single large cache of all ActivityDates; that isn’t the way they’re used. Instead, we have a hash on activity\_id, which returns a list of ActivityDates. In this case, it’s implemented as Hashtable<Integer, ArrayList<ActivityDate>>.

There are differences in the design solution, so they are treated separately below.

## “Simple cache” design concept

This is the design concept as currently implemented:

### Redis

* The persistent cache will be in Redis, not in individual servlets.
* To match the current application model, the Redis cache for a table will be preloaded with the same records we’re currently caching.
* Only the first servlet requiring a given cache will have to load the Redis cache.
* The values in Redis will be serialized Java objects.

### Servlet

* There is a new class RemoteCache<BO>, which implements the same interfaces (Map and Iterable) as the current code uses for Vectors, ArrayLists, Hashmaps, etc., allowing change to the caching object with minimal change.
* Minor changes to AdministrativeTable and BusinessObject changeCache and removeFromCache methods:
  + They update the RemoteCache, just as they do with other cache types.
  + They don’t use CacheNotification for RemoteCache caches.
* Caches are changed from their current datatype to RemoteCache<BO>, and necessary (generally minor) changes are made.

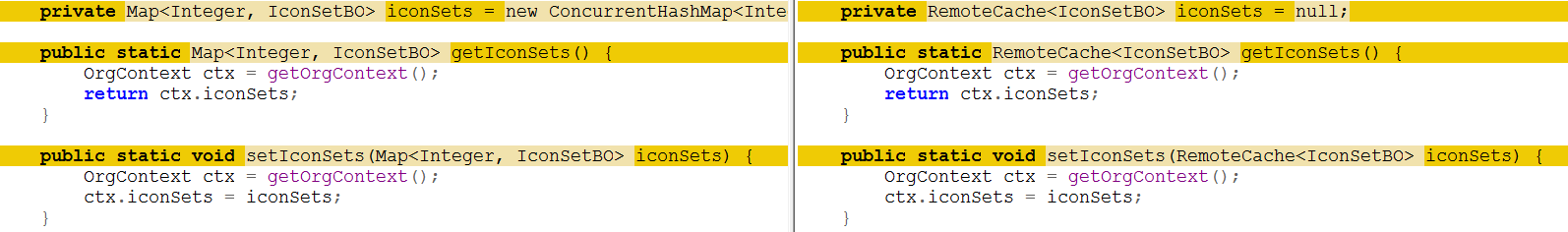
## Code modification example

A small class of each existing simple cache type was selected for proof-of-concept modifications:

|  |  |  |
| --- | --- | --- |
| **Class** | **Cache type** | **Admin interface** |
| ActivityUsers | Vector | AdministrativeTable |
| Cities | ArrayList | AdministrativeTable |
| CouponBatchBO | HashTable | BusinessObject |
| IconSetBO | ConcurrentHashMap | BusinessObject |

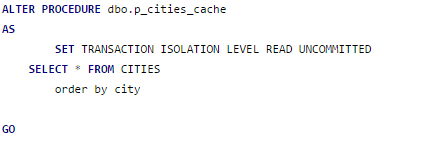
Going through these samples, here are the types of changes required. They are very localized, limited in number, and easy to code-review.

### Change OrgContext type



### Create proc for query to fill cache

* Standard name is p\_<table>\_cache
* Should have ORDER BY to minimize sorting cost.



### Add import for RemoteCache

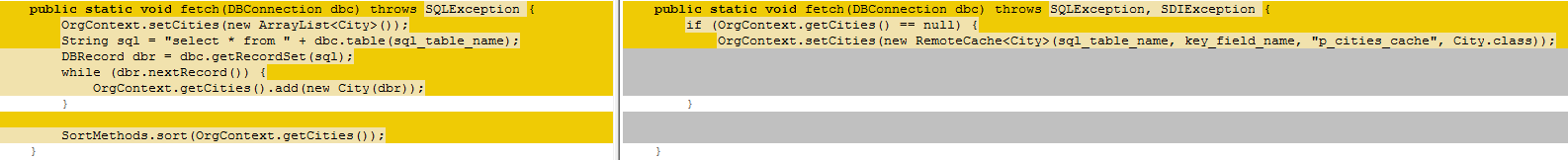


### Change DBRecord constructor to public (if private)

An exception will be thrown during init if this is not done.



### Change fetch to use RemoteCache constructor, and throw SDIException



### Change types in getCache, setCache and getFullCache methods

Compiler will catch this.



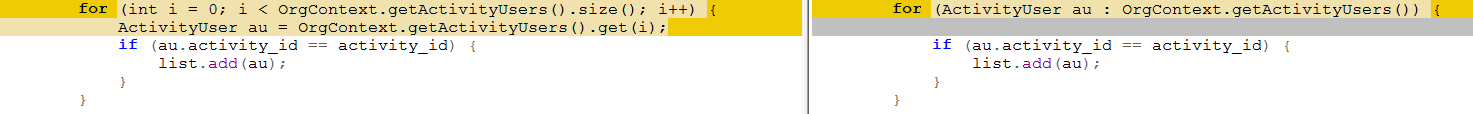
### Change toHtmlSelect

Compiler will catch this. (This could be eliminated if HTML.selectStatement took a RemoteCache, but it’s in ActiveNetPackages…)



### Change any for loops to use Iterator

The above changes are single point changes in a class. This one is likely in classes which had a Vector implementation, and may occur multiple times. The compiler won’t catch this, but the size() method will throw an SDIException.



## RemoteCacheProvider interface

The system has been designed to be able to be run without Redis, as a convenience to developers doing conversion:

* The RemoteCache<BO> class accesses the actual class via the RemoveCacheProvider<BO> interface.
* RemoveCacheProviderLocal<BO>: Has a local Hashtable to simulate a Redis Hash.
* RemoveCacheProviderRedis<BO> implements this interface, and gets its data from Redis.

## Redis implementation

### Redis data type

Redis supports several different types of data. See <http://redis.io/topics/data-types>

For ActiveNet’s cache, the requirements are:

* Map interface: Random access by primary key (get, put, remove operations)
* Iterable interface: Ability to get all the objects for a given cache.

For just the Map interface, we could have used a simple String object, keyed by <org>:<table>:<pkey>. It is possible for enumerate just the entries for <org>:<table>, However, this would not allow for efficient enumeration.

The List, Set and SortedSet types can return the full collection, but don’t support random access.

For ActiveNet, we’re using the Hash type. The key is <org>:<table>; within the hash, the “field” is <pkey>; this supports the Map interface. Hashes provide the ability to get the complete collection, so we can support the Iterable interface.

For Hash capabilities, see <http://redis.io/commands#hash>.

### Java interface

The Jedis library is used to access Redis.

## “Cache of lists” design concept

This is the design concept as currently implemented.

### Redis

* The same RemoteCacheProviders are used. However, the cache key is for the “pseudo-table” <org>:<table>:<parent\_id>.

### Servlet

* There is a new class RemoteCacheList<BO>, which implements the same interfaces (Map) as the current code uses for Hashes of Lists.
* RemoteCacheList keeps an internal hash list by parent\_id. Each entry has a SoftReference to an ArrayList<BO>, which is maintained with the same rules as the simple cache.
* The get method initialized the cache for the parent\_id, if necessary, updates the local cache, and returns an ArrayList<BO>.
* The put method takes an ArrayList<BO>, clears the entire remote cache, and rewrites it, also updates the local cache.
* Each entry maintains a creation time, so the same discard on stale can be implemented.
* There is no Iterable interface (so far), since we don’t seem to use it. However, there is a the values() method of the Map interface is implemented, and used by the code for similar purposes.

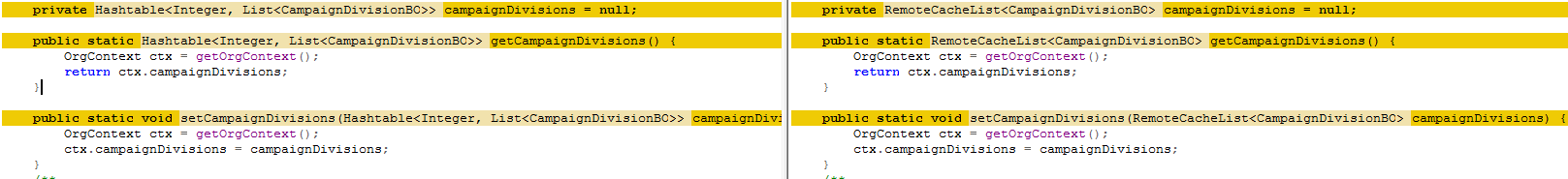
## Code modification example

A small class of each existing simple cache type was selected for proof-of-concept modifications:

|  |  |  |
| --- | --- | --- |
| **Class** | **Cache type** | **Admin interface** |
| CampaignDivision | Hashtable<Integer, ArrayList< CampaignDivision >>. | BusinessObject |

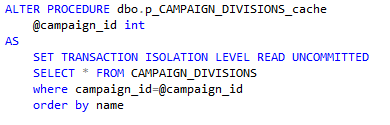
Here are the types of changes required. They are very localized, limited in number, and easy to code-review.

### Change OrgContext type



### Create proc for query to fill cache

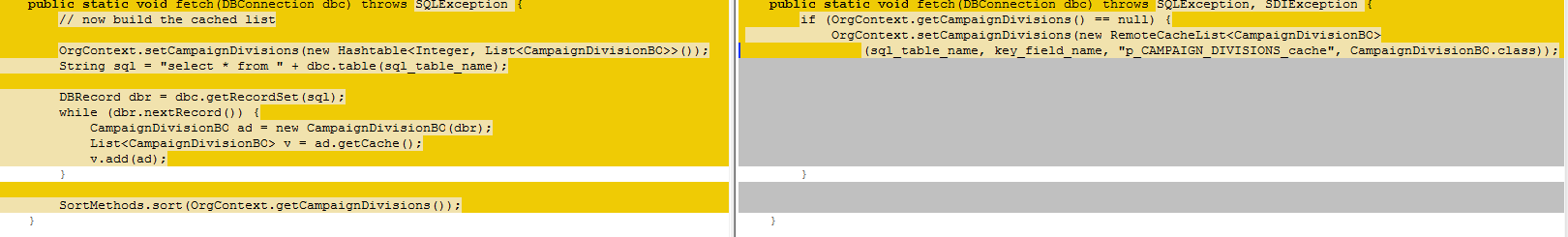
* Standard name is p\_<table>\_cache
* Must take parent\_id as a parameter
* Should have ORDER BY to minimize sorting cost.



### Add import for RemoteCacheList

### Change DBRecord constructor to public (if private)

### Change fetch to use RemoteCache constructor, and throw SDIException



### Change types in getCache, setCache and getFullCache methods

# Redis configuration

The configuration settings for the Redis cache are stored in two places:

* ActiveNetSites SystemInfo table, which gives the default values for the entire data center.
* The orgsite’s SystemInfo table, which optionally gives overrides for that specific orgsite, primarily for testing purposes.

The settings are as follows:

|  |  |  |
| --- | --- | --- |
| **SystemInfo key in ActiveNetSites** | **SystemInfo key in Org DB** | **Meaning** |
| Redis\_server | Redis\_org\_server | URL or IP of Redis server. If unspecified or “local”, Redis will be emulated by the RemoteCacheProviderLocal class, to support development without a Redis server. |
| Redis\_port | Redis\_org\_port | (Optional) Port number which Redis is listening on. If blank or 0, then the Redis default of 6379 is used |
| Redis\_password | Redis\_org\_password | (Optional) If specified, the password required for the Redis connection. |
| Redis\_timeout | Redis\_org\_timeout | Maximum time to wait for a reply from Redis (ms) |
| Redis\_environment\_prefix | Redis\_org\_environment\_prefix | A prefix which is appended onto each key, to specify the environment, e.g., “Int”, “Stage” or “Prod”. To allow multiple developers to use the same Redis instance, it should be configured as the developer’s username, i.e., “dchristie”. |
| Local\_cache\_expiry\_secs | Local\_org\_cache\_expiry\_secs | Number of seconds to keep a local cache before it’s considered “stale” and will be generated on next usage. |
| redis\_cache\_load\_timeout\_secs | redis\_org\_cache\_load\_timeout\_secs | Maximum number of seconds a cache can be in the “loading” state before another process can decide the load must have hung and will take over. |

# Redis keys

The following key patterns are used by the ActiveNet Redis implementation:

|  |  |  |
| --- | --- | --- |
| **Key** | **Hash field meaning** | **Hash contents** |
| <env>:<site>:<table> | Primary key of table | For simple cache, serialized object for table / primary key |
| <env>:<site>:<table>:<parent\_id> | Primary key of table | For cache of lists, serialized object for table / parent key / primary key |
| <env>:<site>:!cache\_status | <table> | No field = Cache not loaded “Loading@YYYY-MM-DD hh:mm:ss”: Load in progress <version>: Loaded with data from that version |
| <env>:<site>:!cache\_status | <table>:<parent\_id> |  |

Where:

|  |  |
| --- | --- |
| <env> | Redis\_environment\_prefix (e.g, “prod”, “int”, or developer username). Allows a Redis server to be shared in a development environment. |
| <site> | Orgsite name (e.g., “chicagoparkdistrict”) |
| <table> | Table name being cached |
| <parent\_id> | For caches of lists, the parent\_id in <table> |
| <version> | ActiveNet version of the serialized data (e.g., “V16.22.0.0.099”). Assumed to start with “V”. |

# RemoteCache<BO> operation

The following diagram shows the basic life-cycle of a RemoteCache<BO> used as a cache.



## Initialization

1. In the AdministrativeObject or Business object (collectively referred to as “BO”s below), the fetch method is fired during initialization or reload. This creates a new RemoteCache<BO> object in the OrgContext for the BO.

2. The RemoteCache constructor in turn creates a RemoteCacheProviderRedis<BO> (the “provider”), and gets a JedisPool for threadsafe access to Redis through the Jedis library. Normally there is only one Redis server in a data center, hence only one pool, but for testing purposes, a site may have a different Redis server than the data center default, hence there may be more than one JedisPool.

3. The provider’s needsInitialization() method is fired to determine whether the corresponding Redis cache has been loaded with a set of serialized Java objects. It accesses a special hash <env>:<org>:!cache\_status to determine the status of that cache.

If the cache is uninitialized, or initialized for a previous ANet version, the provider uses the !cache\_status hash to acquire a “fetch lock”, preventing any other servlet from simultaneously initializing the same cache.

4. RemoteCache uses the proc provided in the constructor to query all the records which should be in the cache.

5. RemoteCache calls the provider to delete the old cache, if any, then write new values to it.

6. Because Java serialization is too slow, a product called Kryo is used for serialization/deserialization of the BOs.

## Record level access (put, get, remove)

7. Without any code modification, the existing access to the cache works with RemoteCache, because it implements the Map interface. These get passed directly through the provider to Redis, being serialized/deserialized by Kryo.

## Iteration

For many purposes, the current code also iterates through the cache. For example, iteration is used to populate select lists, display lists of records in the admin interface, for specialized searches such as Activity search, and other purposes.

8. RemoteCache implements the Iterable interface. This means any construct like this will work:

For (Activity a : OrgContext.getActivities()) {

9. RemoteCache could enumerate directly through the provider, but this would be a very bad solution under load. E.g., if a site had 5000 activities, each activity search request would read from Redis and deserialize 5000 activities, which would then be garbage-collected. Instead, RemoteCache creates a local cache (an ArrayList) for the results, and passes back its iterator. Subsequent iteration requests will use the local cache. The local cache is kept in a SoftReference, so the garbage collector may free it under memory pressure. There is also a configurable time-to-live for the local cache, so it is discarded and recreated if it is “stale”.

10. If there is a local cache, when put and remove methods update the remote cache, they also update the local cache.

# RemoteCacheList<BO>

A RemoteCacheList differs from a RemoteCache, because what is being cached are ArrayLists of child objects of a parent\_id, not individual objects. There is currently no Iterable interface, and the Map interface put and get methods are passed ArrayList<BO>s. Much of the operation is similar to a RemoteCache, so this description only highlights the differences:



## Initialization

1. The RemoteCacheList has an inner class SubCache. There is one SubCache instance for each parent\_id, accessed via a hash. The SubCache may contain a local cache, which is an ArrayList of the child records for that parent.

2. The Redis cache is similar, but where the for a RemoteCache has <table> as the last token of the key, a RemoteCacheList has an additional token <parent\_id>. So in a typical case, for the test class CampaignDivisionBO, the hash for the records with parent\_id (campaign\_id)=3 might be

dchristie:linux01:campaigndivision:3

<env> :<site> :<table> :<parent\_id>

3. The BO classes’ fetch method first initializes the RemoteCacheList, but this doesn’t cause any of the SubCaches to be created. Then it forces all the SubCaches to be loaded, and the Redis remote cache to be initialized, if not already done. So typical fetch code looks like this:

public static void fetch(DBConnection dbc) throws SQLException, SDIException {

RemoteCacheList<CampaignDivisionBO> cache =

new RemoteCacheList<CampaignDivisionBO>(

sql\_table\_name, key\_field\_name, "p\_CAMPAIGN\_DIVISIONS\_cache",

CampaignDivisionBO.class, cacheSerializer.kryo);

OrgContext.setCampaignDivisions(cache);

// Force all subcaches to be initially loaded

for (Campaign c : OrgContext.getCampaigns()) {

cache.initializeRemoteCacheList(dbc, c.id());

}

}

## Updates

4. A get method causes the SubCache to return its local cache, if present, and otherwise, to call the provider to get the full set of objects and create its local cache, very similar to a RemoteCache iteration call.

5. A put method causes the SubCache to clear the remote cache and repopulate it from the ArrayList, very similar to what happens during the fetch process.

# Testing approach

## Test DB selection

* Select one or a few databases, which came from real orgs, which have large numbers of key records

## Confirm local cache release under memory pressure

This testing can be done initially by Dev, since it will use in-servlet tools. The idea is to confirm that the dynamic creation and release of the local cache memory does work as memory pressure is applied. Until we have a significant memory release, it doesn’t make sense to proceed with multi-site testing.

* Apply RemoteCache to a small number of large memory classes, so we can have a measurable improvement in initial memory usage.
* Build testing page to show status of each cache, and total memory
* Build test facility to force all RemoteCache’s to have local cache; confirm how much memory used
* Build memory stresser to force large memory usage, to force SoftReferences to release
* Test that memory returns to pre-load levels
* Quantify improvement of minimum memory

## Normal perf test

This will be expected to keep the cache

## Measure CPU load caused by local cache reloads

# Prior notes

# Caching service implementation

## Working assumptions about external cache service

* It will be a keyword-value store.
* To match the application model, the keyword will be some version of <org id>:<table identifier>:<key>
* The value will be a serialized Java object, like we use in cache synchronization now.
* The semantics of the key may vary by cache implementation; could primary key (for hashmap-style), value for SystemInfo type tables…
* The service must have basic get, put and delete functions by keyword.
* In order to support the ability to enumerate through a cache, the service must provide the able to get <org id>:<table identifier>:\*. (Ideally in a predictable order, alpha by key)
* Since the deployment of a new version of ANet means the previously stored serialized objects may be unusable, it should be possible to entirely clear an organization’s cache; i.e., to delete <org id>:\*
* Must be redundant and highly available; not LRU / throwing data away.

## Assumptions about implementation

* Classes using the new cache system will not need to use CacheControl.
* However, the CacheNotification class can be modified to put or delete entries in the cache service.
* **A major impact on the current code may be changing how admin objects reference to other admin objects, so we don’t recursively most lots of objects into memory.** For example, the ActivityClass instance data the ActivityTypeBO, which is set in the refreshCacheLinks method with a cache lookup (ActivityTypeBO.find), which is then accessed by getActivityType(). I haven’t determined how universal this child object resolution on demand pattern is. But If we store a SoftReference to the ActivityTypeBO, then getActivityType() can fetch it from the cache as needed, without forcing it to stay in memory, and the implementation is easier.

### Architectural model

A pretty simple architectural model of the code, which would fit the current application structure:

* Write our own replacements for Vector/ArrayList and HashMap, Arraylist; say CachedMap and CachedList, which implement the same interfaces (Collection, Iterable, Map).
* In the base design, rather than managing a local collection, these classes would make requests to the caching service for gets, puts and deletes.
* The two base classes (AdministrativeTable and BusinessObject) would need to recognize these cache implementations.
* CacheNotification would be modified to put and delete data to the cache service.

### Object references from cached objects

* When an object like Activity references related objects like its ActivityType, it will have an activitytype\_id (int), but we also might have an ActivityType object (which Activity does). In cases like this:
  + The ActivityType object should be transient, so it’s not serialized with the Activity.
  + Ideally, the ActivityType object should be created on demand.
* In fact, the Activity class seems to be good in this respect, but not all classes are. For example, the Facility class has an instance variable: Vector<FacilityOpeningTimeBO> facility\_opening\_times. Most importantly, it is not transient; it is also created when the object is created, even though not all usages of a Facility may require facility\_opening\_times.
* When converting a class to the new caching model, all object instance variables should be reviewed.
* When our classes do create object references, they do so in variety of ways. In some cases, it is on demand on the first use of a getter (preferred); in some cases, as part of a constructor, or immediately after object creation. We currently don’t use SoftReferences at all, but creation in a getter using a SoftReference would seem to be the ideal pattern.

### Modifications to existing classes

Then the modifications to an implementing class could be fairly limited, and readily code-reviewed:

* Changing the types of the cache object to the new Cached\*.
* In some cases, changing the iteration idiom, at least for current Vector implementations, to use Iterable (for-each).
* Reference review and modification, at least making .

## Performance issues and optimization for enumeration

* Single record operations like gets, puts and deletes should be reasonably efficient.
* However, to enumerate through all the objects in a table for any reason requires we get the entire set of objects across the wire, construct them, and then allow them to be garbage-collected. This will create a huge amount of churn. In a rush environment, multiple threads will be doing activity searches at the same time. A simple implementation would end up with multiple copies of the activity cache in memory.
* An optimization which could be implemented in our Cached\* classes would be to keep the cache local under load. For example, it could be kept in a SoftReference, and only have a hard reference from the iterator.
* Because the local cache copy when present may be stale, and not reflect configuration changes, we should implement a time-to-live for it, so even if it is not released by the SoftReference, if it’s too old we’ll reload it.
* As a further optimization, if a local cache did happen to be present, the get method could use it and eliminate a network call/object instantiation; this would also require the put and remove methods to update it in parallel with the remote cache.
* If we can get adequate performance with the above architecture-only solution, over time, we will want to use other strategies to eliminate or reduce cache enumeration. To support this, the Cached\* classes can provide logging for enumeration calls, and of whether the local cache fulfilled the request.

## Load cache for a class

* The startup code should detect what version of the application wrote to the current cache; if it is different than it’s own version, it must be able to delete the entire cache for the organization to force a full reload.
* Since there is one shared cache for a class, we don’t want each servlet resetting it on startup. Instead, the fetch() method should be able to determine if the cache is already loaded.
* We only want one servlet at a time initializing a cache, so some type of lock will need to be implemented, either in the caching service, or in System\_Locks.
* Probably that lock will need to be checked on all cache accesses, so it should be very efficient.

## Get object by ID

* Very straightforward to construct key from ID and get from cache.

## Enumerate through cache

* The simplest implementation is that Our Cached\* classes will provide an Iterator; when it is created, it would read the entire remote cache into a local cache, then return a single object on each .next() call.
* Vector and ArrayList cache implementations often are sorted. Enumeration is then in sort order. How will this be implemented?

## Insert, update or delete records

* The CacheNotification class should be able to be used to generate the put and delete calls to the cache service.

# Further notes:

## Session out-of-process

Improve reliability by eliminating requirement for F5 sticky sessions, by storing session data in cache: <org id>:SessionData:<session id>

## Basic cache operations

Reviewing the code for a few classes which implement caching, by looking for OrgContext references to get or set the cache used by that class. I believe these are the key functions which use the cache.

### Load cache for a class



* At servlet startup, the startup code calls the fetch() method of each caching class.
* The fetch method creates an empty cache object and sets it into the Orgcontext.
* The fetch method queries the database, creates an object for each record, and adds it to the cache.
* If there are N servers deployed with the site, each one will separately query the DB and build its own internal cache.

### Get object by ID



* Many classes have a static method find(int id), which returns an object from its primary key. In the Hashmap style cache implementations, this is a very simple get.
* For Vector / Arraylist style caches, the code implements find(id) by enumerating through the cache.

### Insert, update or delete records



* In order to be simple, the above sequence diagram isn’t exact in terms of what’s implemented in the base class versus its extension.
* For inserts and deletes, objects are added to or removed from the cache.
* Generally, the base class handles sending a change notification to other servlets via the CacheNotification class.

### Enumerate through cache



* Probably every cached class has some need to enumerate through its cache. Reasons include:
  + For non-Hashmap caches, implementing find(id) by looking at each object.
  + Populating a select box with a list of values.
  + Displaying the record list for editing.
  + In important cases like activity search, doing an in-memory search.
* Once the cache object is fetched from the OrgContext, the style of enumeration depends on the type of cache object (and when the code was written).