Article
Benjamin De Boe · Apr 3, 2017 11m read

Keeping your iKnow domain synchronized

If you've worked with iKnow domain definitions, you know they allow you to easily define multiple data locations iKnow needs to fetch its data from when building a domain. If you've worked with DeepSee cube definitions, you'll know how they tie your cube to a source table and allow you to not just build your cube, but also synchronize it, only updating the facts that actually changed since the last time you built or synced the cube. As iKnow also supports loading from non-table data sources like files, globals and RSS feeds, the same tight synchronization link doesn't come out of the box. In this article, we'll explore two approaches for modelling DeepSee-like synchronization from table data locations using callbacks and other features of the iKnow domain definition infrastructure.

In this article, we'll assume that you have a basic familiarity with iKnow domain definitions. While you can achieve most of what's described below through the iKnow Architect, for implementing the callback methods you'll have to open the domain definition class in Studio or Atelier. See this article for an overview of working with domain definitions.

Approach 1: Appending with a parameterized query

In this first approach, we'll leverage simple callbacks on the iKnow domain definition interface and use them to parameterize the query that's loading our data. That means, we'll append a WHERE clause that selects only the records that were added since the last time we built. Such a WHERE clause obviously needs a column on the table expressing when each record was added to filter by. In this article, we'll assume your data table has one called "CreateTime" of type %DeepSee.Datatype.dateTime, for example populated by an %OnNew() callback:

```
Class iKnow.SimpleTable Extends %Persistent
{

Property SomeText As %String;

Property CreateTime As %DeepSee.Datatype.dateTime;

Method %OnNew() As %Status [ Private, ServerOnly = 1 ]
{
   set ..CreateTime = $horolog
   quit $$$OK
}

}
```

To select the records that were added since the last time we built a domain, we can use the following SQL statement in the data location for our domain definition:

```
SELECT %ID, SomeText FROM iKnow.SimpleTable WHERE CreateTime >= ?
```

To supply that parameter in our query, we can use either of these two customization options supported by domain definitions:

- Use %domain.MyMethod() syntax to call straight back to a class method on the domain definition class
- Use %expression.MyExpression syntax to work with registered expressions defined with <expression> elements in the domain definition.

The second option is very powerful and mostly similar to working with intermediate expressions in DeepSee. It allows you to supply values for those expressions at runtime, when calling %Build(), but also more complex than what we actually need here. For our purposes, we can simply add a GetLastBuildTime() method and call that through the first option, updating that value appropriately using the various other callbacks in %iKnow.DomainDefinition:

```
Class iKnow.ParameterizedDefinition Extends %iKnow.DomainDefinition
{

XData Domain [ XMLNamespace = "http://www.intersystems.com/iknow" ]
{
<domain name="Parameterized build">
  <data dropBeforeBuild="false">
    <query sql="SELECT %ID, 'SimpleTable' _group, SomeText FROM iKnow.SimpleTable
                WHERE CreateTime &gt; %domain.GetLastBuildTime()"
           idField="%ID" groupField="_group" dataFields="SomeText" />
  </data>
</domain>
}

ClassMethod GetLastBuildTime() As %String
{
  quit $g(^iKnow.LastBuildTime(..%GetDomainId()), 0)
}

ClassMethod %OnAfterBuild(pDomainId As %Integer) As %Status
{
  set ^iKnow.LastBuildTime(pDomainId) = $horolog
  quit $$$OK
}

ClassMethod %OnAfterDropData(pDomainId As %Integer,
pDropDictionaries As %Boolean = 1, pDropBlackLists As %Boolean = 1) As %Status
{
  kill ^iKnow.LastBuildTime(pDomainId)
  quit $$$OK
}

}
```

Now, when you build the domain, it will only append new records added since the last time %OnAfterBuild() completed. Note also that the dropBeforeBuild attribute on the <data> element is set to false, overriding the default behaviour that wipes out a domain prior to building it.

The above code is somewhat simplified and glosses over the fact that records may be added between when your %Build() method starts (and the SQL statement gets executed) and when it finishes with the callback to %OnAfterBuild() runs. Those records would be ignored in the next run and therefore never end up in your domain. To safen the code (left out for simplicity in this article), you can add %OnBeforeBuild() and GetCurrentBuildTime() callbacks to register and then retrieve the build start time, using it in the SQL WHERE clause to only process the records added since GetLastBuildTime() and before GetCurrentBuildTime(). That'll ensure no records are overlooked between domain builds and all insertions will eventually end up in the iKnow domain.

Approach 2: Synchronizing through leveraging DSTIME

While the first approach described above is easy to implement, it only addresses append scenarios. With some work, it could be extended to cover updates as well, but to cover deletions in the source table would really drag it quite far. In those cases, it would be more practical to leverage DeepSee's DSTIME class parameter, which enables automatic tracking of insertions, updates and deletions of a source class so "consumer" classes can follow up on those events. See this chapter for more details on how this works for DeepSee.

```
Class iKnow.SimpleTable Extends %Persistent
{
Parameter DSTIME = "AUTO";
Property SomeText As %String;
}
```

For leveraging the DSTIME data iKnow, we'll have to implement our "consumer" class through subclassing %DeepSee.TimeSync and then implement a behaviour for processing the list of to-be-synced records. In order not to complicate things too much, we'll let this implementation populate a simple table containing the IDs of all new records and then extend our data location's WHERE clause to only select records whose ID is in that table. For records that were deleted, we can invoke the %iKnow.Source.Loader:DeleteSource() method, and for updated records, we just do both.

```
ClassMethod %OnProcessObject(pClassName As %String(MAXLEN=""), pID As %String(
MAXLEN="") = "", pFileAction As %Integer = 0, pTimeStamp As %Integer) As %Status
{
  set tSC = $$$OK
  try {

    if (pFileAction=0) || (pFileAction=2) { // UPDATE or DELETE

      // construct External ID and then retrieve Source ID
      set tSourceID = ..GetSourceIDForTable(pClassName, pID, .tSC)
      quit:$$$ISERR(tSC)

      // Note: this is a costly operation!
      set tSC = ##class(%iKnow.Source.Loader).DeleteSource(..%GetDomainId(),
tSourceID)
      quit:$$$ISERR(tSC)
    }

    if (pFileAction=0) || (pFileAction=1) { // UPDATE or INSERT

      // queue for picking up by WHERE clause on iKnow.SyncedDefinitionBatch
      set ^CacheTemp.iKnow.Sync($namespace, ..%GetDomainId(), pClassName, pID) = ""
    }

  } catch (ex) {
    set tSC = ex.AsStatus()
  }
  quit tSC
}
```

In the above code, note that pClassName refers to the table containing the data we'll be indexing, and not the current class.

We then create a read-only class iKnow.SyncedDefinitionBatch which in its Storage definition maps to the ^CacheTemp.iKnow.Sync global we're writing to. This could have been achieved with a regular class as well, using

%New(), %Save() in the above code, but I chose to cut the overhead as much as possible.

```
Class iKnow.SyncedDefinitionBatch Extends %Persistent [ Final ]
{

Parameter READONLY = 1;

Property DomainID As %Integer;
Property RecordTable As %String(MAXLEN = 500);
Property RecordID As %String(MAXLEN = "");

Index PK On (DomainID, RecordTable, RecordID) [ IdKey, PrimaryKey, Unique ];

Storage Default
{
<DataLocation>^CacheTemp.iKnow.Sync($namespace)</DataLocation>
<Type>%Library.CacheStorage</Type>
}

}
```

We can now leverage this helper table in a WHERE clause on our <table> data location:

```
<domain name="Synced Domain">
  <parameter name="$$$IKPIGNOREEMPTYBATCH" value="1" />
  <data dropBeforeBuild="false">
    <table tableName="iKnow.SimpleTable" groupField="'SimpleTable'" idField=
"%ID" dataFields="SomeText"
           whereClause="%ID IN (SELECT RecordID FROM iKnow.SyncedDefinitionBatch
              WHERE DomainID = %domain.%GetDomainId() AND RecordTable = 'iKnow.Simple
Table')" />
  </data>
</domain>
```

Note that in the above code, we assumed our domain definition class is named "iKnow.SimpleDefinition" and set the $$$IKPIGNOREEMPTYBATCH domain parameter to avoid verbose errors when there's nothing to load. Note also that, if your source table isn't empty before you set DSTIME to auto, there will be no track record of the rows already in the table at that time. In that case, you may want to build the domain without the above WHERE clause first and only add it after this initial load completes.

Now that we have the code to leverage DSTIME info by preparing a table with IDs of records that need to be added. The right time to call this would be as part of the %OnBeforeBuild() callback in our domain definition. If we let our domain definition inherit from both %DeepSee.TimeSync and %iKnow.DomainDefinition (to keep things together and get access to %GetDomainId() easily from the TimeSync implementation), that callback could look like this, leveraging the object representation of the domain definition XML (cf %iKnow.Model.* classes):

```
ClassMethod %OnBeforeBuild(pDomainId As %Integer) As %Status
{
  kill ^CacheTemp.iKnow.Sync($namespace, pDomainId)
  set tSC = $$$OK
  try {
   #dim tModel as %iKnow.Model.domain = ..%GetModel()
   quit:'$isobject(tModel.data)

   // go through the data locations and call %Synchronize() for each <table> element
```

```
   for i = 1:1:tModel.data.lists.Count() {
      if 'tModel.data.lists.GetAt(i).%IsA("%iKnow.Model.listTable") {
        set tSC = $$$ERROR($$$GeneralError,
"only <table> lists can be used as data locations")
        quit
      }

      #dim tListTable as %iKnow.Model.listTable = tModel.data.lists.GetAt(i)

      // store group reference for External ID retrieval
      set ^CacheTemp.iKnow.Sync($namespace, pDomainId, tListTable.tableName) = $lb(
tListTable.groupField)

      set tSC = ..%Synchronize(tListTable.tableName, 0)
      quit:$$$ISERR(tSC)
    }

  } catch (ex) {
    set tSC = ex.AsStatus()
  }
  quit tSC
}
```

The only thing we still need is how we're going to tie records from our source table to source IDs on the iKnow side, in order to know which iKnow sources to remove for DELETEd or UPDATEd rows in the table. You might in fact already have noticed the GetSourceIDForTable() reference earlier on in the %OnProcessObject() implementation. We can track the full structure of our external IDs based on the idField and groupField attributes of the <table> data locations, but that may get clumsy or costly quite quickly. So let's assume the groupField is static (i.e. the table name) and the idField correspond to the table's %ID column. In that case, getting the Source ID based on the record's %ID becomes almost trivial:

```
ClassMethod GetSourceIDForTable(pClassName As %String(MAXLEN=""), pID As
%String(MAXLEN=""), Output pSC As %Status) As %Integer
{
  set pSC = $$$OK
  try {
    set tSourceID = "", tDomainId = ..%GetDomainId()

    // Assumed External ID structure:
    // - SQL lister structure: ':SQL:groupname:localref'
    // - groupname is static and saved at ^CacheTemp.iKnow.Sync($classname(), pClassN
ame)
    // - localref is equal to class' %ID
    set tGroupName = $lg($g(^CacheTemp.iKnow.Sync($namespace, tDomainId,
pClassName)))
    set tGroupName = $replace($piece(tGroupName,"'",2,*-1),"''","'")


 // the safe way to combine these elements is through the BuildExtIdFromName() method
:
    set tExternalID = ##class(%iKnow.Source.SQL.Lister).
BuildExtIdFromName(tDomainId, tGroupName, pID, .pSC)
    quit:$$$ISERR(pSC)
    set tSourceID = ##class(%iKnow.Queries.SourceAPI).GetSourceId(tDomainId,
tExternalID, .pSC)
    quit:$$$ISERR(pSC)

  } catch (ex) {
```

```
    set pSC = ex.AsStatus()
  }
  quit tSourceID
}
```

Note that we saved the groupField value (assumed to be static per pClassName) to our ^CacheTemp.iKnow.Sync global as part of our %OnBeforeBuild() callback implementation.

Full samples and further work

The code pasted above are slightly simplified versions of the actual samples posted on this GitHub repo. For the first approach, look at iKnow.ParameterizedDefinition for a full example, which does take into account the start and end times of the batches as described in the first section of this article. For the second approach, iKnow.SynchronizedDefinition offers the main code as an abstract class you can inherit from, combining the %DeepSee.TimeSync and %iKnow.DomainDefinition interfaces with the callback implementations described above. It also adds a few validation rules to ensure the assumptions we made (such as the idField being set to %ID) are indeed true.

The second approach could be elaborated a bit more. For example, you could try to avoid the costly operations to delete a source from an iKnow domain by reviewing whether an UPDATE tracked by DSTIME actually changed anything to the corresponding text field. You could also leverage an <expression> and spice up your WHERE clause to support reading the initial batch (not yet traced through DSTIME) through a build parameter, or create your own %iKnow.Source.Lister implementation (leveraging much of from the existing SQL one) that automatically satisfies our assumptions and appends that WHERE clause.

Disclaimer: all sample code provided here is meant for illustration purposes only. We welcome any feedback and will try to address any reported issues within a reasonable timeframe, but this code is not certified for production use and not supported through the WRC.

#Best Practices #iKnow