GENESIS64 – Using IRepository to Access Configuration Databases

APPLICATION NOTE

October 2015

Description: Guide to IRepository API that is able to programmatically operate with configuration databases for ICONICS modules configurable via Workbench

General Requirement: Advanced GENESIS64 knowledge, Intermediate C# .NET programming knowledge (Inheritance, Generics, and Interfaces), Visual Studio 2012 / Visual Studio 2013 / Visual Studio 2015 (even Express edition) installation, Relational databases basics

Introduction

The IRepository API (Application Programming Interface) consists of .NET objects allowing to automate operations with configuration(s) of ICONICS modules, in other words - to operate with these configurations via .NET code, using C# .NET or Visual Basic .NET programming languages. **Entities, their relations and properties used throughout whole API are reflecting design of SQL databases and their tables holding configuration data.**

We need to mention that because IRepository API uses .NET Generics you cannot write the code using Jscript.NET (the language used for scripting within GraphWorX64) because Jscript.NET does not support Generics. If you need to make configuration changes from within GraphWorX64 displays it is recommended to use the following approach:

- Write the required code in Visual Studio (using C# or VB.NET) as Class Library.
- For easy and straightforward usage in a GraphWorX64 display, your library should expose only the required methods with necessary parameters.
- Compile the code to the .dll library file.
- Reference this library in GraphWorX64.
- Call your publicly exposed library methods from Jscript.NET code in order to perform desired operations.
- Dispose of all objects and connections when you are exiting GraphWorX64 or switching to another display (use AnimationStopping or AnimationStopped GraphWorX64's ThisDisplay events).

Code samples showed in this Application Note are written in C#. Object and Type naming remains the same for Visual Basic .NET. We will also explain important aspects of the API and its usage without more-than-needed details and in understandable form.

API Libraries

IRepository API can be divided into two parts:

- Common part that creates and manages the communication to the configuration databases and common business logic
- Module-specific (or provider-specific when speaking in Workbench-like language) part containing business entities and business logic for specific module

Based on this, in your Visual Studio project (WinForms application, WPF application, Class Library or any other project type) you have to always reference these common (shared) libraries, and you would also need to reference the specific libraries for the module you would like to work with.

Libraries to reference in your application

The generic libraries you always need to reference in your project are:

- IcoComponentModel.dll
- IcoConfigCommon.dll
- IcoConfigCommunication.dll
- IcoConfigStorage.dll
- IcoMvvmCommon.dll

The module-specific libraries consist of two .dll files for each module. The names of the Assembly files follow this convention:

- Ico*Configuration.dll
- Ico*Definitions.dll

Where the asterisk is replaced by module's name abbreviation. The following list shows the ICONICS module name and the abbreviation used in the .dll library name:

Table 1 - Module to IRepository API library filename mapping

ICONICS Module Name	IRepository API name abbr.
AlertWorX	Alert
AssetWorX	AssetCatalog
AlarmWorX Logger	AwxLogger
AlarmWorX Server	AwxServer
BACnet	Bac
Energy AnalytiX	Ea
FDDWorX	Fa
FrameWorX Server	Fwx
Global Aliasing	Gas
GridWorX Server	Gdx
Hyper Historian	HH
MobileHMI	Hmi
Language Aliasing	Las
MergeWorX	Mgx
RecipeWorX	Rcp
ScheduleWorX	Schwx
Security	Security
SNMP	Snmp
TrendWorX Logger	TwxLogger
Unified Data Manager	UDM
Web Services	Ws

All of these libraries are located in the Component folder of GENESI64 installation, and by default the path is C:\Program Files\ICONICS\GENESIS64\Components.

A situation that may occur is operations with specific entities may require other libraries since the related objects or methods are stored there.

API Concept

All IRepository entities (in other words building blocks) represent configuration "pieces" (for example in case of AlarmWorX Server the entity is AlarmWorX Server Tag with class name of AwxSvrSource, AlarmWorX Server Area with class name of AwxSvrArea, etc.) that implement specific interfaces. Methods, which are parts of IRepository classes, expect objects that implement these interfaces and/or return such objects. Understanding .NET Generics, Interfaces, Inheritance and this concept used in the API is crucial. The basic building block of the API is object is implementing IEntity interface, which is defined as:

```
namespace Ico.Config
{
    public interface IEntity
    {
        IEntityType EntityType { get; }
        IEntityKey Key { get; set; }
        IEntityKey ParentKey { get; set; }
    }
}
```

Each object in the tree has its own Type (EntityType), unique identifier (Key) and unique identifier of the parent (ParentKey). Unique identifier (object implementing IEntityKey interface) consists of an integer number – ID and EntityType – this combination is always unique within single configuration database. ParentKey allows you to get reference to a parent object (thanks to this you can move in the tree and you will know the relation between two entities).

For our purpose and basic understanding of this we need know following:

- Object implementing IEntityType is named **EntityUniqueType**.
- Object implementing IEntityKey is named **EntityUniqueKey**.

Important objects and methods

Repository

Each module-specific library contains module-specific entities allowing you to work with configuration databases. Most important and common for all modules is the IRepository-derived class which is contained in the Ico*Configuration.dll. Repository object name is unique for each module, and follows a simple naming convention: *Repository where * (asterisk) is a module name abbreviation as is specified in Table 1. It resides in the Ico.*.Configuration.v1 namespace.

Each Repository object contains a set of methods; here is a list of the most important ones:

- Read()
- ReadAll()
- ReadByParent()
- Create()
- Delete()
- Update()

Other helpful methods can be:

- Import()
- Export()

- BeginTransaction()
- CommitTransaction()

Note: The list of methods above is not a complete list.

Module-specific entities

Each module contains its own-specific entities representing building blocks that are specific to each Repository or each module's configuration. As mentioned before, each entity implements the IEntity interface, and it also exposes its own specific properties that hold related information. Entities reside in the Ico.*.Definitions.v1 namespace.

Common operations

In this section we will show all basic operations and techniques you need for interacting with the configuration databases. Each section will contain examples related to the **Hyper Historian** Repository and some of these examples will expect sample configuration that is included with GENESIS64 or Hyper Historian installation. In this example, the Visual Studio project will have to reference:

- IcoComponentModel.dll
- IcoConfigCommon.dll
- IcoConfigCommunication.dll
- IcoConfigStorage.dll
- IcoMvvmCommon.dll
- IcoHHConfiguration.dll
- IcoHHDefinitions.dll
- IcoTriggerCommon.dll (HH-only related library)

You should also specify these namespaces in "using" section of the .cs file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data.SqlClient;
using System.Data;

using Ico.HH.Configuration.v1;
using Ico.HH.Definitions.v1;
using Ico.Config;
```

Physical connection to SQL database containing module's configuration data

Before you can start using any methods on the Repository object you need to bind it with a valid System.Data.SqlClient.SqlConnection object instance which has a valid connection string pointing to the repository-related configuration database on the SQL server. Binding is done by assigning the SqlConnection object to *Repository.Connection property.

You do not need to pass SqlConnection open, since the connection will be opened and closed by logic inside *Repository class (it is done as is needed internally, including closing after *Repository class instance disposal). If you pass in the SqlConnection opened, it still remains opened and you need to take care about its closing and disposal.

Here is an example (in case of copy/pasting this example to your project, do not forget to change the SqlConnection's connection string):

```
SqlConnection hhCfgDbConnection = new
SqlConnection("Server=myServerAddress;Database=myDat
aBase;User Id=myUsername;Password=myPassword;");
HHRepository hhRepository = new HHRepository();
_hhRepository.Connection = hhCfgDbConnection;
//now you're ready to work with HH configuration
programmatically (if your connection string and
database connection is valid, otherwise you will get
exception during first method call touching database
data
```

Querying for existing objects

For this purpose the Read(), ReadAll() and ReadByParent() methods are intended.. These methods are very important as they returns specific entity instances which can be used in Delete(), Update() and other methods. One aspect to consider is that these methods are synchronous and querying larger amounts of data can take longer. Based on this there are some parameters related to monitoring these requests:

- *System.Threading.CancellationToken cancellationToken* - allows to cancel the running operation
- *System.IProgress<Ico.Config.OperationProgressInfo<IEntity> progress* - allows you to monitor query operation progress

These Read-related methods also accept the *string[] requestedPropertyNames* parameter, which allows you to query for specific entity-related properties which may improve the performance. This is useful when you want to query larger amount of items and you are interested only in specific properties (for example entity's name and description). When this *string[] requestedPropertyNames* parameter is not used all properties of an entity are read from configuration database.

Keep in mind that if you want to update a specific entity, all of the properties you want to update need to be requested and if they are not, they will be lost.

All these parameters are optional or a null value can be passed.

ReadAll()

This method returns all entity instances of specific EntityUniqueType in whole configuration database as IEnumerable<IEntity> object.

In the following sample code we will show how to call the method to get all Hyper Historian tags in the configuration (notice usage of EntityUniqueType.CreateFor<TEntity>() method creating specific EntityUniqueType object), as well as options how to filter out specific items (simple search) from returned result set:

```
IEnumerable<IEntity> readAllResult =
hhRepository.ReadAll(EntityUniqueType.CreateFor<HHTa
g>());
//by foreach loop you can go through each retuned
entity and perform what you need, for example we may
want to store all names starting with "Sine" to the
string variable
string hhTagNames = string.Empty;
foreach (HHTag hhTag in readAllResult)
{
       if (hhTag.Name.StartsWith("Sine"))
       hhTagNames += hhTag.Name + ";";
}
//in the loop above we just extracted the name but
by this technique you can grab whole object
instances for additional operations (e.g. changing
the datasource and pushing changes back to the DB)
//you can get these instances of HHTag entity by
using a single-line LINQ query
IEnumerable<IEntity> hhTagsStartingWithSine =
readAllResult.Where(x =>
((HHTag)x).Name.StartsWith("Sine"));
```

Read()

This method is used in cases when you need to get entity instance reference(s) while you know EntityType and ID (in other words objects implementing IEntityKey, like EntityUniqueKey).

In following sample code we will continue and use what we did with ReadAll() method example above - we will get references to the HHFolder entities which are parents of HHTags with name starting with "Sine":

```
//we need to initialize List<IEntityKey> where we
are storing Entity Unique Keys we want to get
references for to be used in Read() method
List<IEntityKey> parentKeys = new
List<IEntityKey>();
foreach (HHTag hhTag in hhTagsStartingWithSine)
    parentKeys.Add(hhTag.ParentKey);

string folderNames = string.Empty;
IEnumerable<IEntity>
parentEntitiesOfTagsStartingWithSine =
hhRepository.Read(parentKeys);
//because parent can be a root of Hyper Historian
Data Collections we need to take this into account
//following technique shows how to compare
EntityUniqueType
foreach (var parentEntity in
parentEntitiesOfTagsStartingWithSine)
{
    if ((EntityUniqueType)parentEntity.EntityType ==
EntityUniqueType.CreateFor<HHFolder>())
        folderNames += ((HHFolder)parentEntity).Name
+ ";";
    else
        folderNames += "Root;";
}
```

ReadByParent()

This method returns all child items (not recursively, children of itself) of entity specified by IEntityKey.

This example will again continue in ours previous code and gets references to all child items (HHTags and HHFolders (if there are any) of first folder in parentKeys List<IEntityKey> and lists their type and name (as code uses previously used techniques it is not commented):

```
IEnumerable<IEntity> firstFolderChildItems =
hhRepository.ReadByParent(parentKeys[0]);

string firstFolderChildItemsNames = string.Empty;
foreach (var item in firstFolderChildItems)
{
    if ((EntityUniqueType)item.EntityType ==
EntityUniqueType.CreateFor<HHTag>())
        firstFolderChildItemsNames += "HHTag - " +
((HHTag)item).Name + ";";
    else
        firstFolderChildItemsNames += "HHFolder - "
+ ((HHFolder)item).Name + ";";
}
```

Editing existing objects

For committing changes in specific entity instance you need to call Update() method with the member or the Repository class.

This method accepts single parameter of IEntity type which is the entity instance with changed properties to be updated in the configuration database. For this example, you need to reference the specific instance by using one of the Read methods, change values of wanted properties and call the Update method.

This is shown in following example, where we get our very first instance of HHTag object with name starting with "Sine" (ReadAll() method section), change the name, and update it in the database:

```
HHTag veryFirstTagWithSineName =
(HHTag)hhTagsStartingWithSine.First();
veryFirstTagWithSineName.Name += "SlightlyChanged";
hhRepository.Update(veryFirstTagWithSineName);
```

Creating new objects

For this section, we will not use the Create method to add in new entities. First you need to create a new instance of the entity you want to create, specify values in all parameter which cannot be null and you do not want to forget to specify an EntityUniqueKey so that this object will have a parent object.

In this example we will create a new folder in the root of Hyper Historian Data Collection:

```
//get Root Folder EntityUniqueKey (key) of HH Data
Collection
EntityUniqueKey dataCollectionsRoot =
EntityUniqueKey.CreateFor<HHDataCollectionsRoot>(Ent
ityUniqueKey.RootId);

//set up new HHFolder with appropriate name, as well
as other properties
HHFolder hhFolder = new HHFolder();
hhFolder.Name = "MyNewFolder";
hhFolder.DisplayName = "MyNewFolder";
hhFolder.Description = "My New Folder Description";

//now set up the parent key
hhFolder.ParentKey = dataCollectionsRoot;

//create folder in the configuration database and
store the result to newFolder variable which will
hold IEntity instance
IEntity createdFolder =
hhRepository.Create(hhFolder);

//if you want, you may get the key of newly created
folder (for example when you want to reference this
folder as a parent of another folder or tag)
EntityUniqueKey createdFolder Key =
(EntityUniqueKey) createdFolder.Key;
```

Deleting existing objects

Deleting is also straightforward and you just need pass single IEntity instance, or multiple IEnumerable<IEntity> instances as parameter of Delete() method.

To delete the folder we just created:

```
hhRepository.Delete(newFolder);
```

Improving performance

One way to improve performance is to use overloaded variant of methods accepting the IEnumerable<IEntity> input parameter instead of single IEntity. For example methods we explained above – Create(), Delete() and Update().

Other way to improve configuration database operations can be achieved by using BeginTransaction() and CommitTransaction() methods. It is useful in cases where you are performing large amount of operations in the loop. Normally the create/delete/update operation is made immediately after calling appropriate method. This may not be as efficient, since you are calling operations in a loop. An unexpected exception may also occur while looping and you may want to roll back the changes you made since the transaction began.

Example:

```
try
{
    hhRepository.BeginTransaction();
    foreach (var item in largeListOfItems)
    {
        //perform some operations with the objects
        //and call database operations methods, like
hhRepository.Update()
    }
    hhRepository.CommitTransaction();
}
catch (Exception e)
{
    //handle the exception and roll back
    try { hhRepository.RollbackTransaction(); }
    catch { //even roll back can fail }
}
```

Sample application

If you are interested in a C# .NET sample project showing operations with Hyper Historian configuration please contact your ICONICS Tech Support department.