Code Generator Architecture and Features

Author: Kevin Guerra

**Scope**

This document contains information about a Windows form application which we refer to as the code generator. Because the code generated is used for our core application the ID engine, this document will touch briefly some points, however a comprehensive explanation is beyond scope and will be explained in detail in a separate document.

This document will attempt to describe JakeKnows code generator. The current version lacks proper design, and it is something that grew out of necessity to provide rapid application development database access and web service wrapper code.. The system uses .NET framework 4.0 since it provides numerous improvements over the previous version. C# is the chosen language. The current version does not full automation of how the code is used for all features. This means that for some features it is necessary to manually edit the code and these cases will be explained later in this document.

# Background

This application is a Windows form application with some buttons to indicate the type of code we want to generate, and some text boxes to indicate what files it will write to, but these are not complete in the sense that it also generates quite a lot of files that are not specified by the user, but rather hard-coded in the app. It does have a configuration file to store the database connection string.

# Benefits

1. Generates sql CRUD (Create, Retrieve, Update Delete) stored procedures uniformly.
2. Generates C# code to access the data, only through those stored procedures.
3. Generates Web Services wrapper code, based on database table entries.
4. Generates data import code, for another application that is used to import data from the head widget database into the new ID engine database.
5. Template based code generation. (Not fully, since there is some code in the app that is C#, but should be changed later)
6. Application is not generic. It is written specifically to be used for the development of the ID engine. But can be extended for other uses. Can be used for other apps as is, as long as the way to access the data remains the same.

**Creation of Stored Procedures**

Stored procedures are created from database schema. A view to a systems table is created prior and it provides information to the code generator about the tables, columns, data types, lengths for varchar fields, etc... Another table is created prior and it contains field mappings so that the web service call parameters can be mapped automatically and ID fields can be obtained from strings, conversion and validation is also achieved via the information provided by this table.

The code is generated based on template files, there is one template for each type of stored procedure, the templates contain the code in the style and features that we want the generated code to contain in a generic and uniform way. The template files contain tags that mainly consist of two types, table iteration and field iteration. Table iteration are tags that are updated when a new table is in the loop and this will cause the full set of stored procedures applicable to that table to be written to the output file. Field iteration tags are those that are updated in the loop when a field changes, so for example, let's say we have a stored procedure to update a table row, this sp will contain the table name and field names, so the names of those fields are obtained from the database and appended to the list of fields that need to be updated in the database.

Some of the template files are for the whole stored procedure and others are used to template the list of fields. The code generation is done by a series of substitutions of these tags and when the loop completes we will have a file with a list of all stored procedures to be imported into a database.

Note that, the code generator and the templates assume a particular style of coding, schema and naming conventions in order to work. All generated stored procedures have a prefix of cg\_usp, cg (code generated) and usp (user stored procedure). This is so that when the generated file is used in the database all cg\_usp\* will be deleted first and then regenerated, this avoid several update problems during import.

In the ID engine, tables serve different purposes and these purposes are abstracted so that we categorize table types. We have main tables with a tbl prefix, association tables with either assoc, link, ovr, flt, plus other prefixes, these table types all associate records from two or more tables in a uniform manner, but have different prefixes due to different uses and this is also understood by the C# code generated. Other tables are used to contain fairly static data akin to enumeration types in many programming languages, the prefix used is status and type, fields of these two types are ST and TY respectively.

Some tables are used to look up information that is mapped to Id values. These tables share in common two fields, it's ID field starts with the ID prefix and the information field according to type. For example, we have called tblEmail with two fields IDEmail, Email, TYEmail (type of email). Then we also have a tblProfile which can be associated to many emails and these emails all have an associated type to distinguish them. Then we have assocProfile\_Email, association tables are generated depending on the tables associated and these tables names are separated by a '\_' character, so the above example means that a row entry in that table would be IDProfile, IDEmail, SAQ, DateCreated, etc...

The code generator also knows which fields values are autogenerated by the database so that during a record creation it does not have to pass in that field but it will return the ID of the field that was just generated, so that the code can save it for future reference and identification of the particular record.

As long as uniformity is kept in the schema it will work flawlessly, otherwise, when a table deviates from the norm, it is necessary for the user to edit the produced code, and sometimes adjustments to be made until after the import. An example of this situation would be the creation of code for a table which uses an ID field but it is not an auto-increment field, for example imported records from other sources, but otherwise we want the benefit of the association of the ID filed with a particular data column, in this case, we must edit the stored procedure so that during the record creation we must pass in the record ID that we got from the imported data. A concrete example of this situation is given in the new ID engine documentation.

**Generation of data import code**

Our production database is operational 24/7. It has been necessary on several occasions to import this data. The first time, it was done manually, and since the new ID engine has some differences in schema, the operation took several days. This was done in a semi-manual process using the MS SQL Data Import Utility, which as explained below proved infective. So, the code generator was extended to generate code that will be used in another utility I developed for the data import process. This process takes about 3 or 4 hours without user intervention.

This generated code is used by another project, windows form that uses it to generate an executable that imports data from headwidgets. The reason this was done is to keep as much compatibility as possible, the compatibility is at the functional level mostly and to a great extent on the data itself. It was not the best to keep full data compatibility since the old ID engine had lots of inconsistencies, for example it used unix dates (integer field) and often times it used DateTime fields, or it named a field one way and in another table it had a different name. It would not have been practical to generate code given these circumstances.

The other problem was the MS SQL Studio Import Utility. It proved to be worse than useful, I naively believed at first that I could actually use it for that purpose, only to find out, that all the field mapping I had to do manually using it, would not import the data reliably and the failure of just one record (because of bad data) would abort the whole operation for that table and leave many records in and many out.

My import utility does not suffer from these issues and will import as much as it can and will log what it can't so that we can take appropriate action and we can treat our data with the respect it deserves.

The code generator produces a file with C# code that contains 'for loops' for each table, and through metadata stored in the database, the file contains conversion code when necessary, it catches exceptions, so that if a record import is in error it would not cause the failure of the whole table import. So it is a long list of for loops for each table, where each one is tailored for a particular table mapping.

**Creation of Web Services wrapper code**

In the current architecture, because we are using IIS, and because we want to use cached data, we need the ability for the server app to be alive at all times. The nature of web services is similar to how you access a web page, there are no moving parts, so to speak until a client makes a web service call. At this point the IIS server invokes the code for that call.

Our new ID engine is a Windows Service application, which is started when the system starts up, but we do not currently provide self hosting, meaning that the engine does not provide web services directly. It does so via a IIS web app DLL. This DLL 'is' the wrapper code that we speak about. The code in this web application is almost 100% generated, the code that is not generated is utility code that exists as part of the Visual Studio project.

What this means in practical purposes is that there is nothing to do here as far as programming is concerned. All one has to do is to edit a database table that provides the mapping of all web services.

For example, let's say we have a web service called updateUserProfile which takes two parameters, name and phoneNumber, but later we want to add a parameter named email. We add an entry in the database, a new row, it's name will be email, and we well say it's a string, and it's validation type is email, so that auto-validation can be done, and we don't want conversion done so we leave that blank, etc... Then we start up the code generator application and we click on the button that generates the web services. The product will be a file named JKCG\_WebServices.cs, we copy the code from that file, open the Visual Studio project that generates our DLL IIS Web Application and paste all the code in. Then rebuild the application. The generated DLL will then be copied to the directory where our IIS server hosts our web app, usually C:\inetpub\wwwroot\app\_name\bin.

This web app dll, when invoked by a client, will package all parameters in a .NET DataSet structure. This uses the PolyModel pattern. It will then open a connection to the ID engine which is the Window Service application, and send the package.

The ID engine receives the package via a named pipe, the fastest IPC for processes in the same machine. Then it will decode the package, and apply the strategy pattern so forward the call to the appropriate handler. We have exactly one handler per web service call. Each call is handled in a new task/thread which the .NET framework handles as efficiently as possible, in other words in will only create as many threads as the server is able to handle efficiently as opposed to creating as many threads as calls come in, since creating more threads this way leads so performance degradation and in some cases a very severe one.

**Generation of code**

Now the fun part. As you probably realized the above section offered a peak as to how the generated code relates to data access, and mainly how it relates to web service calls. This is the more complicated part, but only complicated due to the sheer volume of generated code in relation to the schema. In reality, the process is almost exactly the same as the process for stored procedure generation. It uses template files and table names, field names, and information about those fields obtained itself from the database.

This is the list of types of generated code:

1. Tables
2. Table Associations
3. Field Mapper code
4. Objects (Database)
5. Objects Containers, uses Ring Buffer dictionary for caching and uniform data access, from previously written code for this purpose that is not generated.
6. Strategy files, the handlers for each web service call.