Web Service Architecture

This document will attempt to describe the new Identity Engine application as it is being developed. It will describe several modules (in the broad sense of the word, a module can be a .NET assembly or a group of related classes.) It was decided that we keep the current Microsoft technologies for this new development. The new system will use .NET framework 4.0 since it provides numerous improvements. C# is the chosen language, using Object Oriented technologies to back a Service Oriented Architecture.

Please note that this is an evolving document and also an evolving application, since it is early in the development process. Also, there are several alternative pathways that can be taken, for example, we can host the web services directly in the engine and do away completely with IIS, or, we could simply use a subset of the cache and have all the objects be retrieved from the database to avoid possible concurrency issues, similar to how the current application handles it, but this we could do only if concurrency becomes an issue, since the availability of objects in the engine would enhance performance greatly.

# Background

The current application lacks a uniform design and many features that are necessary for world class application status. We won’t go into many details. The application is hosted as a Visual Studio web application in which all web services are available in the same class. It uses a common library for diverse functions and utilities. When a client request a service call, all initialization takes place on the fly, including data retrieval from a remote database, other database operations happen usually via the other functions and there is repetition of retrieval, the data is usually retrieved as needed and this happens many times (fine granularity, in this case not a good thing.) The database does not enforce referential integrity and does not provide other features that a RDBMS should provide an application, as a result, the responsibility of record de-duplication is left to the application; this, should not be.

# Description of Modules

1. Web application to host web services
2. Windows Service application to host the application engine
3. Library module that implements the engine
4. Communications module (TBD)

The web app communicates with the windows service via WFC named pipes. It uses per call thread instantiation which means that each new call (service request) receives its own thread, this is so that the transmission of parameters and results data can be done asynchronously. For each call the engine assigns a task with its own thread, this is a separate thread from the previous one, and so that the code execution takes place asynchronously as well.

The application is to be backward compatible and will support versioning. The code in the web application is generated almost entirely by the code generator and it is a wrapper that packages the parameters in a dataset and it adds a function id and a version number. The web application does some clean up of strings (removal of duplicate contiguous spaces) and it also knows to omit packaging unused parameters. It then opens a connection (named pipe) to the engine (windows service) to transmit the data and waits synchronously for a response. Note that tests conclude that we can also use multitasking at the web application level should we need to and that this provides asynchronous operation, so that for instance a web service call can make more than one requests to the engine, and that the operations can be chained, this would facilitate refactoring of code for related calls. The engine, upon completion of a call returns a string representation of xml data, which is then converted to xml, and the web application then retrieves the relevant data from that and returns the data according to its return type, usually xml, but can be a string, integer, or void.

When the engine receives a service request it first performs validation of input and data conversion, since most parameters passed in the web application are inconsistent as to naming and data type used, this allows for a consistent handling of all parameters. The input validation is done according to the parameter. We have a table that describes each parameter in all web service calls, along with data describing its original type, what it should be converted to and other data that the code generator uses to make the input validation seamless, consistent and automatic.

Should input validation fail a customized return object will be filled with relevant information to be sent back to the caller. The input validator uses regex expressions that get compiled into an assembly (DLL) for greater execution speed. Upon successful validation, the engine uses the strategy architectural pattern to resolve the type of call. This is done via the function ID and version, and each call resolves to one C# class file where the relevant code will be.

Next is the data domain resolution for parameters that should resolve to Identity fields in a database table, this is handled automatically by the code generator and uses reflection and the database objects cached in a module. For example, a parameter name may be ‘Carrier’ this is known via reflection (via database fields table) that it should resolve to IDCarrier, and the code generator adds the function call to pass in the value of ‘Carrier’ and it will return the ID for that carrier. At this point we are about ready to perform the operations on the data.

This is achieved by having the other tables cached in a Ring Buffer Dictionary Multiton data structure. This is a bit complicated but it basically means that the Ring Buffer will only permit a configurable maximum number of cached records per table, the Multiton Dictionary will make the data available to only one thread at a time, this avoids the performance penalty of using lock or semaphores. The Multiton is similar to a Singleton in that it can only have one instance, but in this case is one instance of a record for all records. The multiton, is not a ‘real’ Multiton since it will not use static data, because doing so will cause a memory leak since the application would not be able to free that memory.

The other table objects have related internal objects that are really association tables, and this is how navigation through the relationship trees is achieved. Microsoft .NET provides a technology called LINQ which is a way to perform queries on objects similar to SQL, this offers a way to achieve SQL functionality that would normally be only available in the database server, in the related objects, for a great deal of speed. This prevents us from having to get data from the server on a small subset basis, since we would expect the data to already be there. When the call is completed, another task can be created to handle the saving of data, releasing the caller much sooner than what is currently.

The code generator connects to the database to figure out how to generate the code, it also uses file templates for different stages, Web Service calls or Database objects for example. In order for the code generator to function with simpler code, it is necessary to keep a consistent way of naming tables and fields. This is how the code generator is able to figure out the different relationships among the tables. An extra table was added to the database to describe each web service call and each parameter and the diverse type of mappings, conversions and other operations that the code generator needs to resolve.

The communications module will handle all email, sms, and any type of communication, the functions are already in a library (TBD) however, it is necessary to put this in a separate module. There are other functions in the utility library that would also go into a module but these would be C# classes in the engine.

# Architecture Features

* Input Validation via regex
* Configuration file to host configuration data and avoid ‘hard-coded’ values in the code
* Database Caching, using a concurrent dictionary for fairly static data
* Full asynchronous multi-threaded operation
* Logging and local and remote Tracing capability, with conditional compilation for debug entries
* Consistent exception and error handling
* Use of patterns to facilitate maintenance and ease of understanding
* Reflection used for resolving and mapping parameters to data origins
* Code generation used for data, web service calls and other parts of the code
* Regression and Unit testing
* High capacity, it should handle about 200 ws calls per second

# Web Service Call Pattern

The web application uses the polymodel pattern, it is mainly used in the Microsoft side of things and it's really not a very popular pattern. So I'm not really using the pattern itself but a modified version.  
  
Benefits:  
1) Avoids the proliferation of web service calls, by providing one entry point  
2) It is backward compatible in my implementation because, all service calls are available and then the wrap their values into the xml structure (dataset) to be consumed by the app  
3) New API calls can simply be added and deployment is somewhat simplified since we'd only update one DLL  
4) More consistency in the application structure, since the one entry point would have the decision logic to invoke the pertinent operations

5) More parameters can be added to a web service call without affecting the application structure, the Decorator pattern would be used to provide the additional functionality  
  
Drawbacks:  
1) Since it uses xml for transmission, there's that overhead, however this overhead is in the milliseconds range, I timed it, and we use xml in (soap) and xml out anyway. In several sites, it was noted that when the transmission is not xml there is other type of overhead for packing of binary or string data that is either similar or less performant than xml transmission.  
2) This is a microsoft centric approach, because of the use of the dataset, but we use them extensively in the application anyway, however that detail can be changed for non MS apps. However, because the wrapping happens in the web application this extra step does not affect the caller and it is completely transparent, so as to provide the required backward compatibility.  
  
I've already implemented, tested and measured this mechanism, and in my opinion it is so far, at least adequate.