Sony Application

Web Service Architecture

This document will attempt to describe the Sony application architecture. 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 over the previous version. C# is the chosen language, using Object Oriented technologies to back a Service Oriented Architecture.

# Background

This application will provide Sony Playstation division the capability to empower university students to apply for jobs offered. The company uses Taleo for managing users, job requisitions, etc...

The application will use Taleo web services to query and save user information as it pertains to job applications, it will also use JakeKnows Identity Engine web services to manage user profiles and other information related to user identity. It will also use a database to store other information relating the two sources of data mentioned above as well as other data which is specific to this application.

# Description of Modules

1. Web application to host web services, hosted in IIS (web server)
2. Windows Service application to host the application engine
3. Library module that implements the engine
4. Taleo web services module
5. JakeKnows Identity Engine web services module

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 supports 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. Note: This is implemented in the code, but it is used only partially since we need to test the validation regular expressions.

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, this is internal, since the phone app still has to wait for a call to complete.
* 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, not implemented yet.
* 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.  
  
Communication with web services

The engine uses WFC to communicate with the web services through a proxy. The proxies' code is generated via Visual Studio by loading WSDL files obtained from each of the web services. Then a wrapper class is used to simplify the calls and to tailor them for use by the app. The wrappers handle the details and idiosyncrasies of each web service.

The wrappers also extend to include our own database, for each table, so that the main code can proceed in a higher level manner, and there is less coupling of classes.

Taleo web services proved to be quite deficient, not well documented and not much in accordance with the few documentation provided. Lots of workarounds were developed based on a trial and error time consuming approach in order to make them useable by our application.

Application Interface – Web Services provided

As mentioned above the web services part is a wrapper in a dll module which is a web application hosted in IIS (Internet Information Server,) a web server. This wrapper then packages the parameters passed in to each web service call by the client. In our case the client is the phone application, and this phone application interfaces with our server app, through a proxy type application called SoftCore.

There are 6 web service calls requested by the phone app, plus one which serves as a generic debug interface which I've added to provide any extra functionality needed today and in the future. The following is a list of these, an overview and then some details.

1. RegisterUser. This function is used to create a new user in our system. Parameters: firstName, lastName, major, university, mobile, email, password, deviceUID, graduationDate, homeTown, playerName, favoriteGame, myTalents, myPhoto, appId. The registration happens in three steps, after validation of some parameters. Preliminary, we look for an existing profile with the same data, if found, an error is returned. Then we register with our ID engine, and the rules there are somewhat different from the previous one, in that the engine tries to match the user with an existing one through several means and if all those fail a new profile will be created, otherwise the found profile is returned. This process generates a token which is saved locally and it is used throughout other calls. The second process registers the user with Taleo. Third, we create a local profile using our database and we store the parameters passed above, along with information returned from Taleo and our ID engine. Last we return a successful call return message which is wrapped and then sent back to the direct caller which is the web services app, and then it will compose the message that is appropriate to return to the phone app (through SoftCore.) Note that in any failure in the prior steps an error message is returned right away.
2. SaveProfile. This function is used to update user data. Parameters: deviceToken, firstName, lastName, major, university, mobile, email, graduationDate, homeTown, playerName, favoriteGame, myTalents, myPhoto, appId. Again, parameter validation first. We then check for a token match to make sure that the caller has rights to that data. Then we search for an existing profile which must be found or an error is returned. Then we only update if any field has changed, this is checked locally and if applicable for Taleo and our own ID engine.
3. LoginUserAndReturnData. This call is used to obtain user data, to login a user and to reset a password. Parameters: email, password, newPassword, deviceUID, appId. Validates profile via email. If new password is provided, after a password reset, it is validated. Then it is updated. Otherwise it is a simple login operation, which it used as a way to query user data, and then the data is returned.
4. GetUpdates. It is used to query jobs available. No parameters. A long list of all jobs available is returned.
5. ApplyForJobReq. Applies for the job. Parameters: deviceToken, reqId, resumeImageB64, appId. Currently we don't support resume image. Validation of params, profile resolution, then we call Taleo web services to apply for the job.
6. Debug. Used to do various operations not possible otherwise. For example, force a refresh of all jobs, universities or majors. Currently only those are available through this interface.

Error handling is done via checking and validating parameters, resolution of profiles, plus other operations and when there is an error a message package is return indicating various codes and a message as a string. Also during system or other exceptions, the same mechanism is used to notify the caller. At the moment we do not attempt error recovery, only in the case when there is a warning it is indicated in the result but the call succeeds.

The system accepts various configuration parameters via a configuration file. There are two in fact. One for the web app and one for the engine.

Logging of messages and other conditions is done via configurable parameters in the above mentioned file, and by default is done to a text file, but can be easily extended to save to a database or even to email.

Should there be a need for additional data to be saved to the database, the code generator can be reused for this purpose and will update the code files as necessary. The code generator produces stored procedures and C# code. The code produced is saved to files which are not modified and modification of the same classes can occur via other files and they can affect the non modified code. In other words, should the generated code need more tailoring, it can be done to a separate file affecting the class of the generated code. This makes maintenance much more feasible.

Database connectivity is done by the Sony app directly via .NET classes, that are used by the classes of code generated. This allows the app to be data agnostic to some extent. This is because complete data independence would mean to have an extra layer, and I wanted to improve performance, specially since we're already using a heavy system such as .NET and C#. In any event the code generator makes up for the need should it arise, of switching technologies.

There is more documentation regarding the Taleo API in PDF documents provided by them. We do not currently have much documentation regarding our own ID engine. I've generated documentation via Doxygen for the ID engine and for this Sony application. The documentation produce is a compilation of the code and diagrams of calls and callers (of functions) are produced. This documentation is quite extensive, and more apt for developers but it is a standard way of documenting code.

Other documentation that could be included would be use case scenarios, deployment diagrams, and class relations. The class relations incidentally are available in the Doxygen html docs.

Application Deployment

Two installers were developed for this purpose. One deploys the application to an IIS server. This has to be done locally in the machine where IIS is, and IIS has to be configured to contain the new app that will host the web app to be installed.

The other installer will install the application as a windows service, which is a process that is started by the system automatically during system startup and it is managed through the MMC (Microsoft Management Console.) I've not yet included automatic registration of the process, so it may have to be done manually via a utility called InstallUtility.exe in the .NET framework directory in the windows system directory. This may or may not be happening automatically.

Then, the database must be available in a MS SQL server so that we can save the application data, such as profiles, jobs applied for, etc.. This process has not been automated and it is done manually, by restoring a database backup into the new server that will host the database. Then a user, jakeAdmin, must be granted rights to the database, this is also done manually so far.

After deployment is complete, the web.config in the web app, and the app.config (this file is no longer named that, but the name of the process, and it is installed in the same directory as the process (Engine process) where you installed the engine part of the app; would need to be edited so that the services and ip addresses of the servers are correct, for database connectivity. Other parameters may be edited, for example, level of logging, log file locations, etc...

Should there be a need of more extensive logging for debugging purposes, we can also enable call tracing, this will created another file with very detail information about what the app is doing and input and output data, it is very verbose and should only be done during debugging or development, otherwise of course the system performance would be affected.

This is a draft one, since we still have a few more issues to resolve and possibly other features to be implemented in a future release. Also, there are other documents available that explain the client or phone app.