Supplier Configuration Utility
TraceGains, Inc.

Orion Kostival

March 8, 2013

I. Introduction

TraceGains, Inc. is an emerging document management company that aims to attach actionable intelligence to the data extracted from their customer's documentation. Focused heavily on the food manufacturing industry, the main goal of TraceGains is to provide their customers with a way to digitize their documents and attach real-time monitoring to the data present in those documents. A majority of food manufacturing companies currently have filing cabinets full of regulatory documents which will often satisfy the federal health and safety requirements, but does not allow the companies to extract and act upon the meaningful data within those documents. By digitizing these documents, extracting data from them and analyzing the data as it is received, TraceGains can provide instant feedback based on the customer defined specifications. This helps reduce costs for the customer as well as the customer's suppliers all while ensuring that the suppliers are meeting the defined quality and safety standards. The ultimate goal for TraceGains is to enable continuous improvement of the food manufacturing process across the industry.

TraceGains has a great solution for their customers once the system is fully configured and they are able to begin utilizing the software as intended, however the initial configuration is currently a very cumbersome process. For each TraceGains customer, the initial configuration phase can take anywhere from two weeks to six months due to the fact that everything about the software can be completely tailored to fit the customer's business needs. There is currently no defined user interface to assist with the initial configuration process of a customer's suppliers and ingredients, which often takes about 33 percent of the time for an initial configuration. The vision for the project is that a new user interface can be designed and implemented to satisfy these needs and ultimately give the customer a way to configure their own suppliers and ingredients. If a user friendly interface can be designed to assist in the configuration phase for each new customer, this will likely free up several weeks of time for the internal staff to allow them to assist with other configuration tasks or customer training. This particular configuration utility is a small piece of the long term goal that will ultimately allow each customer to completely configure their own site.

II. Requirements

A. Functional Requirements

1. The utility must provide an interface for configuring suppliers and ingredients
2. The utility must provide an interface for associating suppliers and ingredients
3. The utility must provide a way to upload suppliers, ingredients and supplier/ingredient associations from a properly formatted Excel spreadsheet
4. The utility should provide the ability to associate multiple suppliers and ingredients at the same time
5. The utility must provide a way to delete suppliers, ingredients and supplier/ingredient associations
6. The utility should allow for inline editing of supplier and ingredient information on their respective pages
7. The utility should not commit any data to the database (including data from an Excel spreadsheet) until the user performs and confirms a commit
8. The utility should provide a method for exporting suppliers, ingredients and supplier/ingredient associations to an Excel spreadsheet
9. The utility must not allow for duplicate suppliers or ingredients to be created (based on hID)
10. The utility must not allow suppliers and ingredients to be associated more than once
11. The utility must not allow for suppliers or ingredients to be deleted if they are part of any supplier/ingredient association
12. The utility should provide sorting and filtering capabilities

B. Non-Functional Requirements

1. The utility must interface with the existing security and login framework
2. The utility should utilize the standard page template present in the existing web application
3. The utility must run under IIS 7.0
4. The utility should be written in C#
5. The utility must store data in the existing database with no schema changes
6. The utility should be fully supported on all IE6 – IE10 browsers
7. The utility should be fully supported on la 1024x768 resolution screen

III. System Architecture

The architecture, which was largely determined by the existing internal architecture, is outlined in Figure 2.1 of the Appendix. TraceGains customers can access the application login page from any web enabled device that supports an Internet Explorer browser and has a resolution of at least 1024x768. As the company has realized over the past couple of years the food manufacturing industry is often very slow to adopt new technologies and as such, many customers access the application from a browser as old as Internet Explorer 6.0. TraceGains has also realized, however that there are a significant number of managers and smaller more progressive companies in the industry that may also opt to use the latest and greatest technology which includes devices that support Internet Explorer 10.0. The future goal is to also support Firefox, Chrome and Safari internet browsers as TraceGains has realized that tablets and cell phones may very well be the future platform of choice for many of their customers and a majority of these smaller devices do not support Internet Explorer. This currently allows for only partial access to the application on mobile devices.

Rather than attempting to interface their website with Active Directory and enforcing login policies though the native Microsoft interface, TraceGains opted to manage all of their own user accounts and credentials in an internal database. Since the login page sits behind the external facing firewall the site must honor all valid web requests which include http requests on port 80 and https requests on port 443. This architecture requires that all account credentials are validated against logic built into the login page in order to determine which accounts are valid and which customer sites each user should have access to. Furthermore, anyone with a valid email address is allowed to register for a free TraceGains account, but the user should not have access to any customer sites until access is granted by a site administrator. In order to satisfy the requirement that the new utility interface with the existing login page and procedure, I had to build in a method for allowing site administrators to grant each existing TraceGains account with access to the customer site or the configuration utility or both.

Before the project began, each site that a user had been granted access to would show up on the login page after successful authentication. Upon selecting a site the user was redirected and automatically authenticated to the selected site. This allowed TraceGains to manage the user session exclusively through the login application and allowed each user to move between customer sites without being forced to log in to each site separately. In order to remain as consistent as possible I decided that each instance of the configuration utility would be configured just like all other customer instances. This decision enabled the user management for the new configuration utility to be maintained in an identical manner to that of all existing customer instances. Upon successful authentication to the login application, the user was presented with a list of logos and customer names which included all accessible instances of the configuration utility. The only distinguishing feature of the configuration utility was that the company name was appended with " – Configuration Utility". This required that the login application was able to redirect to either application using the same procedures and security protocols so that a single login application could be maintained and none of the existing functionality had to be modified.

The interface between the existing application and the database does raise some concerns; however the focus of the new configuration utility is not on the existing interface, but rather improving the new interface. Rather than attempting to manage the database connections and all of the associated code in the classes corresponding to each page I elected to maintain all of the database

connections within a single class. I took the approach of creating a database access layer to maintain and administer all of the connections for the new utility. With this approach any communication with the database can be completed in a single place. This makes it extremely easy to add consistent logging to the query execution and troubleshoot any connection issues including orphaned connections and failed connections. Even though the entire configuration utility is data driven and relies on the data from within the database there are no explicit connections made to the database from any of the web pages. The only communication with the database happens inside of the database access layer, explained in more detail in section four on technical design.

One final piece of the general system architecture outlined in Figure 2.1 of the Appendix is the interface between the Import Web Service and the new configuration utility. The Import Web Service is responsible exclusively for importing data from an Excel spreadsheet. Given an Excel spreadsheet, a selected worksheet and an object type the Import Web Service was designed to return a list of objects to the calling function. In order to encourage the use of the Import Web Service across all future utilities, I decided that the most effective method for implementing the new utility was through a web service that could be called from any internal application. There are several methods defined within the web service that are useful for managing the data import from an Excel spreadsheet that can be utilized by any calling function. The decision to maintain a separate web service for the import process centralizes the logic in order to reduce the maintenance overhead and aims to keep the logic consistent for every calling web page. The overarching theme across the entirety of the architecture for the new utility was that the solution should be easy to integrate with the existing architecture while providing a platform that encouraged future expansion to other utilities.

IV. Technical Design

During the design phase of the project, there was little room for modifications at the database level based on the requirements outlined by TraceGains, however the database schema, as outlined in Figure 3.1 of the Appendix, was very important to the success of the new configuration utility. In order to be successful the new utility was required to interface with the existing database in order to create new records and update or delete existing records while maintaining a format that was compatible with the existing application. Due to the fact that there are several different customers, each with a distinct configuration, the database relies heavily on associations with the customers table. Every table has an ID that corresponds to the ID of a record in the Customers table which allows all of the data for all customers to share a single database while maintaining a distinct dataset. In an effort to facilitate expansion and growth of the configuration utility, I created an object to closely match the schema outlined in the Suppliers, Ingredients and SupplierIngredientReference tables. The ultimate goal with this approach was to consolidate all of the database logic for each table into a corresponding object. This allows for all of the create, read, update, delete and validation functionality to be managed in a single location within a commonly referenced class.

In the existing application a majority of the database calls are maintained within the code behind for each web page. In my opinion this can result in inconsistencies between each database call and can often lead to subtle bugs which can be hard to track down. I determined that for this project it was best to create a database access layer that would be in charge of all database calls. This required that inside of each class or object, the T-SQL queries were built up and then passed into the DataAccessLayer object for execution. As illustrated in Figure 3.2 and Figure 3.3 of the Appendix, the DataAccessLayer builds upon native methods in the System.Data.SqlClient library which include the ability to return a data set, a data reader, a result status or the first column of the first row in the result set. For the purposes of this utility, these four methods were the entirety of what was required for all database interaction. Once the methods were developed and working, any valid T-SQL command could be added to the supporting classes and executed through the data layer encapsulating all of the connection management and query execution in a single place.

Originally, I had intended to have a single class corresponding to each database table for Suppliers, Ingredients and SupplierIngredientReferences that extended the DataAccessLayer; however, I found that in order to keep as much code as possible out of the individual web pages it was easier to create two classes corresponding to each database table. For the Suppliers table, the first class, as outlined in Figure 3.2 and Figure 3.3 of the Appendix, was the Supplier class which contained a corresponding attribute for each database column in the Suppliers table as well as Create(), Delete(), Update() and Validate() methods. In the context of the application, the Supplier object was used to manage a single supplier record. The second class was a Suppliers class which extended the database access layer and contained methods for managing all of the supplier records for a given company. The defined methods provided the ability to load a single supplier, load all suppliers, delete all suppliers, load all unassociated suppliers and load all associated suppliers given an ingredient. By maintaining a Suppliers class I was able to manage or return an entire list of supplier objects which became the primary data source for the SupplierConfig page. Similarly I developed an Ingredient class and an Ingredients class that were used in the IngredientConfig page as well as a SupplierIngredientReference class and a SupplierIngredientReferences class that were used in the SupplierIngredientReferenceConfig page.

Once all of the objects and a majority of the corresponding methods were defined, I began work on the user interface for the project. For the purposes of this utility, I added three new pages that were designed to interface with the existing architecture. All three of the pages inherited the existing master page and base page so that each could take advantage of a consistent layout as well as all of the logic included in these pages for managing the customer information and account information. As outlined in Figure 3.2 and Figure 3.3 of the Appendix, a SupplierConfig page was defined for managing suppliers, an IngredientConfig page was defined for managing ingredients and a SupplierIngredientReferenceConfig page was defined for managing all of the associations between suppliers and ingredients. Based on the requirements provided by TraceGains, each page allowed for the user to create new data and edit or delete any of the existing data.

Using the SupplierConfig page as an example, the page would first call the Suppliers class to retrieve a list of all of the currently active suppliers in the system. The list of suppliers would then become the data source for the editable grid on the SupplierConfig page where each row in the grid corresponded to a Supplier object. One of the main functional requirements for the project was that the grid would support inline editing such that the user could edit the data relating to any supplier directly in the grid. The other major functional requirement for each page was that the grid would support filtering and sorting of the data. Keeping these two requirements in mind, I elected to build up the data grids on each page using Telerik controls. The basic RadGrid control for data grids natively supports inline editing, sorting and filtering which made the Telerik controls an easy implementation choice. The other benefit of using Telerik controls is that the remainder of the application was built using Telerik controls which allowed for a more consistent look and feel across the application.

After the corresponding page had been loaded the user was provided with a method for editing any of the existing suppliers directly in the grid, creating new suppliers, deleting existing suppliers, exporting all of the suppliers to an Excel spreasheet or uploading suppliers in bulk from an Excel spreadsheet. The Telerik RadGrid provided all of the inline editing functionality, while the inherited master page provided command buttons for the ‘Save’, ‘Create’, ‘Delete’, ‘Import From Excel’ and ‘Export From Excel’ events. These inherited command buttons required that a corresponding event function be defined in the code behind for the page, which made it very easy to build in custom logic for each event and also provided a consistent place to perform session state management on each server post back. Each of the aforementioned event functions first called the UpdateSession() method in order to update all of the objects maintained in session state and then made the appropriate call into the DataAccessLayer to complete the operation.

Aside from the basic functionality of the new configuration utility, one of the main requests from TraceGains was that no data was committed to the database until the user performed and confirmed a save operation. During an event generated by the web page, a post back to the server is generally completed in order to perform data processing and manipulation. Over the course of this process, the existing data is typically released from memory and the page is reloaded. As a way to maintain the user modified data, I elected to store all of the objects for the current page in session state so that they were available after the page reload. As mentioned above, each of the new configuration pages had a method specifically for updating session state on a post back to the server. Within this method, all of the original data and accompanying user updates from the editable grid were mapped to a list of corresponding objects. Once the list of objects was prepared, it was added to session state and then reloaded during the subsequent page load event and used as the data source for the editable grid.

The final major piece of functionality requested by TraceGains was the ability to load Suppliers, Ingredients and their associations from a properly formatted Excel spreadsheet. In order to facilitate expansion of the configuration utility to other areas of the infrastructure, I decided to create a generic ImportLibrary web service and a corresponding ObjectType enumeration. In conjunction the standalone import web service and object enumeration were developed to handle the entire Excel import procedure for every type of enumerated object. This structure ultimately allowed the Excel import to be contained within a single web service call while allowing the proper objects to be created and passed back to the calling function. In each new configuration page, the 'Import From Excel' button was wired up to an event function that launched an ImportLibrary dialogue box. Within the dialogue the user would select the Excel worksheet to import and then pass the spreadsheet off to the import web service for completion.

Rather than attempting to validate the objects as they were being created inside of the web service, I opted to simply remove all of the columns in the selected worksheet that did not correspond to any of the class attributes, create the requested objects and pass them back to the calling function. Instead I relied on the validation function within the Supplier class, Ingredient class and SupplierIngredientReference class that was called by the create and update methods designed to validate the object before committing any changes to the database. Initially I thought that the Excel import functionality was going to be difficult to develop and maintain; however managing the import in a standalone web service turned out to be a very good decision. Much like the database access layer, I was able to manage and maintain all of the necessary components within a single location. I also found that using the System.Data.Oledb library provided a nearly identical management interface for Excel spreadsheets that the System.Data.SqlClient library provided for managing SQL Server connections.

V. Design & Implementation Decisions

Throughout the course of the project, many of the design decisions were made knowing that the new utility was required to interface seamlessly with the existing internal infrastructure. In order to facilitate ease of use and rapid integration with the existing implementation process, no changes were allowed to the existing database schema or existing code base. There were also a number of existing software support agreements between TraceGains and their customers which specified supported browsers and screen sizes. Ultimately this meant that the new utility was allowed to utilize any existing functionality and was required to act as a standalone module to be interfaced with the existing software in order to enhance the user experience. These restrictions significantly limited a number of design decisions and also introducted a number of interesting implementation issues along the way.

When deciding what languages to use for the project, TraceGains explicitly stated that everything in the user interface and front end database access layer was to be written in C# using either ASP.NET controls or Telerik controls as necessary. This requirement was primarily due to the fact that all software solutions were required to run under IIS on a Windows Server instance and in part because all existing software was written in C# using either native Microsoft components or third party Telerik components. In order to maintain 100 percent compatibility between the applications and allow the rest of the development team to maintain the utility moving forward, it was best to maintain a common set of languages across both applications. For the same reason, the requirement for database development was that all database queries were to be written in T-SQL and interfaced with the existing SQL Server instance.

Based on the requirement to use C# as the primary language, I still found that there was a significant amount of flexibility in the design decisions relating to the new utility. One of the first major decisions was whether or not to use an object oriented approach for the project. Most of the existing application was written in a non-object oriented manner where each page was a class and encapsulated all of the necessary code. In general, there were no objects being passed between pages and raw data was generally manipulated rather than performing manipulation against objects. Ultimately this was due to the comfort level of many of the developers with a non-object oriented approach. While there was a case to be made for remaining consistent, I felt that it was best to adopt an object oriented approach for the project in order to make the code cleaner, easier to maintain and more portable to other platforms and future configuration utilities. Initially I started with a non-object oriented approach and found that it was very difficult to interface the Suppliers and Ingredients particularly when attempting to associate the two. It was must easier and much cleaner to adopt a Supplier object and an Ingredient object and simply provide a mechanism for interfacing the two.

Another one of the other major changes that I adopted from the existing application was the presence of a database access layer. In the existing application nearly all of the database calls and database queries were coded directly in to the web page code behind files. While this approach maintains all of the code for a single page in one place, I felt that it was messy, inconsistent and often resulted in a significant amount of duplicated code. When implementing the database access layer I found that it was very natural to build up the necessary T-SQL queries inside of the supplier or ingredient objects and then pass the queries off to a database access layer. Within the SQLClient libraries in C# there are a number of different methods for executing database queries, so I initially found it very challenging to accommodate all of the different scenarios that I would need for the project,

but once the functions were implemented, this approach made the database access easy and consistent across all of the new pages.

Once a majority of the back end was developed and ready to go, the rest of the development effort was spent meeting the criteria outlined for the user interface. The first design decision relating to the user interface was which set of controls to adopt for maintaining the various web forms. Initially I had decided to use the ASP.NET controls due to the fact that they are native to Microsoft and are significantly faster within a web browser in comparison to Telerik controls. Implementing the data grids was very easy using the standard ASP.NET controls; however a number of the other requirements became too complicated in the time frame allotted without using a set of third-party controls. The number one reason for switching over to Telerik controls was the ability for the user to perform inline editing. The ASP.NET data grid provides a mechanism for editing one row at a time, but requires user interaction in order enable editing on each row. Telerik on the other hand provides an editable grid that is much like an Excel spreadsheet where the user can click in to any editable cell and immediately edit the contents of the cell. After discussing the two options with TraceGains, the management team unanimously agreed that the spreadsheet approach provided by Telerik far outweighed any performance differences in the two controls.

By far the biggest challenge that the project presented was maintaining all of the user's data and updates while meeting the specification that nothing was to be committed to the database until the user performed a commit. This meant that hundreds of existing records could be updated and hundreds of new records could be created in the UI, but never committed. In particular I found that maintaining the data modified by the user in the front end was most difficult across a post-back to the server. Based on feedback from other developers and some research about the Microsoft recommended approach, I found that there were two viable solutions. The first was to develop a web service that could be called by JavaScript or AJAX which would allow server side processing without a full post-back. The second approach was to rely heavily on session state and ensure that all of the objects were maintained in session state during a post-back. Ultimately I found that maintaining and debugging all of the JavaScript functions necessary to support the web service calls quickly became cumbersome and cluttered, so I decided to rely more heavily on session state to meet a majority of this requirement. The biggest hurdle was ensuring that all of the data in the user interface was committed to session state immediately on a post-back to the server in order to maintain the current state. Once I finalized a method for saving everything in to session state, this problem became quite trivial even during the validation process and the Excel import process as the collection of objects could be loaded from session state and manipulated or added to with ease.

VI. Results

At the end of the project all of the functional and non-functional requirements had been met. Combined with other internal TraceGains staff, I spent the last week of the project testing the requirements and functionality of the new configuration utility. The company was in the process of implementing several new customers at the time and utility immediately became critical to the new customer provisioning and implementation process. Due to the fact that this utility was initially designed to assist internal staff the utility was adopted immediately once the QA department had signed off and the utility was released to a production environment. Aside from the basic ability to manage suppliers and ingredients, the feature of the utility that was deemed the most important by the implementation staff was the ability to mass upload suppliers, ingredients and their associations. Over the course of many customer implementations TraceGains realized that their customers, almost exclusively, kept records of their suppliers, ingredients and their associations in a digital format such as an Excel spreadsheet. Ultimately the implementation staff felt that the single largest waste of time during the configuration of suppliers and ingredients was extracting the data from the spreadsheets and manually performing the configuration steps using the customer facing interface which requires that someone manually enter the data for each supplier and each ingredient one at a time. Even though there were often a number of changes that had to be made to the lists of supplier and ingredients provided by the customer in order to match the database schema, the general consensus was that the new utility would eliminate nearly 30 percent of the typical configuration time for a new customer.

Due to the immediate impact that the utility had on the internal implementation staff and the perceived ease of use that the utility presented, the goal for TraceGains is to begin providing customers with access to the utility within the next couple of months. Nearly one third of the total development time for the utility was spent on error handling and error checking to ensure that the user was not configuring any of the suppliers or ingredients in error. The software which is maintained and provided by TraceGains is entirely configurable to the customer's needs and as such one of the main goals of any configuration utility provided by TraceGains is to build in enough of the business intelligence so that the software can prevent any configuration errors.

The supplier and ingredient configuration utility was the first of a series of planned utilities aimed at minimizing the overhead associated with new customer implementations for TraceGains. Using the knowledge obtained over the first two years in dealing with food manufacturing companies, TraceGains has acquired a good understanding of the format in which their customers currently store their data and how that data will be provided in order to perform the initial configuration. Based on feedback from the customers and internal implementation staff, the long term goal of the project is to use the framework outlined by this utility to expand support for customer self-configuration utilities. Future plans include support for the configuration of business logic, electronic notifications, user groups, users and customer documentation.

Appendix

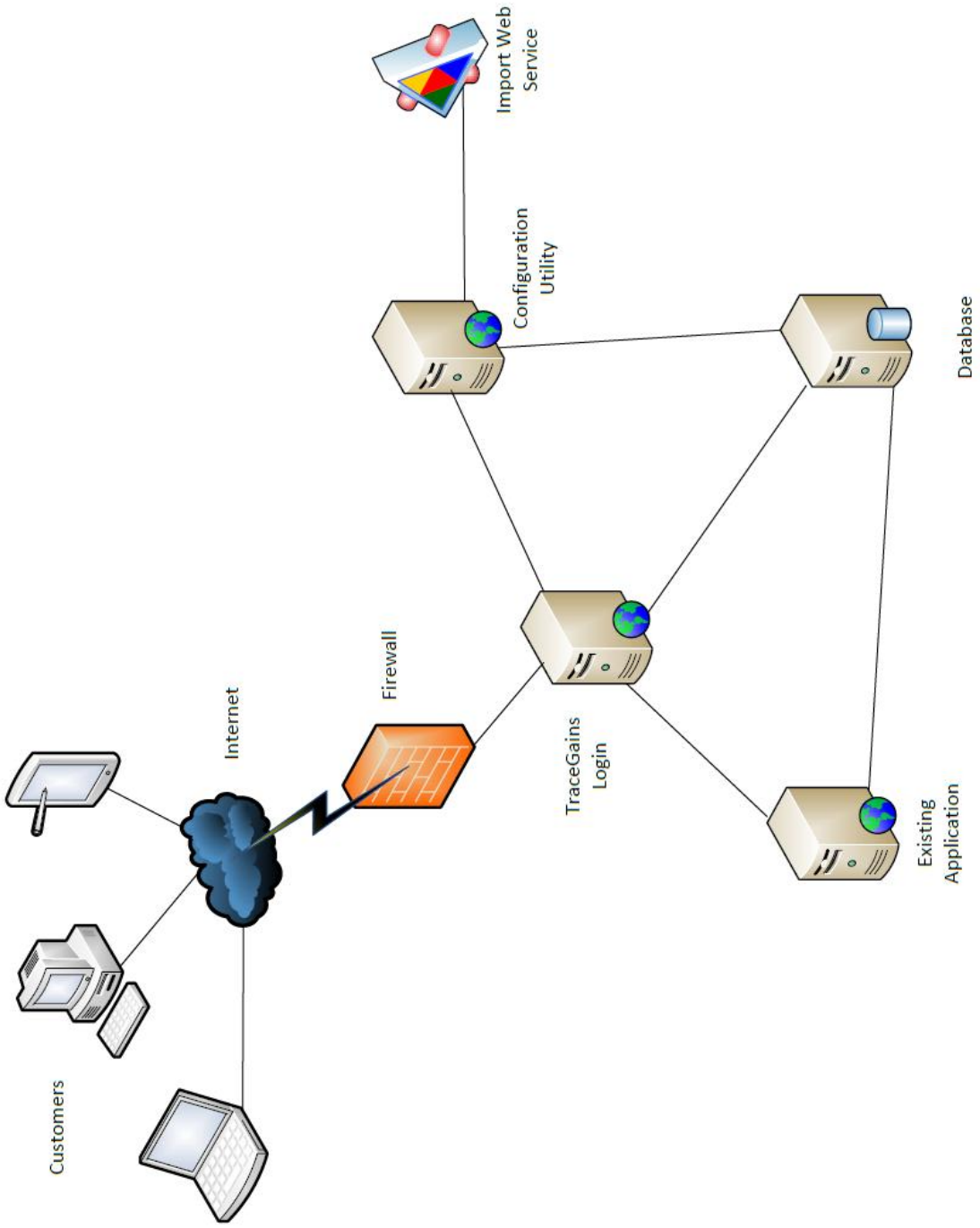

Figure 2.1: System Architecture

Figure 3.1: Database Schema

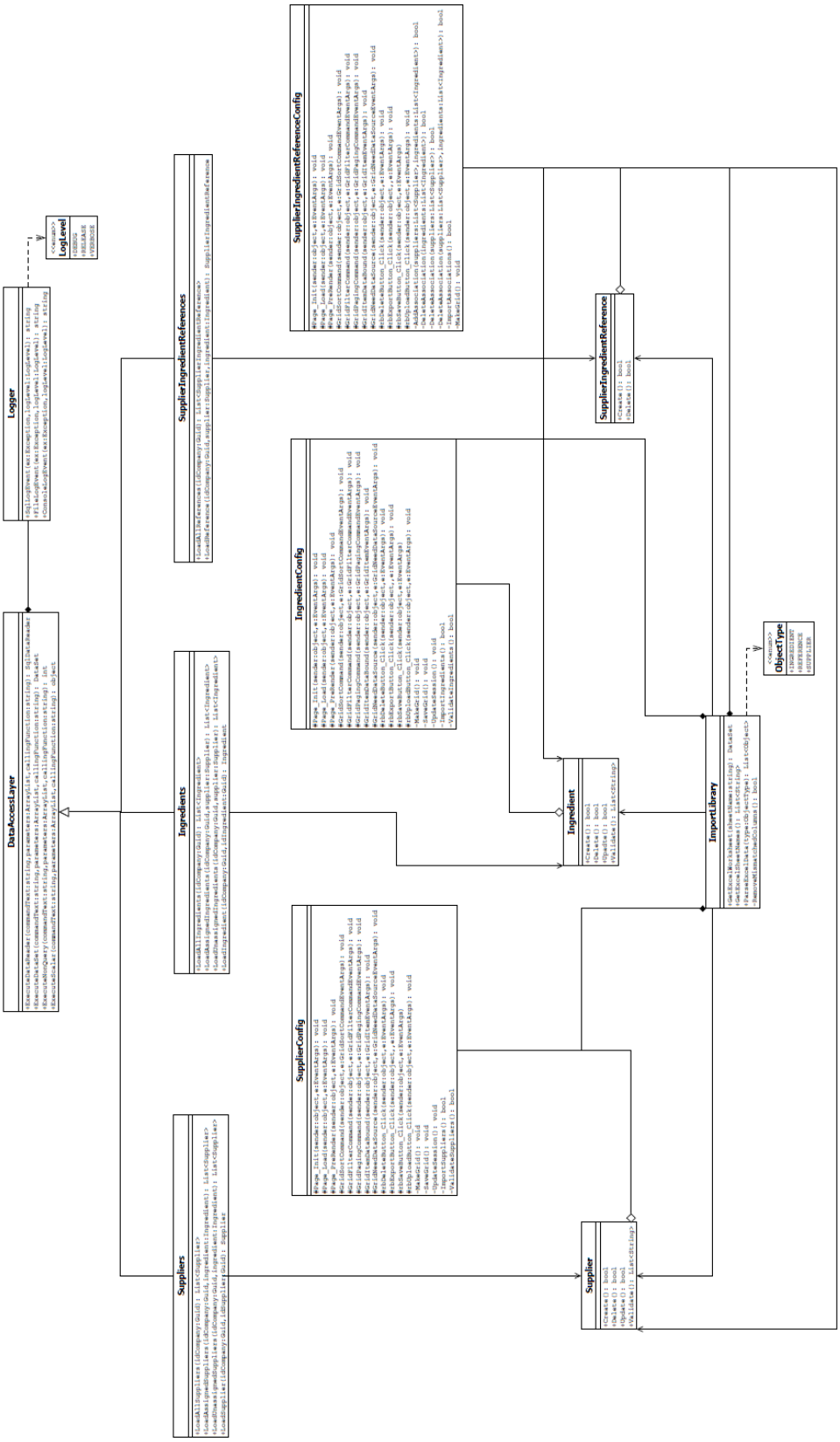

Figure 3.2: Application UML With Methods

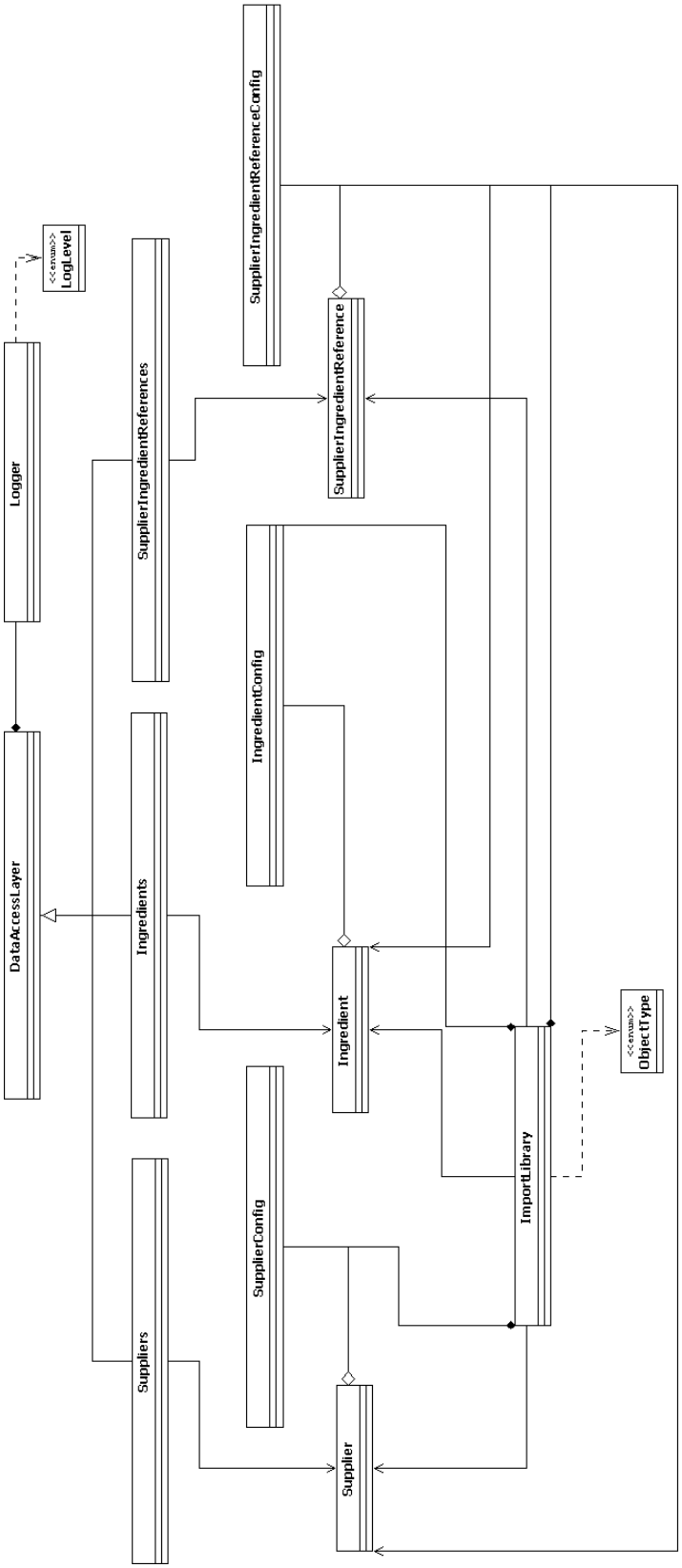

Figure 3.3: Redacted Application UML