Volume

1

Growthware

Application Design

growthware

Application Design

© Growthware

<http://sourceforge.net/projects/bedrockgrowth/?source=directory>

Fitchburg Ma, 01420

Emal: maui.regan@gmail.com

Table of Contents

[Overall design 2](#_Toc187471083)

[Security (A closer look) 4](#_Toc187471084)

[GrowthWare.Framework (A closer look) 5](#_Toc187471085)

[GrowthWare.Framework.Model (what is this) 7](#_Toc187471086)

[GrowthWare.Framework.Web 8](#_Toc187471087)

[Purpose of the Core Web Application 10](#_Toc187471088)

[Security 11](#_Toc187471089)

[What are Function/Pages 12](#_Toc187471090)

[What are Clients/Accounts 12](#_Toc187471091)

[Roles 12](#_Toc187471092)

[Groups 12](#_Toc187471093)

[Security Entity and Security 12](#_Toc187471094)

[Logging 13](#_Toc187471095)

[Navigation 13](#_Toc187471096)

[Skinning 13](#_Toc187471097)

[Data Access 13](#_Toc187471098)

[Application flow 14](#_Toc187471099)

[How To Technical 16](#_Toc187471100)

[Examples of Security 16](#_Toc187471101)

[Adding a new Module/Page 17](#_Toc187471102)

[Adding a new skin 17](#_Toc187471103)

[To create a skin: 17](#_Toc187471104)

[To skin a single control: 17](#_Toc187471105)

[Summary 19](#_Toc187471106)

Chapter

1

Overall design

In this chapter you’ll find information pertaining what is being solved and the overall design considerations for the Growthware framework and core web application.

B

efore speaking of design let’s take a look at what problem we are attempting to solve. As developers we know that code reuse is among the best ways to deliver quality, and not only quality but to deliver a project quickly with that quality. To my way of thinking coming up with common features that end up across projects is the way to accomplish the quality. In a rudimentary sense, it’s what can be “reused” from project to project. My answer was, well, security, as in logging in and dealing with what happens after said login. Specifically what I’m talking about is Authentication and then Authorization. Most projects that I have run across deal with Authentication/Authorization in one way or the other and this is what Growthware is meant to solve in a generic way.

Compared to Authorization, Authentication is the simpler of the two. Authentication is given some information we verify that you are who you say you are, or in others account/password pair verification.

Authorization on the other hand is all about, now that we have some sort of Authentication what resources are available? I am not about re-inventing the wheel here so I considered how other systems/applications do this like Windows NT (LAN-manager), and Netware. Both use roles and groups to accomplish this by assigning roles and groups to accounts and to resources.

Roles and groups with regards to resources isn’t quite enough information, we need to apply the roles and groups to permission of the resource. It made sense to think of a resource in a similar way a database relates to data, what can we do to data in a database? When you think about it there are really only 4 things that can be done to data in a database and this is Select, Insert, Update and Delete. Permissions are taken pretty much straight from what can be performed to the data, but, with slightly different names. I chose View, Add, Edit and Delete.

To add in one more twist to the Authentication/Authorization I wanted an organization or department to be able to define what a given role could do. Why, people in general have slightly different view/understanding of what a given word means. Example “Administrator” means something to someone in HR but a completely different meaning for someone in IT.

Authentication and Authorization becomes the core of what Growthware will provide, but, there are other features that are common now that there is Authentication/Authorization:

* Access to data
  + Maintenance of data
* UI/API
  + Login/Logoff
  + Administration
    - Roles
    - Group
    - Account
      * Registration
    - Resources
* Logging

Beyond Authentication/Authorization I did not want to tie Growthware to a specific database so you’ll find code to accommodate that requirement.

The .Net Core version of Growthware has many of the same specification requirements as did the original version, and yes a not insignificant reuse (copy paste) of the code because of this. With the change to .Net core I did separate out the API from the UI so that we could support any UI we wanted. The overall design is a Monolith specifically N-Tier.

The following is a list of the projects involved in Growthware:

* BusinessLogic – Contains the code to implement the business logic and is the separation other projects from the type of database. Depends on DataAccess and Framework
* DataAccess – Contains the code to actually access the database for each database supported IE SQL Server, Oracle, etc. Note that no other projects will access the database. Depends on Framework
* Database – Contains all of the “current” DML for in this case a SQL Server database
* DatabaseManager – Serves as method to upgrade/downgrade the database using C# and database specific upgrade/downgrade scripts. Depends on DataAccess and Framework. The ReadMe.md file is worth looking at
* Framework – Contains code that is to be used by other projects and should have no dependencies on any other project. The type of code represented here is any type of Model object, Enumerations, and Interfaces. Utilities like crypto, logging, object factory, and configuration settings.
* Web.Angular – The Angular UI implementation should have no dependencies on any other projects
* Web.Api – Is very light and serves mostly as example of someone that wants to use some or all of Growthware would accomplish that. Here you’ll find the Controllers but they pretty much only inherit from a base class from Web.Support. Depends on Web.Support
* Web.Support – Implements any logic required by Web.Api through the BussinessLogic, is responsible for enforcing Authorization, and any optimization needed. Depends on BusinessLogic and Framwork

# GrowthWare.Framework (A closer look)

The framework incorporates two major areas of functionality, the “Business Logic Layer” or BLL and the “Data Access Layer” or DAL. The third but smaller area of functionality is the “Common” code section. Each of the areas of functionality are separated into an appropriate namespace.

I will not cover direct implementations of code in this section, but, rather the underlying design concepts.

One of the overlying design concepts here is derived from the design of the database, so just to note that the database design basically implements a set and get methodology. Meaning that there are stored procedures to get the data and stored procedures to set the data. The get type of stored procedure are exactly what they sound like they retrieve data. A set type of stored procedure does exactly what it sounds like it will perform and insert or an update of data. The implementation of a set stored procedure reduces the number of stored procedures needed, but, increases the complexity of each of them.

As stated before the BLL requirements is where the data store (database technology) independent as possible, this accomplished through the factory pattern and implemented with the FactoryObject from the Common namespace.

Each BLL classed has two basic common implementation structures. The first structure is the instantiation of the class, there is no “new” without parameters and the second is that the new with the parameters always include the “MSecurityEntityProfile” model object and a Boolean indicating a central management environment.

The “MSecurityEntityProfile” model object provides the connection information and the SecurtyEntityId that is necessary for most every administration task. The central management Boolean will determine if the DAL object will be re-created for every use. Passing true will re-create the DAL object with each use. Re-creating the DAL object will allow the BLL to administrate different data stores, but, decreases performance. Passing false transversely allows the DAL object to be created once and stay in memory as long as the BLL object does and saves the overhead of creating an object each time the DAL is needed.

So why go through this effort of central management, it’s simple you may have many implementations of the data base structure but want nothing to do with the rest of the projects. You may even be using another technology all together for your application, say java for example. When none of the rest of the framework is desired then you can manage those database with a single working instance of the CWA.

Let’s take a look at some code for a moment by opening the BAccounts.XX where XX is language you’re working with (.vb or .cs). So you can see how this works, there is a private DAL object at the top of the class and it’s given an instance in the “new” constructor. Notice that the private declaration of the class preventing the creation of an instance without passing any parameters.

With the private instance of the DAL object being created you can see from the use of the FactoryObject class that the DAL code can be separate for each type of data store you’re using. Currently there are provisions for SQL server versions 2000 and 2005 or greater as well as for Oracle. At the writing of this document code has only been fully implemented for SQL 2005 or greater. SQL Server 2000 and Oracle may be done at a later time.

Looking a bit deeper into the DAL code open the DAccounts.XX file where XX is the language you’re using (.vb or .cs) in the “GrowthWare.Framework.DataAccessLayer.SQLServer.V2008” namespace/file structure.

Take notice here of the use of interfaces this is what allows the BLL to use the many types of data stores without having to change any BLL code. So each of the DAL objects have to implement a corresponding interface. Now you can start writing the code necessary to interact with your data store of choice.

Each of the DAL classes that interacts with the data store does so through inheriting from an abstract class called “DDBInteraction”. It is within the “DDBInteraction” class the actual interaction to the data store happens. One should take note that for each type of data store that will be sued there will be a corresponding “DDBInteraction” abstract class.

Stating that the “DDBInteraction” class is what interacts with the data store “technically” accurate. For example for SQL server actually uses the Microsoft Enterprise Library to do the work in the data store. Making the statement at the “DDBInteraction” does the interaction of the data store when it relay’s on yet another layer is just quibbling over semantics – after all ADO.Net is really the same sort of thing, it’s the concept that’s important not the implementation.

For the purpose of security any of Growthware’s code will only use stored procedures to retrieve/insert/update data in an effort to avoid SQL injection. The framework itself will facilitate execution of SQL but does not.

To recap:

1. The BLL defines a private object through an interface that it’s going to work with
2. The BLL then creates an instance of the object using the FactoryObject using information defined in the “MSecurityEntityProfile” model object
3. The DAL class inherits from DDBInteraction class that performs all data store interaction

# GrowthWare.Framework.Model (what is this)

The Model objects project contains the Enumeration and Model Objects.

Enumerations are exactly that, not much to say and I don’t want to debate the validly of their usage.

Here we are going to take some time to expand just a bit on the earlier statement that they are a way simplify passing parameters. Simplifying passing parameters is the main purpose of the model objects but they also a representation of data, that is to say they work on a two way street. Model objects do both, pass to, and pass back. When being used to pass back information the BLL will get a row of data from the DLL, pass that into the “new” constructor of the Model object and there you go a fully populated object that represents the data.

Again inheritance is used to propagate common properties and or methods. So the database design pokes it head in and influences the code design. What does that mean? Well most every table in the database design had an identification, Name, Added by, Added date, Updated by, and Updated date columns. In order to populate these properties when a row of data is passed to the “new” constructor commonly needed methods are also helpful.

Not all objects that are ever going to be created, will need the above mentioned columns. To that end, there is more than one level of inheritance happening.

So structurally there is the “MBaseProfile” of which the “MProfile” inherits then the specific model object inherits from the “MProfile”.

The “MBaseProfile” abstract class provides the common population methods GetBool, GetDateTime, GetInt, and GetString. The common methods do the mundane things such as column existence and that it’s not null before setting the property. Assuming that your reading to help you understand what’s been done and how you may use it this is a point where you can use the “MBaseProfile” for your own objects.

The “MProfile” abstract class contains the common properties that are used widely throughout the Growthware product.

# GrowthWare.Framework.Web

Here is where all that logic and code to facilitate a working web based application/site is organized. OK what did that mean? When creating a .Net web application/site there are performance considerations that need to be addressed. Considerations like caching, application context, and of course the decoupling of code from the “presentation” layer.

Caching – well this is a subject that can go on, for like ever. Simply I needed a way to allow an application to use cache and to simplify the built in .Net facilities. To accomplish caching in the manner I needed the façade pattern was used. The goal being the ability maintain common application cache within a say a web farm. The “CacheController” is the implementation of that goal. Briefly … if you setup directory replication (free with windows) for the “CacheDependency” directory in the web application/site across the web farm, then the CacheController will maintain synchronization with in that web farm.

Context – HttpContext is maintained in two HttpModules, the HttpContextModule and the ClientChoicesHttpModule.

The HttpContextModule performs the following:

1. Initiates the ClientChoicesHttpModule
2. Error logging on an application level in the “Error” and “EndRequest” methods
3. Aids in maintaining the common cache mentioned in Caching in the “BeginRequest” method
4. Enforces security during the “AcquireRequestState” method

The ClientChoicesHttpModule performs the following:

1. Maintains the client choices within the context in the “AcquireRequestState” method
2. Ensures the client choices are saved to the data store in the “End Request” method

The bulk of access to the framework is provided through the “Utility” objects and are mostly static classes. The exception to the static classes would be the LogUtility, this is an implementation of the singleton pattern and didn’t need to be static seeing that there could only be one to begin with.

There are utility classes/objects for the following areas:

1. Accounts
   1. AccountUtility – Management of the account information kept in the data store.
   2. ClientChoicesUtility – Management of the choices a client (account/user) may make using the application.
2. Security
   1. RoleUtility – Management of roles kept in the data store.
   2. GroupUtility – Management of groups kept in the data store.
3. Functions
   1. FunctionUtility – Management of functions kept in the data store.
   2. FunctionTypeUtility – Management of the function types kept in the data store
4. Security Entities
   1. SecurityEntityUtility -- Management of the security entities kept in the data store
5. Messaging
   1. MessageUtilty – Management of messages kept in the data store.
   2. NotityUtility – Façade to messaging facilities such as .Net Mail. (maybe expanded to handle notification to get application use)
6. Files
   1. FileUtility – Allows for the manipulation of files.
   2. DirectoryUtility (related to both Functions and Files). Allows for management of directory information kept in the data store that is associated with functions.
7. Menus
   1. MenuUtility – Retrieves and builds xml given an account and security entity.
8. Logging
   1. LogUtility – Provide logging for the application.
9. Name value pairs
   1. NameValuePairUtility – Management of the NVP kept in the data store.

You will also find the base code for a page (inherited by ClientChoicesPage), though this is not needed as much as it was in the past and will likely me deprecated shortly because full page post back is no longer done. The original design for the base page was to handle session information on the server cutting down on the amount of data that needed to be transferred through the web.

GWWebHelper – A place to keep application constants and the like as well as writing information to the Web.Config.

WebConfigSettings – Allows read access to information in the Web.Config file.

So that about covers (lightly) what’s in the GrowthWare.Framework.Web project. Please look at the code for further details about any of the functionality you’ll find here.

# Purpose of the Core Web Application

The Core Web Application (CWA) serves as the foundation of a web site/application providing security, menus, navigation, and content delivery. The CWA is also the administration point for the following:

* Account Management
  + Account creation
  + Account edit (including the ability for each client to edit there own information)
  + Account authorization
  + Security Management through roles and groups
* The Security Entity Management
  + The ability to choose the desired Security Entity to work with.
  + Create roles and groups for each Security Entity
  + Manage properties of a Security Entity
* Role management
  + Adding a role
  + Editing a role
  + Deleting a role
* Group management
  + Adding Groups
  + Editing Groups
  + Deleting groups
  + Associating Roles to Groups

**Note:** Role management is by Security Entity, roles are assigned to Page/Modules for 4 “Rights” (View, Add, Edit, and Delete). Roles are also assigned to Clients/Accounts. When a role has been assigned to the Module/Page permission and the account, the access rights are determined programmatically by asking the question, does any the account roles match any one of the roles for each of the permissions? The role based security is accumulative between the “All” Security Entity and the selected Security Entity, however, this could be changed in the store procedures to be recursive thus allowing for any depth of security inheritance.

* Group management
  + Adding a group
  + Editing a group
  + Deleting a group

**Note:** Group management like role management is by Security Entity, roles are assigned to groups and then groups are assigned to Page/Modules for 4 “Rights” (View, Add, Edit, and Delete). Groups are also assigned to Clients/Accounts.

Roles are obtained through union queries in the database that get roles directly applied and by roles applied via any groups.

* Function/Page Management
  + Add Functions/Pages
  + Edit Functions /Pages
  + Delete Functions /Pages
  + Security Management – Assigning roles by Security Entity’s
* Client Choice Management
  + As a client makes choices in the application the information is put into the data store and retrieved when necessary.

**Note:** Client Choice Management of the application will need to be altered to some degree for each project as projects will define the information to be associated with client’s choices. Client Choice Management is primarily handled an HTTP Module in conjunction with the MClientChoices. The HTTP module makes the client choices available to every Function/Page that inherits the client choices control or page module. The MClientChoices object aids in retrieving keys from the ClientChoicesState object. Adding to the client choices will require changes in the data stores store procedures, since this is the primary controlling factor.

# Security

To understand the root of the role/group security we only need to look at the construct of a single class. The “MSecurityInfo” class found in the “Framework” project is the driving force behind it all. MSecurityInfo simply accepts two objects that adhere to the “IGroupRolePermissionSecurity” and “IGroupRoleSecurity” interfaces found in the same project.

The “IGroupRolePermissionSecurity” ensures that the object contains properties “Collection<string>” for groups and roles that are assigned to the object for each “permission” as well as the derived roles

Security for Growthware consists of three objects:

1. The first being the MAccountProfile, this contains the roles that have be either assigned directly or by association of groups.
2. The MFunctionProfile, this contains the roles that have either been assigned directory or by association of groups for 4 “Permissions”. Currently the permissions are View, Add, Edit, and Delete.
3. The MSecurityInfo object, this populates 4 “Permission values” of MayView, MayAdd, MayEdit and MayDelete. When an instance of the MSecurityInfo is create an object that implements IMSecurityInfo is and the MAccountProfile. DerivedRoles

Implementation of security is performed in the AcquireRequestState handler of the HttpContextModule call found in GrowthWare Framework Web name space. Each request for a page/function will instantiate an MSecurityInfo object and place it into the HttpContext. Logic is executed during the AccquireRequestState to determine if the needs to change their password (forced), if they are allowed to view the request or if their “session” has expired and a logon is required. Growthware does not use HttpContext.Current.Session, due to the nature of ajax request, so HttpContext.Current.Cache is used to maintain information that would normally be contained in HttpContext.Current.Session.

# What are Function/Pages

Functions are primarily a holder of security information that can be a number of things:

1. Page, in the same vain as any other .Net page using the familiar Front/Page/Controller pattern. HTML is in the Front or ASPX portion, coupled with a “code behind”.
2. Security item

To obtain the security information for a module/page you create an instance of the “ModuleProfileInfo” and populate it with the module profile information from the “AppModulesUtility”. Once you have the module profile information get an instance of the “ClientInfo” MODEL.ClientInfo data structure. The “ClientInfo” object will take the roles from the module profile information and match them to the client information to produce “MayView”, “MayAdd”, “MayEdit”, and “MayDelete” properties for the model/page. You can now use theses properties for what will or will not appear in the UI of the module with the “ASPX” or “ASCX’ code behind.

Modules being what they are in relationship to the system needed special consideration made with regards to performance. If you take a look in the “GLOBAL.ASAX.VB” you’ll find that they are loaded into cache as a collection. Again this is to aid in performance, modules won’t change much during the normal run time of the application. Modules will generally be added or changed during design time, so with that in mind they are loaded into cache when the web application starts and should be referenced though out the application from the “ModuleCollection”.

# What are Clients/Accounts

Clients/Accounts are used in conjunction with roles to control access to any module/page within the application and are defined by the class “ClientProfileInfo” in the DALModel project. It is important to note that there is an Anonymous account, role, and Security Entity to allow access to any portion of the application anonymously.

# Roles

Roles can be created and assigned by each Security Entity. Three roles exist for the system SysAdmin, Anonymous, Super User, and Authenticated.

# Groups

Groups can be created and assigned by each Security Entity. Roles are assigned to groups and are returned through union queries from the data store.

# Security Entity and Security

Roles, groups and modules/pages are always managed by Security Entity entities. The “All” Security Entity is used to denote generic security inheritance, doing this give a new Security Entity the global security settings. Granting access to any given Security Entity is done by adding an account to any role/group for that Security Entity.

# Logging

Logging is provide by LOG4NET and is exposed through a singleton class name “AppLogger”.

“AppLogger” exposes the normal logging of LOG4NET such as logging debug through fatal errors, as well as, setting the threshold programmatically. The “AppLogger” uses the SYSTEM.REFLECTION and the SYSTEM.DIAGNOSTICS name spaces to retrieve the calling class of the “AppLogger”.

# Navigation

Navigation is the portion of the application that implements the “C” of MVC the most, where the code has no attachment to what needs to be displayed. All navigation flows through the “NavControler”. It is the “NavControler that will determine how and what is to be done based on information of the module requested by the code behinds. The NavControler has two signatures for the NavTo method, both of which require the “Action” to be passed. Information is obtained about the module or page based on the “Action” and the appropriate response is to be made.

# Skinning

Skinning begins with the root DEFAULT.ASPX code behind. Each Security Entity has a skin property that defines which look and feel will be used. Each look and feel correlates to a subdirectory under the UI directory. When the DEFAULT.ASPX page loads and executes the code behind retrieves the theme information about the Security Entity from the clients selected Security Entity and the proper “DEFAULT.ASCX” user control is loaded into the page. Each user control contains the layout and presentation information for a particular theme and further loads the user controls that do the actual work. Each individual user control can also be skinned.

# Data Access

Data access within the application is accomplished through abstraction and use of the factory pattern. The process to access data goes as follows:

In many cases there has been a “utility” class created but this is not necessary to keep with in the overall design of the application.

First a class from the BLL projects is invoked example:

BBusinessUnits.GetAllRolesForBusinessUnit(myBusinessUnitRoles, Business\_Unit\_Seq\_ID). When you examine the “BBusinessUnits” class from the BLL one of the first things to note is that the DALFactory and DALInterface are imported. Importing the two namespaces allows for a call similar to the following:

Private Shared iDal As IBusinessUnits = FBusinessUnits.Create

The above call is creating an instance of the DBusinessUnits in the DAL using the interface and the factory. To see how this is done examine the following snippet of code:

Public Shared Function Create() As IBusinessUnits

' Look up the DAL implementation we should be using

Dim path As String = System.Configuration.ConfigurationSettings.AppSettings("BaseDAL")

Dim className As String = path & ".DBusinessUnits"

Dim retVal As Object

' Using the evidence given in the config file load the

' appropriate assembly and class

retVal = CType(System.Reflection.Assembly.Load(path).CreateInstance(className), IBusinessUnits)

Return retVal

End Function

The above code will now expose any of the public functions or sub routines in the DALInterface for the actual code that connects to the data store. The web.config file contains the name of the assembly and in this case the value is “DALSQLServer”. In the end the specific object is loaded from the assembly and the return. When you look at the DALSQLServer you notice that it implements all of the public functions and sub routines in the DALInterface.

In an attempt to keep things separated the following naming convention was use.

* BClass – Business Logic Layer classes
* MClass – Data structure or model classes
* IClass – Interface classes
* FClass – Factory classes
* DClass – Data access layer classes

The DALSQLServer uses two helper classes, SqlHelper, and SqlHelperExtention. The two classes were taken from the Microsoft Data Access Application Block and slightly modified to help decrease the number of connections to the data store.

What is important to note here is that the data store can be of any type so long as there is a .NET client, this includes Oracle.

To use any other data store all you should need to change is the web.config file and create the code necessary for the DAL. Meaning that should you need to use Oracle as your data store change the ApplcationDAL key value and create the project for it.

Please note an Oracle DAL has been written already for example purposes.

# Application flow

Application flow is fairly simple it beings in the WEB.CONFIG file with the httpModules section. A single http module (The ClientChoicesModule from the CLIENTCHOICES project) has been created to handle choices made by the client.

The next step is to the GLOBAL.ASAX.VB where thee methods are used the, “Application\_Start”, “Session\_Start”, “Application\_BeginRequest”, “Application\_AuthenticateRequest”, and “Application\_Error”

After the GLOBAL.ASAX.VB comes the root DEFAULT.ASPX, from there flow is handled by actions. Each ACTION is associated with a module or page and the NAVCONTROLER loads the appropriate module/page.

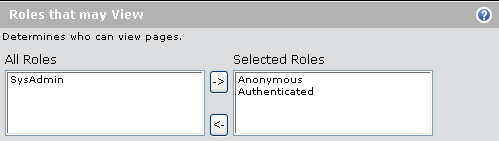
How To Technical

In this chapter you’ll find information pertaining to the usage and development tips for the Base Application

# Examples of Security

The BaseApplication comes with two example modules, the File Management, and Calendar. Let’s take a look at how security can be applied within the application to grant anonymous access to the file management module.

The file management (FM) modules purpose was to allow file/directory management through a web interface. The FM received a Security Entity property of 1 indicating that the module would be available to all Security Entitys. The security for Security Entity 1 was as shown in here:



Having this setup for the security allows the module to be displayed as a menu item for both anonymous access as well as when someone has been authenticated.

Now since the FM needs information about what directory it will display the All Security Entity was used as a foreign key in the FM’s data table. What does all this accomplish? When the global.asax.vb executes the code to determine view access the client information will be returned for the Anonymous account. The anonymous account been assigned view access through the Anonymous role (you may be interested to know that only 1 account is assigned to the anonymous role and that is, you guessed it, the anonymous account). That’s it; the system will now allow view access to the FM anonymously.

Let’s break this down into simple steps.

1. Select your desired Security Entity to apply the security for. (“All” will be for all Security Entitys)
2. Setup the module/page information assigning the anonymous role to the area’s you wish (View) for example.
3. Ensure the module can be seen when not logged on (Left hand menu).

Well that’s it in a nutshell.

# Adding a new Module/Page

Only the basic steps will be covered here for the most part most of what will need to be done is entirely up to the developer.

1. Create the module or page
2. If necessary create a model object
3. Create a DALInterface class
4. Create a DALFactory class
5. Create a DALOracle or DALSQLServer class
6. Create a BLL class to call the factory using the interface
7. In the module/page code behind call the BLL (or perhaps a utility class) to retrieve the data
8. Add the module/page information into the DB via the application

# Adding a new skin

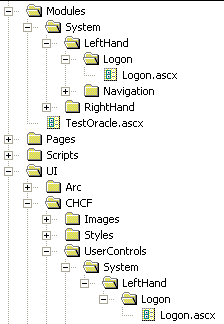
## To create a skin:

1. Create a new folder in the UI directory for your skin (the name of the directory will appear in the application as the theme name)
2. Add a DEFAULT.ASCX to the new directory
3. Place any controls in the layout you wish from the DEFAULT theme (most user controls from the DEFAULT theme have no images or unique attributes) or create your own.

Note: Use one of the existing themes as an example.

## To skin a single control:

1. Recreate the directory structure of the desired module in your new theme the directory in a directory named “UserControls” structure should look like this:

FIGURE 1

(In figure 1 the “Logon.ascx” module is being skinned)

1. Add a new user control with the same name as the target skinned module
2. Delete the VB file for the new control that was just created
3. Change the user control directive so it inherits the already existing modules code behind (In figure 1 the Logon.ascx file in the theme directory will execute the code behind of the Logon.ascx in the modules directory reducing the amount of code needed)

Summary

In this chapter you’ll find a summation for the Base Application.

I hope this is helpful to any that read it. It was never meant to be the end all be all for development and usage, just enough information to use what’s already been written with some insight as to why. In most cases parts of the BaseApplication will need to be altered to achieve the businesses needs but at least the application will serve as a starting point.

Further documentation should be forth coming for greater detail where it’s necessary.

Good luck to all.