DHSS .NET Development Manual
Revised 11/10/2014

Introduction and Table of Contents

This document contains the .NET applications development practices to be used by Delaware Health & Social Services' developers (both employee and contractual). The maintainer of this document is the IRM Base Technology group. Please direct any questions, comments, or suggestions for change to the maintainer at *dhsshelpdesk@state.de.us* or (302) 255-9150.

Consistent programming and development patterns are among the most important elements of predictability and discoverability in a managed class library. Widespread use and understanding of these standards should help eliminate unclear code and make it easier for developers to understand shared code. Though the methods described here depart from what has often been used at DHSS in the past (in particular, largely eschewing the use of so-called "Hungarian Notation"), it should be noted that these reflect the prevailing practices throughout the .NET community and that following a similar standard will make it easier when reading and evaluating code from sources outside DHSS.

It is assumed that the reader of this document has at least some familiarity with the general principles of .NET development including C#, HTML, and ASP.NET. Also, the reader is expected to have some familiarity with the principles of database use, including at least basic SQL skills.

This manual is divided into the following sections:

General .NET Programming Principles

Introduction

This section describes the general philosophy of .NET programming at DHSS and indicates the mind set for the entire DHSS .NET Development Manual. However, even though this was written specifically with .NET in mind, most of the ideas apply to programming in any environment. Therefore, you may have been instructed to follow the practices of this manual even though you are not working in .NET. If there is any question about what portions apply to your work, you should address them immediately to the IRM Applications Manager with whom you are working. Developers working in .NET must always assume that all portions apply to their work.

Exceptions to this Manual

In general, employees and contractors will not be allowed to vary their practices from the methods described in this manual. However, in the event that a practice is completely unfeasible in a specific application, there is a procedure for requesting a variance. A written request naming the requirement(s) to be waived must be submitted, in advance, to the DHSS Manager of Base Technology. The request must also include a detailed description of why the requirement(s) is/are unworkable. The exception will be evaluated by the DHSS Manager of Base Technology, the IRM Applications Manager responsible for the project, the IRM Director of Applications, and the Director of IRM. Only if all of those parties agree will the exception be granted.

In some portions of this manual, specific mention is made of the exception policy. This is not to imply that other portions are not required to follow the same policy, it is simply a reinforcement on what have proven to be common questions.

If an exception is granted, it will only apply under the specific circumstances for which is was granted. This means that simply because an exception was granted in the past

does not mean that it will be given again. Work must never be done based on the assumption that an exception will be granted.

Note that the circumstances for an exception must be profound and pose a significant risk to a project to be granted. Things such as simple aesthetic concerns, past habits, and so on are not considered valid reasons for exceptions.

Finally, be aware that willful and/or habitual disregard for the practices of this manual could result in disciplinary action.

The Act of Coding

Always create code that is clear and easy to follow. This means naming identifiers carefully so that others can later read code with a minimum of effort, it means formatting code carefully and consistently for the same reason, it also means that a programmer should always write as little code as is required to completely perform a given task. This does not mean that typing should be avoided (i.e., by using short identifier names and so on) but rather that unnecessary steps are avoided and that each line of code has a specific and necessary purpose.

Never perform optimization during initial coding. Never assume that a given piece of code will execute too slowly. Every situation is unique and it may be that your assumptions are faulty. Rather, write code that is as easy to follow as possible and then, if a problem is discovered, optimize later. In many cases (often, all cases), optimization may not be required at all in your application.

The most important factors in considering the creation of code are correctness, accuracy and reliability. Programs are correct when they satisfy their requirements specification and meet the project's objectives. A program is accurate when it produces the proper answer to the proper degree of resolution. Finally, a program is reliable when given a predictable set of operating conditions, it will not fail.

After these factors, the next most important goal of good programming is maintainability. A program that is not maintainable is like a car that has had the hood welded shut. Though it may run for the moment, if it breaks down, nothing can be done to fix it.

Maintainability is also the factor over which a programmer has the greatest control. Failure to follow proper guidelines and standards is the root cause of a lack of maintainability. Therefore, study this manual carefully and insure that all code you write conforms with its requirements.

Note: some developers have the habit of writing code however they wish with the intent of "fixing it at the end of the project when I have more time... ." This practice is unacceptable. There is usually little, if any, time left at the end of projects, therefore if the code is not written properly from the beginning, it will likely never be corrected. Additionally, it is usually the case that this practice will generate more work. Naming an

object properly from the beginning and then using that name is far easier than naming it incorrectly, using the incorrect name in dozens of places and then returning to correct all of those instances at a later date.

.NET Language Guidelines

The standard language of choice for DHSS .NET applications is C#. C# was selected by DHSS after careful consideration because of its strong feature-set and sound design.

In general, all DHSS .NET applications must be composed entirely of ASP.NET Web Forms. Windows Forms applications will only be allowed in very specific instances if the particular situation of an application warrants their use (such as an application that needs to have tight integration with the client workstation operating system). Using Web Forms means that applications will be more readily accessible to non -Windows platforms. Aesthetic concerns will not be considered sufficient cause to waive this policy. Whether or not you can use Windows Forms will be governed by the general exception policy.

Programmatic code attached to pages in DHSS .NET applications **must** be encapsulated in "code behind" files. This practice helps keep code as readable as possible and enhances maintainability.

Layered Application Design

The logical architecture of a DHSS .NET Applications requires, at a minimum, three distinct layers. They are the presentation/user interface layer, the business layer, and the data layer. The nature of these layers is described more fully in the section ".NET Applications Architectural Guidelines".

Distributed Processing Design

A distributed processing model is the required physical architecture for all DHSS .NET Applications. There are a minimum of four types of machines involved in this architecture: the client (i.e., the end user personal computer running a web browser), the web farm, the application farm, and the database cluster. This is a common industry model for applications deployment and is described more fully in the section ".NET Applications Architectural Guidelines".

The Relationship of Logical and Physical Architectures

A simplified view of the relationship between the two architectures is that the presentation layer resides on the web farm, and the business and data layers on the application farm. Communication between the presentation and business layers is required to use .NET XML Web Services and communication occurs between the data layer and the database cluster via stored procedures. This usage enforces loose coupling between the layers and helps to provide a more secure and robust

environment for execution.

Scoping

Proper scoping practices are vital to the creation of maintainable applications. In brief, an identifier is properly scoped if its level of accessibility does not violate the known interface of the code module and a identifier should always be scoped as small as possible. Examples of improper scoping are the use of side-effects and global variables. Identifiers should always be as narrowly scoped as possible to prevent unpredictable results in their use.

Variable Guidelines

To conserve resources, be selective in the choice of data type to ensure the size of a variable is not excessively large.

Keep the lifetime of variables as short as possible when the variables represent a finite resource for which there may be contention, such as a database connection.

Avoid the use of forced data conversion, sometimes referred to as variable coercion or casting, which may yield unanticipated results. This occurs when two or more variables of different data types are involved in the same expression. When it is necessary to perform a cast for other than a trivial reason, that reason should be provided in an accompanying comment.

Module Guidelines

Divide source code logically between physical files. Always break large, complex sections of code into smaller, comprehensible modules.

Modules and routines should be used for one and only one purpose. In addition, avoid creating multipurpose routines that perform a variety of unrelated functions. This is known as conditional complexity. Always look for opportunities to use the Extract Method to move conditionally complex code from an “If Block” or switch statement to a specialized routine.

Stateless components are preferred when scalability or performance are important. Design the components to accept all the needed values as input parameters instead of relying upon object properties when calling methods. Doing so eliminates the need to preserve object state between method calls.

Input Validation

Always use both server side and client side validation. Client side validation is important in making applications as responsive as possible and in increasing scalability by

reducing round trips to the web server. Since some users do disable client side scripting and because older browsers may not support client side scripting at all, server side validation must be provided as well to prevent the entry of incorrect data.

If a validation requires interaction with the database, write the business logic into a business logic layer component accessed via a web service and have the web service method return the success or failure of the validation.

Continuing...

The remainder of the DHSS .NET Applications Manual contains a great deal of information that expounds on the general ideas of this section and all developers working on DHSS .NET Applications should become intimately familiar with its contents.

.NET Naming Guidelines

General Naming Guidelines

The following rules apply to the creation of any identifier in code:

- Avoid abbreviations. When you must use abbreviations, use those that are well known by convention (such as ID for "Identification Number") or be entirely consistent in your use of a given abbreviation.
- Never begin identifiers with numerals.
- Underscore characters are only allowed in a few specific places. If the entity type you are naming does not explicitly state that you may use underscores, do not.
- Any acronyms of three or more letters should be Pascal case, not all caps.
- For true "throw away" loop counter variables i, j, and k are acceptable as they are well known to all programmers. If such a variable is seen, it will be assumed to be only a loop counter. If a variable will be used for any purpose other than only a loop counter, the minimum name length is six characters. It is recommended that identifiers be at least ten characters though for all but the simplest of entities.
- When naming elements, avoid commonly misspelled words.
- Do not use typographical marks to identify data types, such as $ for strings or % for integers.
- File and folder names, like procedure names, should accurately describe their purpose.
- Do not reuse names for different elements, such as a routine called ProcessSales() and a variable called ProcessSales.
- Avoid homonyms, such as write and right, when naming elements to prevent confusion during code reviews.

General Variable Guidelines

- Append computation qualifiers (Avg, Sum, Min, Max, Index) to the end of a variable name where appropriate.
- Use complementary pairs in variable names, such as min/max, begin/end, and open/close.
- Since most names are constructed by concatenating several words, use mixed - case formatting to simplify reading them. In addition, to help distinguish between variables and routines, use Pascal casing (CalculateInvoiceTotal) for routine names where the first letter of each word is capitalized. For variable names, use camel casing (documentFormatType) where the first letter of each word except the first is capitalized.

- Boolean variables which imply a true/false or yes/no value must begin with the word "Is", i.e., isFileFound or “has”, i.e., hasDocxExt .
- Avoid using terms such as Flag when naming status variables, which differ from Boolean variables in that they may have more than two possible values. Instead of documentFlag, use a more descriptive name such as documentFormatType.
- Even for a short -lived variable that may appear in only a few lines of code, still use a meaningful name.
- For constants, create a common constants class that will house all application constants. Constants should be declared using Pascal casing. For instance usuage would be for example myConstants.AppWinWidth.
- Do not use literal numbers or literal strings, such as for (i = 1; i <= 7; i++). Instead, use named constants, such as for (i = 1; i <= NUM_DAYS_IN_WEEK; i++) for ease of maintenance and understanding.

Classes and Structs

Pascal case.

[Code Example]

```
public class FileStream
public class Button
public class String
```

- Name classes with nouns or noun phrases that describe their purpose.
- Classes should not have the same name as the namespace in which they reside.
- Avoid using class names that duplicate commonly used .NET Framework name spaces and custom name spaces, for example system, Collections, Forms, UI, Imports, etc.
- Do not use a type prefix, such as C for class, on a class name. For example, use the class name FileStream rather than CFileStream.
- Occasionally, it is necessary to provide a class name that begins with the letter “I", even though the class is not an interface. This is appropriate as long as "I" is the first letter of an entire word that is a part of the class name. For example, the class name IdentityStore is appropriate.
- Where appropriate, use a compound word to name a derived class. The second part of the derived class's name should be the name of the base class. For example, ApplicationException is an appropriate name for a class derived from a class named Exception, because ApplicationException is a kind of Exception. Use reasonable judgment in applying this rule. For example, Button is an appropriate name for a class derived from Control. Although a button is a kind of control, making Control a part of the class name would lengthen the name unnecessarily.
- Business entities represent a single collection of data elements. For example, the BE structure for a database table called "Employees_T1" would be called "EmployeeData". Business Entity (BE) Classes (also called Data Transfer Objects) are a special case of class/structure that should always have the suffix

"Data" appended to their names. These names should also almost always be singular since they represent a single collection of data elements. For example, the BE structure for a database table called "Employees_T1" would be called "EmployeeData". Please note this method is for applications that do not utilize Object-Relational Mapping (ORM) technologies. ORM is an acceptable method to utilize.

Namespaces

Pascal case.

- Always begin namespaces for DHSS applications with the prefix "Dhss".
- The second component of the namespace name must always be the "application name". This name should be decided at the beginning of the project and used throughout. Note that this name may be English words or an acronym.
- The third component of the namespace name should be established according to the following:

Role	Component	Example
Browser Based UI/Pages	Web	Dhss.<division>.<AppName>.Web
Web Service Proxy Classes	Web.ServiceProxies	Dhss.<division>.<AppName>.Web.ServiceProxies
User Controls	Web.UserControls	Dhss.<division>.<AppName>.Web.UserControls
Business Logic Layer Components	BusinessLogic	Dhss.<division>.<AppName>.BusinessLogic
Web Services/.NET Remoting Facade	Facade	Dhss.<division>.<AppName>.Facade
Data Access Layer Components	DataAccess	Dhss.<division>.<AppName>.DataAccess
Common Application Components	Common	Dhss.<division>.<AppName>.Common
Business Entity	Common.DataModel	Dhss.<division>.<AppName>.Common.DataModel

- Separate logical components with periods, as in Microsoft.Office.PowerPoint.
- There are a number of namespaces that contain general components for use by all DHSS applications. These are contained within the DHSS.Framework namespace. Additions to this namespace can only be made with the approval of the DHSS Base Technology group.

- Use plural namespace names if it is semantically appropriate. For example, use System.Collections rather than System.Collection. Exceptions to this rule are brand names and abbreviations. For example, use System.IO rather than System.IOs.
- Do not use the same name for a namespace and a class. For example, do not provide both a Debug namespace and a Debug class.

Assemblies

Pascal case. If the assembly contains a single name space, or has an entire selfcontained root namespace, name the assembly the same name as the namespace. By convention, the ".dll" extension is always in lower case.

[Example]

```
Kronos.TimeCard.BusinessRules.dll
```

Collection Classes

Follow class naming conventions, but add "Collection" to the end of the name.

[Code Example]

```
public class WidgetCollection
```

Delegate Classes

Follow class naming conventions, but add "Delegate" to the end of the name.

[Code Example]

```
public class WidgetDelegate
```

Exception Classes

Follow class naming conventions, but add "Exception" to the end of the name.

[Code Example]

```
public class WidgetException
```

Attribute Classes

Follow class naming conventions, but add "Attribute" to the end of the name.

[Code Example]

```
public class WidgetAttribute
```

Interfaces

Follow class naming conventions, but start the name with "I" and capitalize the letter following the "I". Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the "I" prefix on the interface name.
[Code Example]

```
public interface IServiceProvider
public interface IFormatable
```

Enumerations

Follow class naming conventions.

- Do not use an Enum suffix on Enum type names.
- Use a singular name for most Enum types, but use a plural name for Enum types that are bit fields.

Functions and Methods

Pascal case, no underscores except in event handlers. Functions and subs must differ by more than case to be usable from case -insensitive languages like Visual Basic.NET. Use verbs or verb phrases. Methods/functions with a return value should have a name describing the value returned, such as GetObjectState().

[Code Example]

```
RemoveAll()
GetCharArray()
Invoke()
```

Procedure-Level Variables (i.e., local variables), Method Arguments and Function Parameters

Camel case.

[Code Example]

```
int CaculateAge(int yearOfBirth){}
int milesToGo
```

- Use descriptive names. Names should be descriptive enough that the name of the object and its type can be used to determine its meaning in most scenarios.
- Use names that describe an object's meaning rather than names that describe a object's type. Development tools should provide meaningful information about a object's type. Therefore, a object's name can be put to better use by describing meaning.
- Do not use reserved object names.

Class-Level Private and Protected Variables/Members

Camel case with a leading "_" (“m_” for legacy applications is acceptable).

[Code Example]

```
public class Person
{
	int _employeeNumber;
	string _firstName;
	int _age;

	public void IncreaseSalary()
	{
	}

	public void TerminateEmployee()
	{
	}
}
```

Properties and Public Member Variables

Pascal case. Members must differ by more than case to be usable from case – insensitive languages like Visual Basic.NET.

[Code Example]

```
Widget.Color, Widget.Height
```

Static Fields

Use Pascal case for all fields declared using the “static” modifier. Use nouns or noun phrases to name static fields.

Visual Controls on Forms

Modified Hungarian Notation using .NET class names. See the appendix at the end of this document for a list of the most common prefixes.

```
txtUserID, lblHeader, lstChoices, btnSubmit
```

Events

Pascal case. Always end with "EventHandler".

[Code Example]

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

.NET Coding Guidelines

General Guidelines

- Use case/switch statements in lieu of repetitive checking of a common variable using if...else if statements.
- Never hard code strings that might change based on deployment, such as connection strings.
- When building a long string, use StringBuilder, not String.
- With the exceptions of property accessors, constructors and overloaded operators, avoid providing methods in structures.
- Explicitly release dynamically created non-primitive object references.
- Divide source code logically between physical files. There must never be more than one class defined in a file.
- If a source file contains more than 500 lines or if an individual method contains more than 25 lines, evaluate them for possible modularization.
- Break large, complex sections of code into smaller, comprehensible modules.
- Always declare local variables as close as possible to their first use.
- Always use zero -based arrays.
- Always explicitly initialize an array of reference types using a for loop. Lync is acceptable for data objects.

  [Code Example]

```
public class MyClass{}

MyClass[] array = new MyClass[100];

for(int i = 0; i < array.Length; i++)
{
      array[i] = new MyClass();
}
```

- Do not put any logic inside AssemblyInfo.cs

- Do not put any assembly attributes in any file besides AssemblyInfo.cs
- Populate all fields in AssemblyInfo.cs such as company name, description, etc.. All assembly references should use a relative path.
- Always run code unchecked by default (for performance), but explicitly in checked mode for overflow or underflow prone operations.

  [Code Example]

```
public int CalcPower(int number, int power)
{
	int result = 1;




	for(int i = 1; i <= power; i++)
	{
		checked
		{
			result *= number;
		}
	}

	return result;
}
```

- Avoid explicit code exclusion of method calls (#if...#endif). Use conditional methods instead.

  [Code Example]

```
	public class MyClass
	{
		[Conditional("MySpecialCondition")]
		public void MyMethod()
		{
		}
	}
```

Classes and Structs

- Provide a default private constructor if there are only static methods and properties in a class. This prevents classes from being created when they should not be.
- Minimize the amount of work done in the constructor. Constructors should not do more than capture the constructor parameter or parameters. This delays the cost of performing further operations until the user uses a specific feature of the instance.

- It is recommended that you not provide an empty constructor for a value type struct. If you do not supply a constructor, the runtime initializes all the fields of the struct to zero. This makes array and static field creation faster.
- Use parameters in constructors as shortcuts for setting properties. There should be no difference in semantics between using an empty constructor followed by property set accessors, and using a constructor with multiple arguments.
- Avoid friend assemblies, as it increases inter -assembly coupling.
- Do not use code that relies on an assembly running from a particular location.

Methods

- Do not create methods with more than seven arguments. If a method requires more data to be passed to it, use structures/classes to encapsulate the arguments.

Namespaces

- A nested namespace should have a dependency on types in the containing namespace. For example, the classes in the System.Web.UI.Design depend on the classes in System.Web.UI. However, the classes in System.Web.UI do not depend on the classes in System.UI.Design.
- Avoid putting a using statement inside a namespace.
- Group all framework namespaces together and put custom or third party namespaces beneath separated by a space.

  [Code Example]

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using MyCompmnay;
using MyControls;
```

Exceptions

- All code paths that result in an exception should provide a method to check for success without throwing an exception. For example, to avoid a FileNotFoundException you can call File.Exists. This might not always be possible, but the goal is that under normal execution no exceptions should be thrown.
- Use a localized description string in every exception. When the user sees an error message, it will be derived from the description string of the exception that was thrown, and never from the exception class.
- Create grammatically correct error messages with punctuation. Each sentence in the description string of an exception should end in a period. Code that

generically displays an exception message to the user does not have to handle the case where a developer forgot the final period.

- Provide exception properties for programmatic access. Include extra information (other than the description string) in an exception only when there is a programmatic scenario where that additional information is useful. You should rarely need to include additional information in an exception.
- Do not use exceptions for normal or expected errors, or for normal flow of control.
- You should return null for extremely common error cases. For example, a File.Open command returns a null reference if the file is not found, but throws an exception if the file is locked.
- Only catch exceptions for which you have explicit handling.
- Design classes so that in the normal course of use an exception will never be thrown. In the following code example, a FileStream class exposes another way of determining if the end of the file has been reached to avoid the exception that will be thrown if the developer reads past the end of the file
  [Code Example]

```
public class FileRead
{
	public void Open()
	{
		FileStream stream = File.Open("myfile.txt", FileMode.Open);
		byte b;

		// ReadByte returns -1 at end of file.
		while ((b = stream.ReadByte()) != true)
		{
			// Do something.
		}
	}
}
```

Enumerations

- Always add the Flags attribute to a bit field Enum type. Such collections are manipulated using bitwise operators and you must specify the enum values in powers of two.
- Avoid providing explicit values for enums that are not bit filed enum types.

Class-Level Private and Protected Variables/Members

- All member variables should be declared at the top, with one line separating them from the properties or methods.
- Always explicitly specify the protection level of members (i.e., private, public, protected, etc.).

Static Fields

- Use static properties instead of public static fields whenever possible.

Events

- Specify two parameters named "sender" and "e". The sender parameter represents the object that raised the event and is always of type "Object", even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named "e". Use an appropriate and specific event class for the e parameter type, this class always ends its name with the "EventArgs" suffix.

Debugging

- Asserting is a powerful tool that .NET gives the developer. Developers should always assert every operational assumption, especially ones where the failure of the action would have potentially destructive results in a production scenario.

Database Guidelines

Policies Specific to DB2 Database Development

The DB2 for MVS platform at DHSS does have a small number of other restrictions. Most notably, database object names are limited to 18 characters in length, which means that when creating names for objects, you must account for that smaller allowed amount. In addition, DB2 tables are prefixed with a structure of the form "T999_" where the "999" varies according to the application to which the tables belong. The DHSS database administrators will be able to provide developers with the appropriate prefix for their application(s).

Special Note on the Use of *varchar* Fields

Though not actually an object naming consideration, there are some special concerns around the use of *varchar* type fields in a DB2 database. Though there is functionally very little difference between *char* and *varchar* type fields in SQL Server, there is an important distinction between the two in DB2. Developers used to working in SQL Server should be aware of this distinction and design their databases appropriately in DB2.

Fields typed as *varchar* in DB2 are used on a far smaller scale than in SQL Server. This is mainly due to differences in the way that DB2 allocates space for data. What this means in effect is that there are a few general rules that should be used when considering the creation of a *varchar* field in DB2.

1. If a field's total length is less than 40 characters, use *char* .
2. If a field can be greater than 40 characters but generally the data inserted into the field will be fairly close to the maximum allowed, use *char*.
3. If *varchar* fields are to be used, then position them at the end of a table's definition.

Stored Procedure Naming Considerations in MS SQL Server Databases

The general rule for naming stored procedures in an MS SQL Server database is the table name that the stored procedure operates on without the object suffix, and underscore, the name of the operation performed and the finally an "_S1" suffix. For example, a procedure which searches the Users_T1 table would be named Users_Search_S1. Naming stored procedures in this manner is required as it allows for the group of procedures that operate on a table to be readily identified. There are three stored procedure operations that are extremely common in DHSS applications and, in fact, generally constitute the bulk of stored procedures defined in a database. The operations concerned are ones that retrieve a single record based on a primary key, insert a new record and update an existing record. Since these are so common, there are specific rules governing their naming. Use the following table to determine the appropriate name for your stored procedure of these types:

Operation	**Table Name**	**Required SP Name**
Retrieve	Users_T1	Users_Retrieve_S1
Insert	Users_T1	Users_Insert_S1
Update	Users_T1	Users_Update_S1

The table names provided above are simply examples, and should be replaced as appropriate with the table with which your stored procedures work.

In addition to the names of the stored procedures themselves, there are also standards for the names and types of stored procedure parameters. Parameters must be named the same thing as any fields that they are matched against AND match that field's datatype. The parameter will be distinguished from the field by the parameter prefix @ which is a requirement in MS SQL Server stored procedures. Therefore, if a stored procedure requires a parameter that will be matched against (or inserted into, or used to update) a field name First_NAME which is a varchar(50), the stored procedure parameter will be called @First_NAME and also have a type of varchar(50).

If a parameter is not directly matched against a field in a stored procedure, it may be named as the programmer wishes, as long as the general rules for naming fields are followed (i.e., an identifier followed by an underscore and a descriptor suffix).

SQL Coding Guidelines

This section includes information on both proper design practices for SQL Programming and how to format your SQL code so that it conforms to established DHSS guidelines.

SQL Programming Guidelines

- Use DHSS Database object naming standards in databases. Refer to the section of this manual titled "Database Object Naming Guidelines" for more information.
- Never use SELECT *. Always be explicit in which columns to retrieve and retrieve only the columns that are required.
- In INSERT statements, always list the fields to be inserted explicitly, do not rely on the natural order of fields to do an insert. This will insure that your statements will not have unpredictable behavior should the structure of the table ever change.
- Refer to fields explicitly; do not reference fields by their ordinal placement in a recordset. (Do not refer by the recordset column numbers).
- Verify the row count when performing DELETE and UPDATE operations.
- If possible, specify the primary key in the WHERE clause when updating a single row.
- Use forward-only/read -only recordsets. To update data, use SQL INSERT and UPDATE statements.
- Never hold locks pending user input. Use optimistic concurrency to control "clobbering" of update operations by users.
- Use uncorrelated subqueries instead of correlated subqueries. Uncorrelated subqueries are those where the inner SELECT statement does not rely on the outer SELECT statement for information. In uncorrelated subqueries, the inner query is run once instead of being run for each row returned by the outer query.
- Never placed embedded SQL statements directly in application code. Write SQL statements in stored procedures and call them from a web service method.
- Always fully qualify field names in SQL with either the actual table name or a table alias that you define in your SQL code (with the table alias being the preferred method). This will insure that there are no ambiguities in your SQL code.
- Do not open data connections using a specific user's credentials. Connections that have been opened using such credentials cannot be pooled and reused, thus losing the benefits of connection pooling. The user information will be stored in web.config for on the component server.
- Any interaction with your database(s) must be from a web service. In the production environment, the web services will be running in a component server that will not be in a DMZ. This way the interaction of the web service to the database is hidden from the public.
- DDL scripts must always be saved into files and structural changes to the database always made through these scripts. Enterprise Manager GUI functions cannot be used to make production database changes, therefore any such changes must be accompanied by a DDL script which creates/changes the object in question and appropriately sets permissions for the object.

- Parameters to SQL statements must always be supplied via bind variables and not through concatenating a string together that contains the values to be input. This insures that SQL code is more readable and prevents the use of so-called "SQL Injection" attacks.
- Stored procedures must never contain business logic. Such processing must take place at your business layer. Stored procedures are to be limited to a single operation of retrieving, inserting, or updating data.
- CodeSmith templates exist that have been set up to generate and properly format common stored procedures based on the database structure. These templates are available upon request from the DHSS Manager of Base Technology.

SQL Code Formatting Guidelines

When writing SQL statements, use all uppercase for keywords and Pascal case for database elements, such as tables, columns, and views.

Put each major SQL keyword on a separate line, then its objects (i.e., the fields or other entities on which it operates) on the following line, indented by one tab space. If there are multiple objects to a keyword, each should be on a separate line. Major keywords which are dependent on a preceding keyword should be further indented by one tab space and their objects indented yet another tab space.

Following are examples of several different types of common SQL statements:

[Code Example - SELECT Statement]

```
SELECT
	C1.FirstName_TEXT,
	C1.LastName_TEXT
FROM
	Customers_T1 C1
JOIN
	Address_T1 A1
		ON
			C1.CustomerID_NUMB = A1.CustomerID_NUMB
WHERE
	A1.State = 'WA'
AND
	AA1.County = 'NC'
ORDER BY
	C1.LastName_TEXT,
	C1.FirstName_TEXT
```

[Code Example - Insert Statement]

```
INSERT INTO
	Customers_T1 C1
	(
		C1.LastName_TEXT,
		C1.FirstName_TEXT
```

```
	)
	VALUES
	(
		:lastName,
		:firstName
	)
```

[Code Example - Update Statement]

```
UPDATE
	Customers_T1 C1
SET
	C1.LastName_TEXT = :lastName,
	C1.FirstName_TEXT = :firstName
WHERE
	C1.CustomerID_NUMB = :customerID
```

[Code Example - Delete Statement]

```
DELETE FROM
	Customers_T1 C1
WHERE
	C1.CustomerID_NUMB = :customerID
```

[Code Example - Stored Procedure Creation Statement]

```
CREATE PROCEDURE
	Customers_Insert_S1
	(
		@FirstName_TEXT VARCHAR(30),
		@LastName_TEXT VARCHAR(30),
		@Customer_IDNO INT OUTPUT
	)
	AS
	BEGIN
		INSERT INTO
			Customers_T1 C1
			(
				C1.LastName_TEXT,
				C1.FirstName_TEXT
			)
		VALUES
			(
				@LastName_TEXT,
				@FirstName_TEXT
			)
		SET
			@Customer_IDNO = @@IDENTITY
	END
```

.NET Applications Architectural Guidelines

1. Introduction

1.1. Goals / Purpose of Document

The goal of this document is to set forth architecture guidelines for application architects and designers at DHSS. DHSS has established these guidelines in order to facilitate quality software development through standardization of recommended architectural patterns.

Note that this section is still under construction but it already contains a great deal of information important to the development of applications at DHSS.

2. Architecture Overview

2.1. What is the DHSS .NET Architecture?

In order to meet the demand of our customers for more complex, more available, and more usable systems, DHSS has identified the following four requirements as top priorities to be met by the application architecture.

1. **Maintainability:** The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment [IEEE 90]. Applications must be flexible enough to adapt to

changing or expanding business requirements without losing integrity and reliability. Applications must also be constructed in such a way that their implementations are comprehendible and navigable by new developers.

2. **Availability:** The degree to which a system suffers degradation or interruption in its service to the customer as a consequence of failures of one or more of its parts [HYPR 04]. The definition of availability will change from one client to the next. It is our responsibility to create applications that will meet the client's initial requirements for availability and will not fail if that requirement is changed.
3. **Scalability:** How well a solution to some problem will work when the size of the problem increases [HYPR 04]. The future user base of an application cannot be predicted with certainty at any point in the development process. Therefore, the application must be built on a sound architecture that will scale to meet an increased demand.
4. **Security :** (define security)

2.2. Logical Architecture Overview

"Logical architecture" refers to the organizational choices made by the designer when determining how to represent the business requirements of an application in code and data. The following figure depicts the DHSS logical architecture.

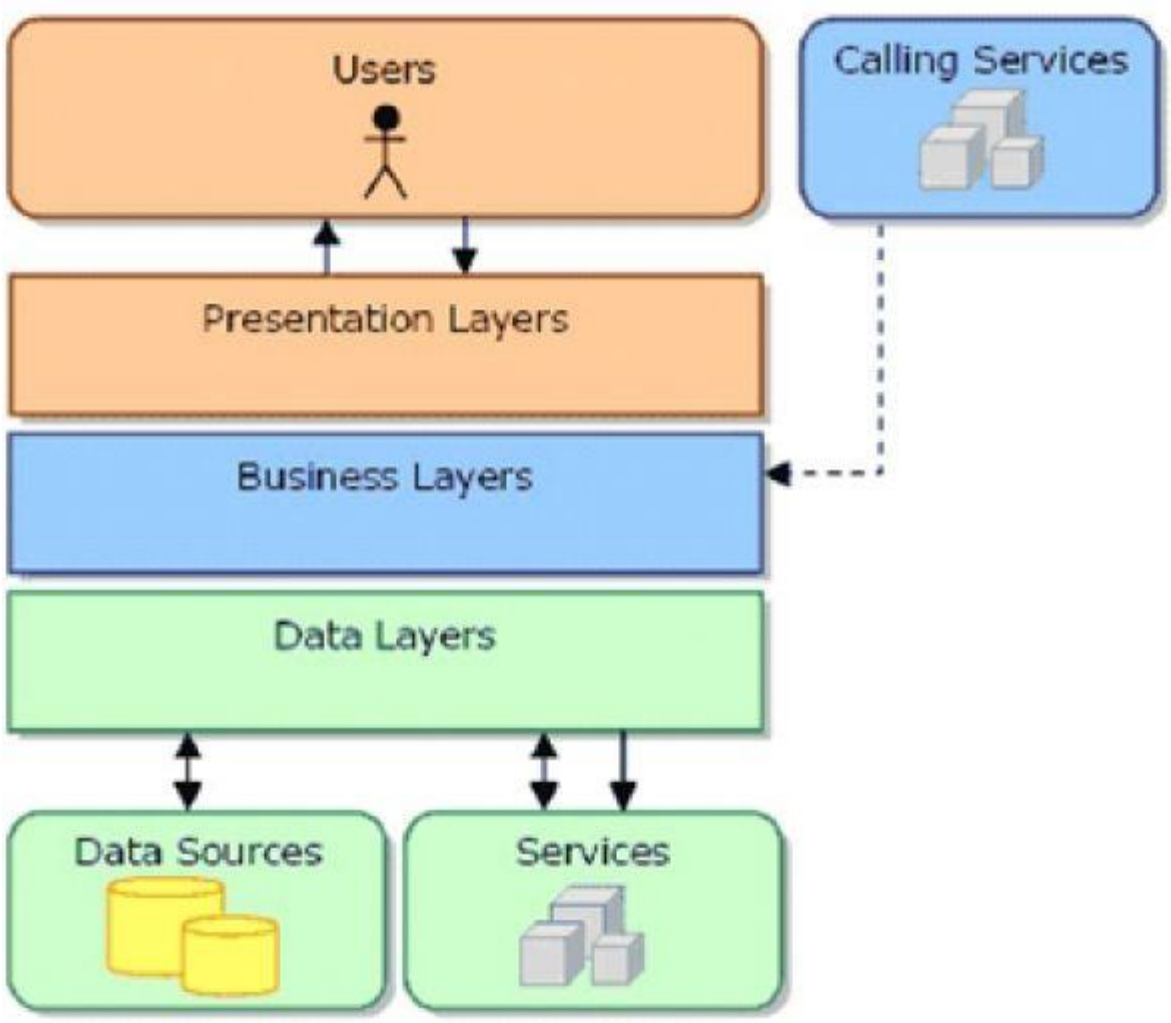


Figure 2.1 DHSS Logical Architecture

The logical architecture shows that we have broken the application into three main layers. These layers are Presentation, Business, and Data. These layers rest on top of each other and operate in a top-down manner, such that any given layer only has knowledge of the layer directly beneath it. This layering approach allows us greater division of labor -- among both employees and system components. Better maintainability is achieved due to decreased component interdependency. While the components will be more easily modifiable, they will also be more understandable due to the principals of layering. Any layer can be understood as a coherent whole without knowing the implementation details of any other layer. The logical architecture is discussed in detail in section 3.

Take note, this does not imply anything about the physical deployment and distribution of these layers across client and server machines. The layers must be designed so that they can be distributed, but they do not require any specific number of physical machines, or **tiers**, in order to be functional or applicable. The same reduction in maintenance cost will be seen, even if the entire layered application is deployed to a single stand-alone machine.

2.3. Physical Architecture Overview

"Physical architecture" refers to the deployment arrangement of an application's components. The following figure illustrates the standard DHSS physical architecture.

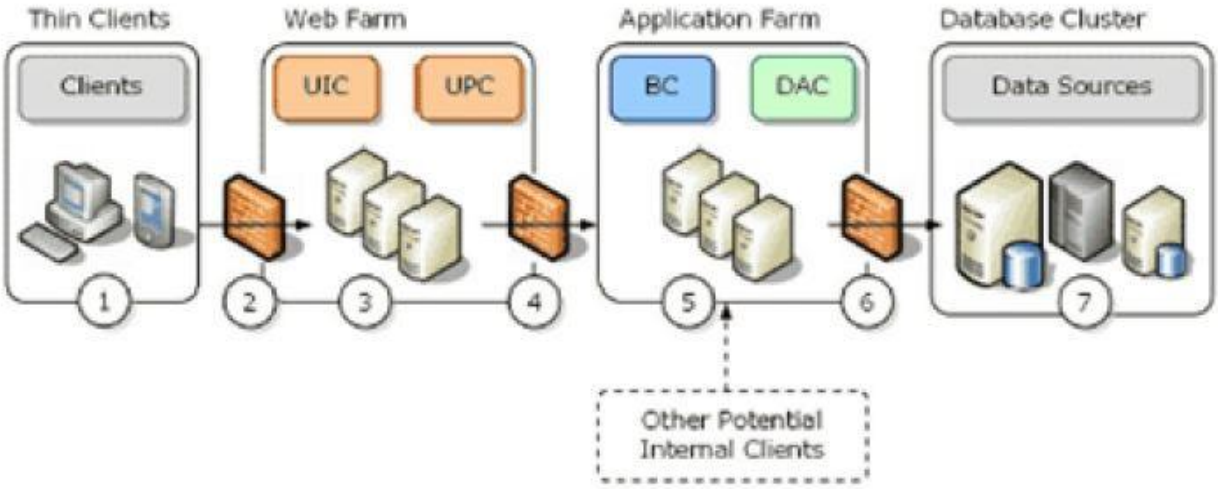


Figure 2.2 DHSS Physical Architecture

The three remaining requirements, availability, scalability, and security, are fulfilled by the physical architecture. Application availability is gained through use of a redundant clustered design. Our application is deployed across the web farm, the application farm, and finally, the database cluster. The failure of any server will not disrupt the application's ability to serve client requests. Scalability is also accomplished through this clustered architecture, as any given cluster can be scaled out by dropping in additional servers. An application deployed in this way will also receive an automatic boost in security since the only exposed components, the user interface, will have no direct access to the database servers. The physical architecture is discussed in detail in section 4.

3. Logical Architecture

As described above, the logical architecture is composed of 3 layers - user interface, business, and data access. Each of these layers has a specific set of responsibilities. In each layer, there are generic object types that only exist in that layer. These object types have been identified through examination of reoccurring patterns in well-designed modern systems. The following is a detailed description of each layer and the object types that must be implemented in order to provide its services.

3.1. The Presentation Layer

3.1.1. User Interface Components

The user interface components (UIC) of an application are what enables user interaction with the system. It is the responsibility of the UIC to accept data from the user and later output that data as human-readable information. In the DHSS Application Architecture, these components are primarily found in the form of .aspx web pages. Web interfaces have been chosen as the standard due to their inherent potential for scalability, the high availability nature of IIS, and ease of deployment.

It is important to understand that no business logic is contained within these user interface components. The business logic that might have been present in a non - layered application has been relocated to the Business Logic Layer. This helps to improve application resiliency and enables business logic to be reuse from a different user interface, such as a lightweight PDA version or another related application.

UICs have the following responsibilities:

- Acquiring data from users and performing early data format validation. This may include range, type, and
- Reacting to a user request by accessing the Business Logic Layer and properly formatting the retrieved data for the user's consumption.
- Accessing User Process Components to facilitate navigation of complex user interaction cases such as wizards.
- Rendering the data, usually for a given business entity, to the screen in a format that is understandable to the user

3.1.2. User Process Components

In order to deal with complex workflows, DHSS will utilize User Process Components (UPC) to help isolate and encapsulate the details of interface navigation. There may be situations, especially in wizards, where the next step in a user's interaction with the system is determined by some variable supplied by the user. These situations are ideal candidates for creation of a UPC. Other times, a long running business process may need to provide the ability to be persisted at random. The logic that allows for persistence may be complex and is best isolated in a UPC.

UPCs have the following responsibilities:

- Providing a simple, reusable interface to a complex business workflow.
- Maintaining current state of the workflow-associated data.
- Persisting data as required.

3.2 The Business Logic Layer

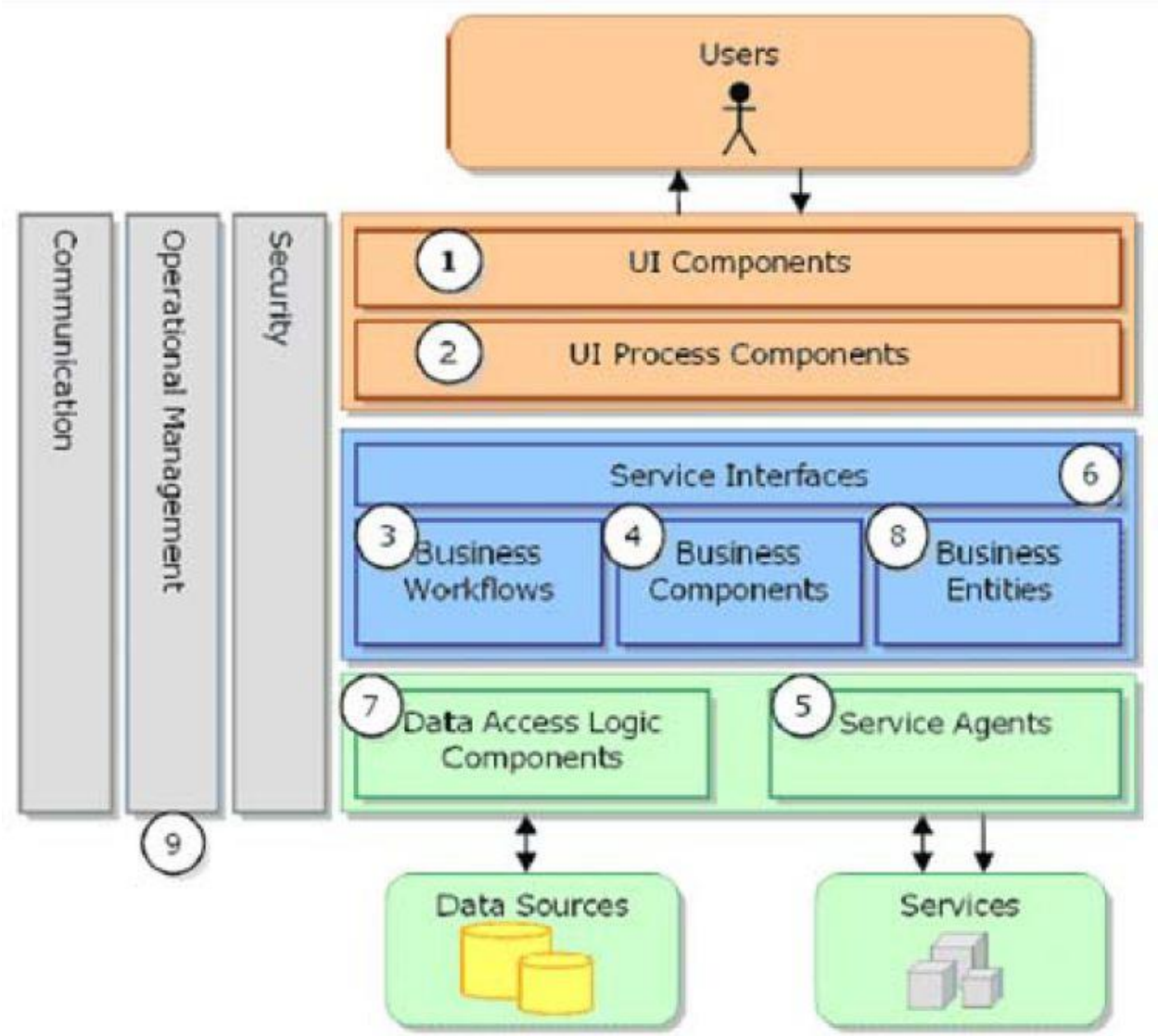


Figure 3.1 DHSS Logical Architecture – Detail

3.2.1 Business Logic Components and Remote Facades

The Business Logic Layer (BLL) is home to the majority of the application -specific code. All of the scenarios that are found in the problem domain must be modeled in the BLL. This includes calculations of input or stored data, validation of operations attempted by the presentation layer, transaction enforcement, and determining which data access operations to execute. In order to provide a loosely coupled design, the BLL must not have any knowledge of the internals of the presentation layer. Other presentation layers or service layers may be constructed on top of the BLL without impacting the code found within the BLL.

Ideally, we would like to implement the BLL as a pure object -oriented model of business object components. A traditional object -oriented approach would involve objects that provide accessor and modifier functions for each property with the object

maintaining its state between these and other functional calls. For instance, you may set several fields of an object and then call a save method in order to persist the object's state to a file or database. This is known as a fine-grained interface. An example follows:

```
Customer newCustomer = new Customer();
newCustomer.FirstName = "John";
newCustomer.LastName = "Doe";
newCustomer.City = "New Castle";
newCustomer.State = "DE";
newCustomer.Save();
```

Keeping in mind that the BLL objects are remote from the web server in the DHSS Physical Architecture (Figure 1.2), it becomes apparent that a fine-grained model won't meet our architectural goals for two reasons. First, the amount of network traffic caused by repeated calls to accessors and modifiers will cripple the performance and scalability of our application. Second, remote web services are by nature stateless, making fine - grained interfaces impossible. In order to solve this problem, we will follow the Remote Facade [fowler] pattern by creating coarse-grained interfaces that will act as batch processing service layers over top of our domain objects. Most often, this will involve the creation of a business entity communication package that is delivered one time across the network to the application server. An example follows:

```
CustomerData newCustomerData = new CustomerData();

// Local business entity, not remote!
newCustomerData.FirstName = "John";
newCustomerData.LastName = "Doe";
newCustomerData.City = "New Castle";
newCustomerData.State = "DE";

CustomerWS.CustomerFacade = new CustomerWS.CustomerFacade()

// The following call crosses the network
CustomerFacade.Save(newCustomerData);
```

Business Logic Components have the following responsibilities:

- Implementing the core functionality specific to an application.
- Beginning and committing or rolling back transactions.
- Respond to consumers (business facades, other business components) by accessing the Data Access Layer in order to persist or retrieve data

Remote Facades have the following responsibilities:

- Provide a coarse-grained interface to remote consumers (presentation layer, other applications) for the business logic components.
- Instantiate and destroy any necessary business logic components for a given operation.

3.2.2 Business Entities

As mentioned above, Business Entities (BE) are used as a means of packaging data for delivery across layers. They will be used by presentation and business layers. DHSS has elected to use very thin classes to represent BEs. More like a "smart structure", these classes will only contain properties, data format validation code, a method to validate, and a method to determine if the data is "dirty" (changed since last retrieval).

BEs do not contain any business logic beyond data format rules. These data format rules include required, range, and co-occurence constraints. BEs also do not contain any data access logic. This is handled by the Data Access Layer.

Business Entities have the following responsibilities:

- Enforce their own data format through use of exceptions.
- Provide their consumer with information regarding broken format rules

3.3 The Data Access Layer

The Data Access Layer (DAL) is the foundation of most information systems. The DAL provides the ability to persist and later retrieve the application's data through the use of a data store. At DHSS, this data store is a SQL Server database or databases that is accessed through a stored procedure interface.

3.3.1 Data Access Logic Components

The stored procedure interface is used by Data Access Logic Components (DALC) that provide a bridge between our application's business logic and the data store. These DALC most often represent a 1-to-1 relationship with the business entities of the system and are always static classes. Each business entity will require a regular set of operations -- create, retrieve, update, and delete (CRUD). The CRUD operations will be used by the BLL in order to satisfy requests from the presentation layer. Data from the DALC is packaged into BEs and returned to the BLL. This helps to encapsulate the database and encourages a resilient BLL which is unaware of the actual implementation of the BEs it is using. Changes affecting the order or names of fields in a database table will not require cascading changes within the other layers of the application.

In order to enforce standard patterns of interaction with the database, DHSS DALC components will always use the Microsoft Data Access Application Block (SqlHelper.cs) as an interface to ADO.NET.

Data Access Logic Components have the following responsibilities:

- Provide a stateless interface to CRUD operations for each BE
- Provide more specific operations not related to a single BE, such as multi-table searches
- Provide some level of relational-to-object mapping by returning BE objects

4. Physical Architecture

Once a logical architecture is established, it must be decided how to distribute that architecture across physical tiers of an operating environment. DHSS has invested a considerable amount of resources in defining and creating a scalable, secure physical architecture for all applications.

4.1 The Standard DHSS Deployment Pattern

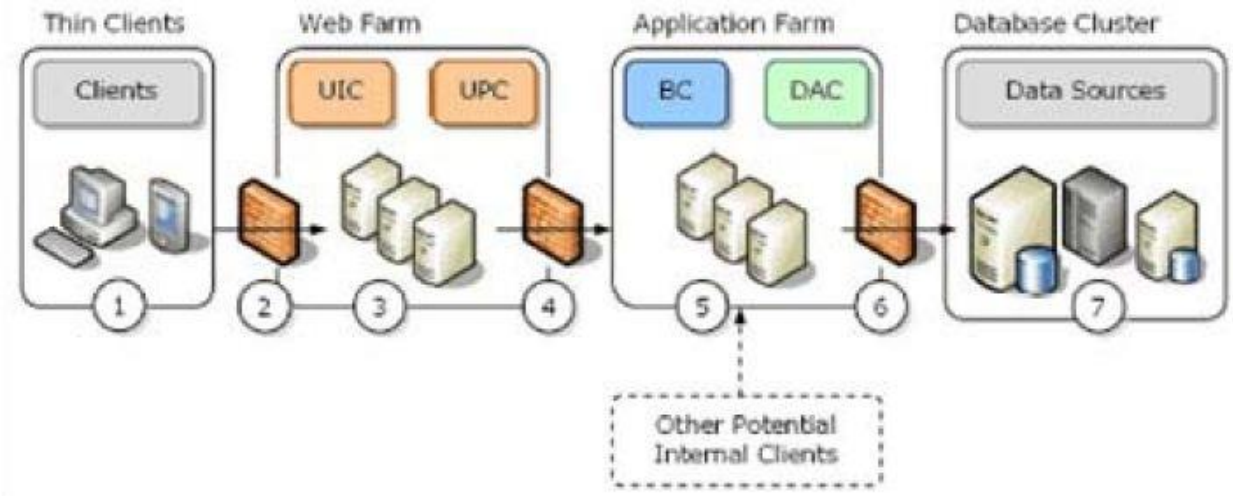


Figure 4.1 DHSS Physical Architecture

Figure 4.1, repeated from figure 2.2, clearly represents the physical deployment guidelines of the DHSS logical architecture. An explanation of each node follows:

1. **Clients:** Applications will be developed as ASP.NET web apps and as such the client will be required to have a system running an up to date browser supporting HTML 4.01.
2. **DMZ Firewall:** The DMZ within which the web server resides is protected by this hardware firewall.
3. **Web Server Cluster:** The web cluster will consist of (X) clustered servers and contain all ASP.NET user interface components and user process components. The cluster will run Windows Server and IIS with the .NET Framework.
4. **Internal Firewall:** An internal firewall protects the application cluster from the web cluster should it be compromised.

5. Standard DHSS Deployment Pattern(s)
6. Hardware
7. Software - As of November, 2014 Windows Server 2008 R2 – 2012 and SQL Server 2008 R2 – 2012 are supported. Since these are subject to upgrade, please see DHSS Basetech Manager for the latest software version.
8. Communication Policies Between Tiers

System Foundations

1. Security: Explain authentication & authorization model
2. Error Handling: Exceptions - how & where to catch, where to log

Page/Form Design Guidelines

The following must be considered when designing web forms:

Page Layout and General Principles

- When designing applications, use the DHSS browser based application/interactive web site templates described later in this document. Use of these standard layout schemes will allow DHSS to produce consistently attractive and functional applications while allowing for developers to focus on business functionality rather than design considerations. Furthermore, the applications templates have been developed using cascading style sheets so that they will gracefully degrade in older web browsers and leave the applications accessible to user clients that may be out of date.
- The applications templates have an already established color scheme and usage of this color scheme is required.
- Do not create graphical elements for use in your application beyond those specified by the DHSS Forms Template without the explicit permission of the DHSS Manager of Base Technology. Generally, such graphical elements will not be allowed unless they are required for good reason.
- Grid layout is not allowed in page design. Use only Flow Layout. The applications template specifies the use of style sheet positioning to layout elements and this is required. The use of tables for layout purposes only is not allowed in general and must be approved by the DHSS Manager of Base Technology on a case by case basis.
- Page titles should be composed thusly: System Name: Sub System Name: Page Name.

Images

- All images must have the alt, width and height attributes defined.
- Use decorative images very sparingly.

Formatting

- <blink> and <font> tags are never allowed.
- Use the template supplied style sheet elements for formatting, creation of new elements must be approved by the DHSS Manager of Base Technology.
- Provide a location to display validation messages so that they are presented neatly and consistently positioned.
- Avoid in-line styles.

List Type Controls

- Each list box must show at least four values (or all values if there are less than four).
- Horizontal scroll must be avoided in list boxes.
- List boxes and combo boxes must be wide enough to allow all text to be viewed.
- Every list box or combo box must have a default selection (Blank/null is acceptable for this purpose if that is indeed a valid entry for the field).

Navigation

- Navigation items must always use links, buttons should only be used for application actions. In general, navigation items should be restricted to the area presented in the sidebar in the DHSS Forms Template.

Using the DHSS Template

The DHSS Template is a reference and training .NET solution to guide you through the various aspects of creating a DHSS Web Application. Please see the document is appended to the end of this document for more information.

ASP.NET/HTML/XML Coding Guidelines

Label Tags

All controls should have their respective labels associated with them via use of the <label> tag. In many cases, Visual Studio will not automatically insert these tags and they must be manually entered into the HTML code.

[Code Example]

```
<label for="txtName">User Name</label><input id="txtName">
```

Working With Dates

The use of calendar controls is not allowed due to the accessibility issues it causes. The data entry of date values must use the format MM/DD/YYYY.

Controls

- User controls should be used when possible to reduce repetitive coding.

- Always assign the id attributes of all controls as this allows non -visual web browsers to function more effectively.
- Use Hungarian notation for controls.

Third Party Controls

Third party controls are to be avoided unless absolutely necessary. If a third-party control is required, written permission must be obtained from the IRM Project Manager in charge of the project and the DHSS Manager of Base Technology. This policy applies even in the case of "free" third-party controls.

Note: in some cases, Visual Studio fails to properly follow some of these standards. This is not sufficient reason for a waiver of these standards. In cases where Visual Studio does not take the proper action, the code must be corrected manually.

ASP.NET/HTML/XML Code Formatting

All the tags and attributes in the page must be in all lowercase (to maintain future compatibility).

Each attribute value must be enclosed in quotes (to maintain future compatibility).

.NET Commenting Guidelines

Properly commented code is crucial to a program's future maintainability. What many programmers learned early on though in commenting practices was entirely wrong. It was common practice in many universities and schools to teach that comments should summarize every line of code, in effect, producing pseudo-code inline with the actual code. This does little though to actually document code in most cases. Properly written modern code is actually fairly self-documenting in the terms of what pseudo-code provides.

Rather than pseudo -code, code comments should provide plain English descriptions of the activities taking place in the code and provide metadata about the code in question (i.e., who is the maintainer of the code, last revisions, and so on). Comments should always be proper and complete sentences. In addition, every major block of code (method, function, class, etc) should be headed by an extensive header comment that explains the purpose of the block, its assumptions and the type of output that one should expect from it.

- When modifying code, always keep the commenting around it up to date.
- At the beginning of every routine, it is helpful to provide standard, boilerplate comments, indicating the routine's purpose, assumptions, and limitations. A boilerplate comment should be a brief introduction that explains why it exists and

what it can do. It is required that you use the XML commenting feature of .NET to accomplish this. Using XML style comments will allow the code to be preprocessed to produce and English language manual of the code. It will also enable the IDE in .NET to provide on -the-fly information as you reference the commented items. See further down this document for specific information on the use of XML comments.

- Avoid adding comments at the end of a line of code; end-line comments make code more difficult to read. However, end -line comments are appropriate when annotating variable declarations, in which case, align all end -line comments at a common tab stop.
- Block comments withing code should be surrounded in a typographical frame of the following configuration:

```
//***                                                    *
//
//
//
//
//***                                                    *
```

XML comments are also acceptable within code that do not fall within block commenting.
With the actual comments placed on the lines between the bounds. Such a structure provides visual distinction without becoming a maintenance problem (in terms of constantly needing to reformat code).

- Make liberal use to the //TODO: comment construct. Any comments begun with that text will appear in the task list in Visual Studio. This means that areas that need work can be readily identified and not lost in the mass of code.

[Code Example]

```
//TODO: John Doe should verify that the below value is correct.
```

Will show up in the task list reminding John Doe of his action item.

- Prior to deployment, remove all temporary or extraneous comments to avoid confusion during future maintenance work. In order to facilitate this, any temporary comments (such as ones used to temporarily disable sections of code) should be marked with the phrase "REMOVE BEFORE DEPLOYMENT", which can then be searched for using your code editor. Such items should be marked using the //TODO: comment structure as described above.
- Use complete sentences when writing comments. Comments should clarify the code, not add ambiguity.
- Comment as you code because you will not likely have time to do it later. Also, should you get a chance to revisit code you have written, that which is obvious today probably will not be obvious six weeks from now.
- Avoid superfluous or inappropriate comments, inappropriate and offensive language should never be used.

- Use comments to explain the intent of the code. They should not serve as inline translations of the code. Avoid comments that explain the obvious. Document only operational assumptions, algorithm insights and so on.
- Comment anything that is not readily obvious in the code.
- To prevent recurring problems, always use comments on bug fixes and work - around code, especially in a team environment.
- Use comments on code that consists of loops and logic branches. These are key areas that will assist source code readers.
- Throughout the application, construct comments using a uniform style with consistent punctuation and structure.
- Separate comments from comment delimiters with white space. Doing so will make comments obvious and easy to locate when viewed without color clues.
- Code that is not properly commented as spelled out in this document will not be accepted as a deliverable, irrespective of whether or not the author is an internal DHSS employee or an outside contractor.

XML Comments

Note: portions of this section were adapted from the online documentation for XML comments provided by Microsoft.

XML comments are a powerful feature of .NET that are especially useful when used in conjunction with the Visual Studio.NET development environment. The .NET framework alone supports the ability to extract XML comments into documentation files that can be incorporated as part of an application's written documentation. Visual Studio.NET goes beyond that by actually using XML comments within the integrated development environment (IDE) to provide on-the-fly information to developers.

Because of the power that XML comments provides, their usage is required in all DHSS applications. This section outlines how to implement them to a minimum degree of acceptability. It does not cover all the possibilities of XML comments and within the minimum acceptable constraints, developers are free to innovate further.

The most basic XML comment has the form of:

```
/// <summary>
      /// Developer supplied description of the item being commented. ///
    </summary>
```

Every line of an XML comment **must** begin with the sequence "///" and occurs on the line preceding the item that is being commented. In the Visual Studio.NET editor, if the programmer types the first three "///" characters, it will automatically fill in a skeleton XML comment containing a summary tag (and in some cases, other tags).

There are a number of tags that can be used other than the <summary> element, but it should be considered the minimal tag needed for an XML comment at DHSS for all

elements that require comments. The contents of the summary tag must be a relatively short but descriptive summary of the item being commented.

If the item being commented is a method/function, the skeleton comment generated by Visual Studio .NET will also contain tags to contain descriptions of the parameters passed to a function and the return value of the function (if applicable). If the item being commented is indeed a method/function, then it is required for these tags to be filled in as well. An example of this follows:

```
/// <summary>
/// Developer supplied description of the item being commented. ///
</summary>
/// <param name="firstParameter">Developer supplied description of ///
the first parameter to the method.</param>
/// <param name="secondParameter">Developer supplied description ///
of the second parameter to the method.</param>
/// <returns>Developer supplied description of the return
/// value of the method</returns>
```

The effect that supplying these comments will have in the Visual Studio .NET IDE is that when you now hover your mouse cursor over an object name in code, it will display the content of the summary. Also, when entering methods in code that have had their <param> elements defined, the code completion display will show those descriptions. This sort of information is extremely useful and speeds coding by preventing the developer from a need to constantly refer to the declarations of methods/functions in other code locations. It can also reduce errors by ensuring that the developer understands the implications of the method/function he/she is calling.

As for what items to actually supply comments, developers are required to supply XML comments for every class and every class data member and method (irrespective of access qualifiers). For classes and data members, only the <summary> is required, for method comments, the <param> elements are required as well. If the item is a private data member, the summary must indicate which public property (if any) through which it is accessed. If the item is a public property, the summary must describe the nature of the property but must not refer to the private data member (as this would violate encapsulation). If it is the case that a member has specific restrictions (such as read-only), that information must be provided in the summary as well.

Note: though the Visual Studio .NET IDE does much in supplying the skeleton comments when entering the initial comment delimiters, it should be remembered that if a method's signature changes at a later date, <param> elements may need to be removed or added.

The <remarks> element is another useful element but its usage is not always required. It should be used where there are relatively long blocks of text that may be useful in printed documentation but are not required for on-the-fly display. An example of its usage is:

```
/// <summary>
```

```
/// Developer supplied description of the item being commented. ///
</summary>
/// <param name="firstParameter">Developer supplied description of ///
the first parameter to the method.</param>
/// <param name="secondParameter">Developer supplied description
/// of the second parameter to the method.</param>
/// <returns>Developer supplied description of the return ///
value of the method</returns>
/// <remarks>Some very long text that would describe things
/// like operational assumptions, where further documentation may be found,
/// the expected type of input and output, and so on.</remarks>
```

As with the prior specifications on commenting, code that is not properly XML commented as spelled out in this document will not be accepted as a deliverable, irrespective of whether or not the author is an internal DHSS employee or an outside contractor.

Accessibility Guidelines

All HTML pages used in DHSS applications (including those automatically generated as part of a ASP.NET application) must be HTML 4.01 Transitional compliant and meet all WCAG 1.0 Priority 1 Checkpoints.

All WCAG 1.0 Priority 2 and 3 Checkpoints must be met as well whenever possible

Developers are **strongly** advised to avail themselves of the validation tools described in "Appendix B: Tools" in order to insure compliance with these requirements.

Vendor/Contract Guidelines

Vendors are held to the same standards as employees of DHSS in the terms of development standards. Any deliverable which fails to meet these standards will **NOT** be considered a completed deliverable. Exceptions to this policy can only be granted by the Director of IRM and the DHSS Manager of Base Technology and will require a written submission of the request for each exception including a detailed explanation of why the exception is required.

The purchase of any tools, books, training or other materials used for application development is always considered the responsibility of the contracting vendor.

If any tool other than the .NET Framework SDK, Visual Studio, CSE HTML Validator will be required for the proper maintenance and functioning of the system being developed, written permission for its use must be obtained from the Director of IRM and the DHSS Manager of Base Technology **before** it is used.

DHSS Unified Application Authentication and Authorization

General Information and Methodology

In an effort to reduce redundant coding and increase security, DHSS has implemented a unified application authentication and authorization framework. This system is a hybrid of Windows domain based authentication and application managed authorization. This means that users are identified by the supply of a Windows login name and domain password and then a SQL Server database supplies application-specific role and scope information to applications subscribing to the system.

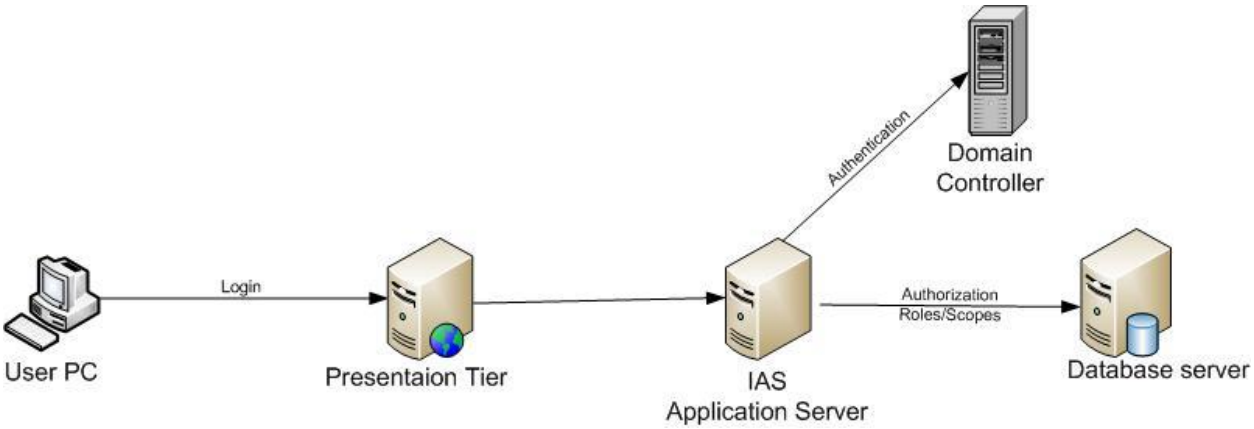
This approach was selected based on a desire to supply "single sign-on" capabilities in as much as was possible while not being tied to the unfriendly features of managing Windows network groups as role indicators. This means that not only can .NET applications use the system, but also almost all applications written in any modern programming language (there are specific concerns around subscribing a non -.NET application to this system though which are not covered in this manual and should a developer desire to subscribe to the unified system, he should speak with the DHSS Base Technology Manager on how best to implement its features).

The unified system is related and tied to another system known as the IRM Authorization System (IAS) which is used for the management of user accounts/profiles in applications. Details on IAS are outlined in other documents available from DHSS.

The basic component of the unified system is a web service that interfaces with the apps.dhss domain for authentication/identification and with the SQL Server database for role/scope information. Additionally, there are two assemblies from the DHSS Global Assemblies that are required for applications to subscribe if .NET based (any language can use the web service component though).

One benefit of this system is that is minimizes the number of times that a user's "true" password is passed across the network. In effect, after initial authentication, the system issues the user a "session key" that is only valid for as long as the user remains active. The session key is, in effect, a temporary password. The result of this is that application handling of real passwords is reduced while providing the continuous authentication that an asynchronous system that uses web services requires.

A model of the unified system (when applied to a .NET application) is depicted thusly:



The general flow of the system is as follows (actual step by step instructions on implementation of this in an application follow at the end of this section):

1. The user access the application's presentation layer login page.
2. The user supplies his login name and apps.dhss domain password to the application.
3. The application presentation layer passes the login name and encrypted password to the unified system web service.
4. The unified system web service first authenticates the password against the apps.dhss domain.
5. If the password is valid, the unified system then access the database store to create a session key and passes that back to the application presentation layer.
6. The presentation layer stores the login name and the session key in a web session for later use.
7. On each call to its web service layer, the application passes the login name and the session key in the WS-Security username token.
8. WS-Security on the application web service layer validates the session key automatically.
9. Within the specific web method, developer supplied code reconstitutes a DHSSPrincipal object based on the login name and session key.
10. Developer supplied code instantiates a PrincipalPermission object that indicates what identity/role/scope is required for execution of the web method.
11. The developer performs a SecurityAction.Demand to validate the current user against the permission object. If the current user does not meet the criteria required, then an exception is thrown. If the current user does indeed meet those criteria, then execution continues as expected.

Implementation Details

Following are step -by -step instructions on how add the DHSS Framework components necessary for using the unified system in your .NET application.

For Presentation Layer Projects

1. Add the following entry within the <appSettings> element of your applications web.config file:

    <add key="DHSSAuthorization.AuthorizationService.AuthorizationWS" value="http://unifiedAuthorization.dhss.state.de.us/AuthorizationWS.asmx"/>

2. Find the <authentication> element in the web.config file and change the mode to "Forms" Within that element, add the following entry:

    <forms loginUrl="login_file_name" name="authorizationCookie" timeout="60" path="/" />

where "login_file_name" is replaced with the name of the aspx file that serves as the login page for your application.
3. Find the <authorization> element in the web.config file and add the following element within it:

    <deny users="?" />

4. Add the following to the using section of your global.asax.cs file:

```
using System.Web.Security;
using DHSS.Framework.Security;
```

5. Implement the following code in your global.asax.cs file:

```
protected void Application_AuthenticateRequest(Object sender, EventArgs e)
{
	// Extract the forms authentication cookie
	string cookieName = FormsAuthentication.FormsCookieName;
	HttpCookie authCookie = Context.Request.Cookies[cookieName];

	if(null == authCookie)
	{
		// There is no authentication cookie.
		return;
	}

	FormsAuthenticationTicket authTicket = null;
	try
	{
		authTicket = FormsAuthentication.Decrypt(authCookie.Value);
	}
	catch(Exception ex)
	{
		// Log exception details (omitted for simplicity)
```

```
            return;
        }

        if (null == authTicket)
        {
            // Cookie failed to decrypt.
            return;
        }

        // This principal will flow throughout the request.
        DHSSPrincipal principal = new DHSSPrincipal(authTicket.Name,
        authTicket.UserData)

        // Attach the new principal object to the current HttpContext object
        Context.User = principal;
    }
```

6. Add the following to the using section of your login_file_name.aspx.cs file:

```
using System.Web.Security;
using DHSS.Framework.Security.AuthorizationService;
using DHSS.Framework.Security;
```

7. Implement the following code in your login_file_name.aspx.cs (see comments in the code for parts that require customization to your application):

```
private void DHSSAuthenticateUser()
{
    //this sets up a variable to store an output parameter from the
    //DHSSFormsAuthentication library

    HttpCookie authCookie;

    //replace <your_application_name> below with the unique application
    //key name for your application



    AuthorizationSession authorizationSession =
    DHSSFormsAuthentication.AuthenticateForm( txtUserName.Text,
    txtPassword.Text,"<your_application_name>", out authCookie);

    if (authorizationSession == null)
    {
        //a null here means that the system failed to authenticate the
        //user within this if statement, execute the actions you wish
        //to take in this event (such as notifying the user via a
        //message on his screen, for example.

        lblMessage.Text = "Your login failed. Please try again!";

        return;
    }

    Session["AuthorizationSession"] = authSession;
    Response.Cookies.Add(authCookie);

    // Redirect the user to the originally requested page
    Response.Redirect(FormsAuthentication.GetRedirectUrl(authorizationSes
    sion.UserName
}
```

8. Any actions you execute to log a user in should execute the DHSSAuthenticateUser method before it takes any other actions. This would often be done within the code for a "Login" button on the login screen, but may take other formats as well.

For Web Service Layer Projects

1. Add the following entries within the <appSettings> element of your applications web.config file:

<add key="DHSSAuthorization.AuthorizationService.AuthorizationWS" value="http://unifiedauthorizationTEST.dhss.state.de.us/AuthorizationWS.asmx"/>

<add key="DHSSPasswordProvider.AuthorizationService.AuthorizationWS" value="http://unifiedauthorizationTEST.dhss.state.de.us/AuthorizationWS.asmx"/>

Please note that these implementation details only enable your application for the basic framework on the unified system (i.e., logging into the application).

Required Database Elements

In regards to database table design in accordance with the DHSS Framework each table for auditing reasons must include the following fields in this exact order:

[PrimaryKey_IDNO] [int] IDENTITY(1,1) NOT NUL,
[FirstInsertedBy_IDNO] [int] NOT NULL
[FirstInserted_DTTM] [datetime] NOT NULL
[LastSavedBy_IDNO] [int] NOT NULL
[LastSaved_DTTM] [datetime] NOT NULL
[Concurrency_DTTM] [datetime] NOT NULL

* Note the PrimaryKey_IDNO field name can be changed accordingly for each table such as <name>_IDNO.

[Example Persons_T1 Table]

[Person_IDNO] [int] IDENTITY(1,1) NOT NUL,
[FirstInsertedBy_IDNO] [int] NOT NULL
[FirstInserted_DTTM] [datetime] NOT NULL
[LastSavedBy_IDNO] [int] NOT NULL
[LastSaved_DTTM] [datetime] NOT NULL
[Concurrency_DTTM] [datetime] NOT NULL
[LastName_NAME] [varchar(35)] NOT NULL
[FirstName_NAME] [varchar(35)] NOT NULL

The DHSS Framework specifically looks for these fields when using the standard framework BusinessObject methods which are: Add(), Save(), Load(int), and List().

Required "NULL" Values

The DHSS Framework DataAccess layer is designed to always assume a NULL value will be never come from the database. This is done, because many errors often occur within code when value is converted or parsed incorrectly due to it being NULL. To alleviate this concern DHSS has required certain database datatypes to always have a value if it is considered to be "NULL". These are outlined as:

DataType	DHSS Null Value
Varchar,Char	"" – Empty String
DateTime	12/31/9999 23:59:59.997 – High DTTM value
Integer	-2147483648 – Min value of a 32-bit integer

The other benefit utilizing these values is that they work seamlessly with the DHSS .NET Web Controls. These are used to display data easily and correctly without having to code cumbersome validation routines.

More information can be found and utilized in the DHSS Web Application Template documentation. This is a separate document that comes with a starter DHSS .NET Web Application solution for developer use. See Appendix F: DHSS Framework/Template for details.

Web Service Guidelines

Any web service that is created to utilize DHSS systems must incorporate the DTI XML Firewall software appliance if one of the following is true:

- It is to be consumed by external entities outside the State of Delaware.
- It is to be consumed by another State of Delaware Department outside of DHSS.

The DTI XML Firewall will not in influence on how a web service is programmed. It is a software appliance to enhance security. For more information please contact the DHSS Base Technology group.

Appendix A: Definitions

Capitalization Styles Defined

There are three possible capitalization styles:

Pascal case

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized.

Example: BackColor,DataSet

Camel case

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized.

Example: numberOfDays, isValid

Uppercase

All letters in the identifier are capitalized.

Example: ID, PI

Hungarian Type Notation Defined

Hungarian notation is any of a variety of standards for organizing a computer program by selecting a schema for naming your variables so that their type is readily available to someone familiar with the notation. It is in fact a commenting technique.

Example: strFirstName, iNumberOfDays

There are different opinions about using this kind of type notation in programming nowadays. Some say that it's useful, and it should be used everywhere to enhance clarity of your code. Others say it just obfuscates your code, because it has no real advantage in modern programming environments.

Our point of view is a moderated one: use it wisely, meaning, we only use Hungarian notation for items where type may be unclear **and** necessary. In effect, this means that the notation is only used in the declaration of visual form objects. Hungarian notation is unnecessary for primitive types in C# because variables are generally declared and used in the same code blocks, meaning that the type of the variable is already within easy reach. There are some exceptions to this general rule, but the trade -off in code clarity and ease of reading makes dropping the ubiquitous type prefixes of older languages worthwhile. The end of this document lists the prefixes that should be used for these type of objects.

There are however some prefixes not related to type determination that are standard. These are explained in other parts of this manual. They are distinct from the prefixes previously discussed because rather than indicating type, they indicate scope.

Definitions of the Types of Web Form Controls

When you create Web Forms pages, you can use the User controls and the HTML Server controls. Each of these controls is explained in the following sections. Any page may contain one or both of these types of controls.

User Controls

Controls that you create as Web Forms pages. You can embed Web Forms user controls in other Web Forms pages, which is an easy way to create menus, toolbars, and other reusable elements.

HTML Server Controls

HTML server controls are HTML elements containing attributes that make them visible to, and programmable on, the server. By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program on the server.

The object model for HTML server controls maps closely to that of the corresponding elements. For example, HTML attributes are exposed in HTML server controls as properties.

Any HTML element on a page can be converted to an HTML server control. Conversion is a simple process involving just a few attributes. As a minimum, an HTML element is converted to a control by the addition of the attribute RUNAT="SERVER". This alerts the ASP.NET page framework during parsing that it should create an instance of the control to use during server-side page processing.

The page framework provides predefined HTML server controls for the HTML elements most commonly used dynamically on a page: forms, the HTML <INPUT> elements (text box, check box, Submit button, and so on), list box (<SELECT>), table, image, and so on. These predefined HTML server controls share the basic properties of the generic control, and in addition, each control typically provides its own set of properties and its own event.

Note: In some situations, server controls require client script in order to function properly. If a user has disabled scripting in the browser, the controls might not function as you intend.

HTML server controls offer the following features:

1. An object model that you can program against on the server using the familiar object –oriented techniques. Each server control exposes properties that allow you to manipulate the control's HTML
attributes programmatically in server code.
2. A set of events for which you can write event handlers in much the same way you would in a client-based form, except that the event is handled in server code.
3. The ability to handle events in client script.
4. Automatic maintenance of the control's state. If the form makes a round trip to the server, the values that the user entered into HTML server controls are automatically maintained when the page is sent back to the browser.
5. Interaction with validation controls so you can easily verify that a user has entered appropriate information into a control.
6. Data binding to one or more properties of the control.
7. Support for HTML 4.1 styles if the Web Forms page is displayed in a browser that supports cascading style sheets.
8. Pass -through of custom attributes. You can add any attributes you need to an HTML server control and the page framework will read them and render them without any change in functionality. This allows you to add browser -specific attributes to your controls.

Other Definitions

Note: some of these definitions are specific to methodologies in use at DHSS. They should not always be assumed to apply generically to the rest of the world.

Static Web Site

A web site in which the content and information is largely or wholly fixed and does not change from moment to moment. This generally means that content is only changed due to the actions of a developer or content writer. These sites are generally open to the public, though in some cases the definition of "public" may vary.

Interactive/Dynamic Web Site

A web site in which some significant portion of the content presented varies based on user input. Though this type of web site often features web forms, the forms are often used only for transitory input and the data captured is often not considered reliable. These types of sites are differentiated from browser based applications by the fact that even if they are password protected, there is no process in place to limit access to genuinely identifiable individuals.

Browser Based Application

Systems or web servers engineered in such a way that they are only accessible to specifically identifiable users where such identity has been established through a formal, human controlled process. Such systems are used for the collection and maintenance of important data that needs a high degree of reliability.

Layer

A logical division within an application. Does not mean the same thing as "tier"

Tier

A physical division within a distributed application. Does not mean the same thing as "layer"

Business Entity (BE) Classes

Classes/structures that are used for the transfer of data between layers of the application.

Business Logic Layer (BLL) Components

Classes that represent an object -oriented view of the application. Also known as the "Domain Model".

Data Access Layer (DAL) Components

Classes that provide a simple interface to the database. These classes do not contain business logic.

Appendix B: Tools

Development Tools

The .NET Framework version in use at DHSS is 2.0, 3.5, and 4.0 as of November 2014. However, this will be kept up to date with whatever the most current version happens to be.

DHSS employees currently use Visual Studio 2005, 2008, and 2010 - for actual programming. In some cases, there is development using a simple text editor (generally TextPad 4.x) and the .NET Framework command line compilers. Generally other parties developing for DHSS will be expected to use one of these two methods, but exceptions may be made in certain cases. All developers must be sure to keep their copies of Visual Studio up to date with all Microsoft issues patches and updates.

Many applications at DHSS make use of the ODBC provider for .NET, this is perfectly acceptable for access to non OLE DB compliant databases but OLE DB or SQL Server connectivity should always be used when possible.

Validation Tools

There are several validation tools used by DHSS that will help insure that your code meets these standards. They are:

CSE HTML Validator

Web site: http://www.htmlvalidator.com

CSE HTML Validator is used to check pages for HTML 4.01 compliance.

Using These Tools

Please be advised that in some cases, these tools may have more or less restrictive guidelines than those outlined in this document. Their output should always be weighed against what the DHSS standards say.

It is **HIGHLY** recommended that all developers creating applications for DHSS use these tools as it will help catch many standards failures. Furthermore, it is also **HIGHLY** recommended that these tools be used from the very beginning of the project and on a regular basis to keep any fixes required to a manageable level. Developers who wait till the end of their project to use these tools will likely find that significant retooling will be required to achieve compliance.

Recommended Settings for Visual Studio

- Always build your projects with warning level 4 (the VS .NET default).
- Treat warnings as errors in all builds (this is NOT the VS .NET default).
- Never suppress specific compiler warnings (VS .NET default).

Appendix C: Hungarian Notation Prefixes

Object	Prefix
Textbox	txt
Label	lbl
Button	btn
List Box	lst
Drop Down List Box	ddl
Radio Button	rb
Check Box	cb
Check Box List	cbl

Radio Button List	rbl
Panel	pnl
Link Button	lbt
DataGrid/GridView	dtg
Reapeater	rpt
Data List	dlist
Validator	valid

Note: *should it be necessary to add to or correct the above list, please e -mail the current maintainer as named at the head of this document.*

Appendix D: DHSS Global Assemblies

There are a number of in-house developed custom assemblies that are used throughout DHSS .NET applications. These assemblies will be delivered through the means of the DHSS Web Application Template.

Note to off-site vendors: copies of these assemblies will be provided to you upon commencement of any development project for DHSS. Since these copies will be independent of the central location at DHSS, it is the responsibility of vendors to insure that their copies are always kept up to date with any changes in the central location copies.

Appendix E: Frequently Asked Questions about DHSS .NET Development

This section is a repository of questions that have come up regarding matters in this manual. Some of them simply clarify items while others offer justifications for why a policy is what it is. Though the DHSS applications management team is always interested in the free discussion of the policies of this manual (and more than willing to revise our positions when necessary), this should not be viewed as an open invitation for debate.

My application is really small, only meant for temporary use, is unimportant, and/or doesn't really need to be worried over too much. Do these policies still apply? Why so much fuss over something like that?

> The short answer is that yes, these policies apply irrespective of the perceived size and importance of your application. There are a number of reasons that DHSS requires a consistent framework for all of its application, some of them are:

- A consistent framework allows staff to move between applications with a minimum of effort, reducing retraining and maximizing productivity.
- Applications have a tendency to grow beyond their intended purposes. Therefore, today's small, throw-away application may become a critical system tomorrow.
- Some of our policies enforce security considerations that apply no matter how important the application is.

Why do you require that all database interaction take place within the context of a web service?

One requirement of our security model is the use of a multi-tiered setup for all applications. This hides as much code as possible from the eyes of the public world. Having all database operations coming from web services located in a secured zone accomplishes this and allows the lock down of all but minimal communications protocols from the outside world into our network.

Aren't there other ways to effect the multi-tiered application model you describe above without using web services/.NET Remoting?

Most assuredly there are other methods. The use of web services/.NET Remoting though offer the most open environment so that if non-.NET applications ever require access to the resources as well, no other work on the application tier will be required.

Why do you not allow any business logic to reside within stored procedures? Isn't it the case that placing business logic in stored procedures could reduce the number of database round-trips required for operations?

This question over stored procedures has spurred much debate in the industry. Those in favor of the practice usually point out that having the business logic on the database server can minimize round-trips to the database and that the logic can be shared amongst all applications that may access the data, reducing the need to maintain multiple code sets.

DHSS has fallen in the other camp though, and for good reasons. Writing robust logic into SQL procedural language is difficult and prone to errors. When errors are discovered, they are difficult to debug. For those reasons, it means it is often unfeasible to have **ALL** business logic reside in the stored procedures. Keeping all of the business logic in one layer is an important consideration for maintainability, so DHSS has decided that since all logic cannot reside at the database server, it must entirely reside elsewhere.

The DHSS framework places all business logic at the application server tier. Done this way, it actually negates the argument for business logic stored procedures reducing code maintenance since if all interaction with the database takes place from that tier, no application need ever repeat the code, all run the code from the application tier.

Another reason for keeping logic away from the database server is that database servers remain the most finite of resources involved in .NET applications (and really in any sort of application). It is relatively easy to add more processing power to the application and web tiers of an application but extremely difficult to add more to the database tier. This means that it is in our interest to keep processing on the database tier to a minimum and thereby maximize the scalability of our applications.

Finally, though the argument that reducing database server round-trips does have merit, it is the simple case that the other factors against the practice of business logic in stored procedures outweigh it as a hazard. It is important to remember here that the guiding principle of DHSS application development is in increasing maintainability with speed optimization remaining a secondary concern.

Would you allow the usage of other frameworks such as Model–view–controller (**MVC**) or **Model**-**View**-**Presenter (MVP) that differ from the DHSS Framework web form implementation?**

While DHSS has nothing against these methods it is still in our best interest to use the Web Form approach. Thus requiring DHSS and contractual staff to utilize the DHSS Framework only.

Appendix F: DHSS Framework/Template

All new web applications developed by and for DHSS must utilize the DHSS Framework and Template standards. This information can be obtained through the Division of Management Services/Information Resource Management Technical Project Manager upon a contract being signed by all parties and a DHSS Non-Disclosure Agreement (below) signed by the vendor.

All custom off the shelf web applications that will be utilized by DHSS must adhere to the DHSS Framework/Template standards. If it is thought that this is implausible for a certain scenario a written waiver with justification must be requested of the DHSS Manager of Base Technology or the Director of IRM.

	*DELAWARE HEALTH AND SOCIAL SERVICES* Division of Management Services *"DMS – Serving Those Who Serve Delaware"* **Information Resource Management**	**NON-DISCLOSURE AGREEMENT**

As a condition of receiving access to State of Delaware information technology policies and standards, vendor hereby agrees to the following:

1. That the disclosure of Information by State of Delaware is in strictest confidence and thus vendor will:
    a. Not disclose to any other person this Information.
    b. Use at least the same degree of care to maintain the Information secret as the vendor uses in maintaining as secret its own secret information, but always at least a reasonable degree of care.
    c. Use the Information only for the purpose of preparing a response to a State of Delaware Request for Proposal.
    d. Restrict disclosure of the Information solely to those employees of vendor having a need to know such Information in order to accomplish the purpose stated above.
    e. Advise each such employee, before he or she receives access to the information, of the obligations of vendor under this Agreement, and require each such employee to maintain those obligations.
    f. Within fifteen (15) days following request of State of Delaware, return to State of Delaware all documentation, copies, notes, diagrams, computer memory media and other materials containing any portion of the information, or confirm to State of Delaware, in writing, the destruction of such materials.

2. This Agreement imposes no obligation on vendor with respect to any portion of the Information received from State of Delaware which was known to the vendor prior to disclosure by State of Delaware.

3. The Information shall remain the sole property of State of Delaware.

Vendor Name _____

Vendor Representative Name: _____

Signature: _____

Vendor Representative Title: _____

Date: _____