



TEXAS TECH UNIVERSITY
HEALTH SCIENCES CENTER™
EL PASO

TTUHSC El Paso

Programming Languages Coding Standards Best Practices

Version 1.2

6/10/2021

Prepared for:

Texas Tech University

Health Sciences Center

El Paso



Document Revision History

Document Version	Revised Date	Modified By	Section, Page(s) and Text Revised	Comments
1.0	4/20/2016	Adrian Cazares	All the document	Initial version
1.1	4/21/2021	Ivan Calderon	Accessibility Guidelines Section 3	Added Accessibility Guidelines (WCAG)
1.2	06/10/2021	Edgar Sosa	Secure Website	Added Security Website, formatted file.

Table of Contents

1	Introduction	5
1.1	Purpose	5
1.2	Scope	5
2	Technologies and Architecture	5
2.1	Programming Languages and project specifications	5
2.2	Applications Architecture	5
3	Accessability Guidelines	7
3.1	Accessability Guidelines	7
4	Programming	7
4.1	Namespaces	7
4.2	Classes & Structures	8
4.3	Interfaces	8
4.4	Constants	9
4.5	Enumerations.....	9
4.6	Variables, Fields & Parameters	10
4.7	Properties.....	10
4.8	Methods	11
4.9	Event Handlers	11
4.10	Error Handling.....	11
5	Formatting	14
5.1	Class Layout	14
5.2	Indicating Scope	14
5.3	Indentation & Braces	15
5.4	White space	16
5.5	Long lines of code.....	16
6	Commenting	17
6.1	Intellisense Comments	17
6.2	End-Of-Line Comments	17
6.3	Single Line Comments.....	17
6.4	// TODO: Comments	17

6.5	C-Style Comments.....	17
7	Capitalization.....	18
7.1	Capitalization	18
8	C# Golden Rules	19
9	Database Standards	20
9.1	General Rules for all database objects.....	20
9.2	Tables	20
9.3	Columns.....	20
9.4	Views	21
9.5	Stored Procedures.....	21
9.6	Functions	21
9.7	Variables.....	21
9.8	DB Coding Conventions	22
10	Secure Website	25
10.1	Authentication and Authentication	25
10.1.1	Install DotNetCasClient using NuGet.....	25
10.1.2	Forms Authentication	25
10.1.3	Using Authorize and AllowAnonymous Attributes.....	26
10.1.4	Role base authentication	27
10.2	Cross-Site Request Forgery (CSRF).....	28
10.3	Cross Site Scripting (XSS).....	30
10.3.1	Regular Expression Attribute and object model.....	30
10.3.2	URL Encoding	30
10.3.3	SQL Injection	31
10.3.4	Secure Cookies.....	32
10.4	Custom Error Page for Error Handling	34

1 Introduction

1.1 Purpose

The purpose of this document is to provide coding style standards for the development of source code written in C#. Adhering to a coding style standard is an industry proven best-practice for making team development more efficient and application maintenance more cost-effective. While not comprehensive, these guidelines represent the minimum level of standardization expected in the source code of projects within TTUHSC El Paso.

1.2 Scope

This document provides guidance on the architecture, formatting, commenting, naming, and programming style of C# source code and is applicable to component libraries, web sites, web services, and client – server applications.

2 Technologies and Architecture

2.1 Programming Languages and project specifications

As part of the TTUHSC El Paso programming standarization the following languages and technologies are defined as primary components to work with when developing new projects:

- A) Visual Studio Professional 2017 Framework 4.7.1
- B) Web applications: ASP.NET with MVC5.
- C) Front End: ASP.Net, CSS, Javascript, and Bootstrap templates to build responsive web pages (automatic adjustment when using wide screens, tablets or smart phones).
- D) Back End: C# (only in very specific cases we will use VB.Net).
- E) API: WCF (Windows Communication Fundation).
- F) DBMS: MS SQL Server 2016 (Database, SP, UDF, Database Mail, Jobs) and Oracle 12c for very specific cases.
- G) Authentication: CAS authentication method and RoleProvider for granular permissions, the only exeption is to use eRaider authentication when the application was built in ASP classic.
- H) Repository: our current version controller is Git, for further information review the document [Source Control Policies Installation and User Guide.pdf](#).
- I) When creating a new project the naming of the solution, database, and repository should match.

2.2 Applications Architecture

The applications shall have at least the following 3 layers in the solution:

- A) Presentation Layer (Views, Models and Controllers).
- B) Business Logic Layer (Functions or methods with business logic)
- C) Data Access Layer (Data entities, DB connections, Entity framework)

The applications might have more layers depending on the application needs, and the layers supporting either front-end or back-end will be Class Libraries.

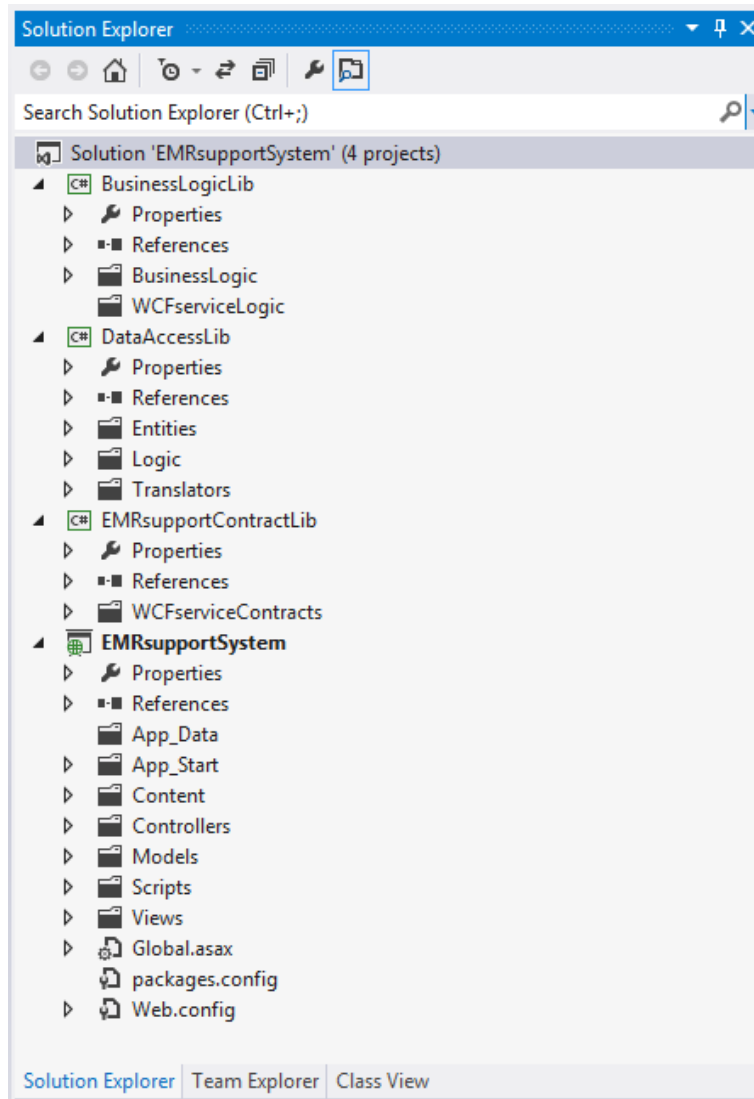


Fig 1.0 Solution Architecture example.

3 Accessibility Guidelines

In order to comply with WCAG 2.0, the scope for all new applications created and developed at TTUHSC must adhere to the Accessibility Guidelines mentioned in the Accessibility Guidelines v1.1 document.

There are 12 basic principles that every application should follow and pursue in order to broaden its use scope to specific demographics with special needs.

3.1 Accessibility Guidelines

Under the principles are 12 guidelines that provide basic goals that authors/developers should work toward in order to make content more accessible to users with different disabilities.

Accessibility Guidelines V1.1 TTUHSC El Paso. Adrian Cazares et al. 2020.

<https://sharepoint16.elpaso.ttuhscc.edu/support/IT/IMS/trainings/SOP%20Guides%20Manuals/AccessibilityGuidelines.docx>

4 Programming

4.1 Namespaces

Namespaces represent the logical packaging of component layers and subsystems. The declaration template for namespaces will be: Company.Product.Library.SubsystemName (or subfolder).

Examples:

TTUHSC.CIS.DataAccess.Translators

TTUHSC.RMS.BusinessLogic

Guidelines:

- 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.
- Use Pascal casing when naming namespaces.

4.2 Classes & Structures

Classes and structures represent the 'Nouns' of a system. As such, they should be declared using the following template: Noun + *Qualifier(s)*. Classes and structures should be declared with qualifiers that reflect their derivation from a base class whenever possible.

Examples:

CustomerForm: Form

CustomerCollection: CollectionBase

Guidelines:

- Use Pascal casing when naming classes and structures.
- Classes and structures should be broken up into distinct #regions as described in the class layout guidelines.
- All public classes and their methods should be documented using the Intellisense triple slash '///' comments built into Visual Studio.Net. Use this comment style to document the purpose of the class and its methods.
- Default values for fields should be assigned on the line where the field is declared. These values are assigned at runtime just before the constructor is called. This keeps code for default values in one place, especially when a class contains multiple constructors.

4.3 Interfaces

Interfaces express behavior contracts that derived classes must implement. Interface names should use Nouns, Noun Phrases, or Adjectives that clearly express the behavior that they declare.

Examples:

IComponent

IFormattable

ITaxableProduct

Guidelines:

- Prefix interface names with the letter 'I'.
- Use Pascal casing when naming interfaces.

4.4 Constants

Constants and static read-only variables should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Example:

```
public const int DefaultValue = 25;
public static readonly string DefaultDatabaseName = "Membership";
```

Guidelines:

- Use Pascal casing when naming constants and static read only variables.
- Prefer the use of `static readonly` over `const` for public constants whenever possible. Constants declared using `const` are substituted into the code accessing them at compile time. Using `static readonly` variables ensures that constant values are accessed at runtime. This is safer and less prone to breakage, especially when accessing a constant value from a different assembly.

4.5 Enumerations

Enumerations should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Example:

```
/// <summary>
/// Enumerates the ways a customer may purchase goods.
/// </summary>
[Flags]
public enum PurchaseMethod
{
    All           = ~0,
    None         = 0,
    Cash         = 1,
    Check        = 2,
    CreditCard   = 4,
    DebitCard    = 8,
    Voucher      = 16,
}
```

Guidelines:

- Use Pascal casing when naming enumerations.
- Use the [Flags] attribute only to indicate that the enumeration can be treated as a bit field; that is, a set of flags.

4.6 Variables, Fields & Parameters

Variables, fields, and parameters should be declared using the following template: *Adjective(s) + Noun + Qualifier(s)*

Examples:

```
int lowestCommonDenominator = 10;
float firstRedBallPrice = 26.0f;
```

Guidelines:

- Use Camel casing when naming variables, fields, and parameters.
- Define variables as close as possible to the first line of code where they are used.
- Assign initial values whenever possible. The .NET runtime defaults all unassigned variables to 0 or null automatically, but assigning them proper values will alleviate unnecessary checks for proper assignment elsewhere in code.
- Avoid meaningless names like i, j, k, and temp. Take the time to describe what the object really is (e.g. use index instead of i; use swapInt instead of tempInt).
- Use a positive connotation for boolean variable names (e.g. isOpen as opposed to notOpen).

4.7 Properties

Properties should be declared using the following template: *Adjective(s) + Noun + Qualifier(s)*

Examples:

```
public TotalPrice
{
    get
    {
        return this.totalPrice;
    }
    set
    {
        // Set value and fire changed event if new value is different
        if( !object.Equals( value, this.totalPrice ) )
        {
            this.totalPrice = value;
            this.OnTotalPriceChanged();
        }
    }
}
```

Guidelines:

- Use the common prefixes for inspection properties (properties that return query information about an object).
- When there is a property setter that sets another property:

- If the code in the other property sets a private member field in the same class, the field should be set directly, without calling the property setter for that field.
- If a property setter sets a private field that would normally be set via another property setter, the originating setter is responsible for firing any events the other setter would normally fire (e.g. Changed events).
- If a value that needs to be set that does NOT correspond to a private field, then an appropriate property setter or method should be called to set the value.

4.8 Methods

Methods should be named using the following format: Verb + *Adjective(s)* + Noun + *Qualifier(s)*

Example:

```
private Ball FindRedCansByPrice(float price, ref int canListToPopulate, out int numberOfCansFound)
```

Guidelines:

- Parameters should be grouped by their mutability (from least to most mutable) as shown in the example above.
- If at all possible, avoid exiting methods from their middles. A well written method should only exit from one point: at its end.
- Avoid large methods. As a method's body approaches 20 to 30 lines of code, look for blocks that could be split into their own methods and possibly shared by other methods.
- If you find yourself using the same block of code more than once, it's a good candidate for a separate method.
- Group like methods within a class together into a region and order them by frequency of use (i.e. more frequently called methods should be near the top of their regions).

4.9 Event Handlers

Event handlers should be declared using the following format: `ObjectName_EventName`

Example:

```
private HelpButton_Click(object sender, EventArgs e)
```

4.10 Error Handling

Use exceptions only for exceptional cases, not for routine program flow. Exceptions have significant

performance overhead.

Guidelines:

- Pass a descriptive string into the constructor when throwing an exception.
- Use grammatically correct error messages, including ending punctuation. Each sentence in the description string of an exception should end in a period.
- If a property or method throws an exception in some cases, document this in the comments for the method. Include which exception is thrown and what causes it to be thrown.

- Example: Comment for Order.TotalCost property might read "Gets or sets the total cost of an Order. If the TotalCost property is set when the cost should be calculated, an InvalidOperationException is thrown."

- Use the following exceptions if appropriate:

- ArgumentException (and ArgumentNullException, ArgumentOutOfRangeException): Used when checking for valid input parameters to method.
- InvalidOperationException: Used when a method call is invalid for the current state of an object.

Example: TotalCost cannot be set if the cost should be calculated. If the property is set and it fails this rule, an InvalidOperationException is thrown.

- NotSupportedException: Used when a method call is invalid for the class.

Example: Quantity, a virtual read/write property, is overridden by a derived class. In the derived class, the property is read-only. If the property is set, a NotSupportedException is thrown.

- NotImplementedException: Used when a method is not implemented for the current class.

Example: An interface method is stubbed in and not yet implemented. This method should throw a NotImplementedException.

- Derive your own exception classes for a programmatic scenarios. All new derived exceptions should be based upon the core Exception class.

Example: DeletedByAnotherUserException : Exception. Thrown to indicate a record being modified has been deleted by another user.

- Rethrow caught exceptions correctly.

The following example throws an exception caught and rethrown incorrectly:

```
catch(Exception ex)
{
    LogManager.Publish(ex);
}
```



```
    throw ex; // INCORRECT – we lose the call stack of the exception  
}
```

We log all unhandled exceptions in our applications, but may sometimes throw them again to let the higher level systems determine how to proceed. The problem comes in with the throw – it works much better to do this:

```
    catch(Exception ex)  
    {  
        LogManager.Publish(ex);  
        throw; // CORRECT - rethrows the exception we just caught  
    }
```

Notice the absence of an argument to the throw statement in the second variation.

The difference between these two variations is subtle but important. With the first example, the higher level caller isn't going to get all the information about the original error. The call stack in the exception is replaced with a new call stack that originates at the "throw ex" statement – which is not what we want to record. The second example is the only one that actually re-throws the original exception, preserving the stack trace where the original error occurred.

5 Formatting

5.1 Class Layout

Classes should be organized into regions within an application using a layout determined by the best practices. These may be based on accessibility, type, or functionality.

Example:

```
// Class layout based on accessibility
class Purchasing
{
    #region Main

    #region Public

    #region Internal

    #region Protected

    #region Private

    #region Extern

    #region Designer Generated Code

}
```

Guidelines:

- Use the same layout consistently in all classes in an application.
- Omit regions if their associated class elements are not needed.
- The Designer Generated Code region created by Visual Studio's Visual Designer should never be modified by hand. It should contain only code generated by the designer.

5.2 Indicating Scope

Indicate scope when accessing all static and non-static class members. This provides a crystal clear indication of the intended use of the member. VisualStudio.NET intellisense is automatically invoked when using this practice, providing a list of all available class members. This helps prevent unnecessary typing and reduces the risk of typographic errors.

Example:

```
string connectionString = DataAccess.DefaultConnectionString;
float amount = this.CurrentAmount;
this.discountedAmount = this.CalculateDiscountedAmount(amount, this.PurchaseMethod);
```

Guidelines:

- Include the `this` keyword before all member fields, properties and methods.
- Include the name of the class before all static fields, constants, fields, and methods.

5.3 Indentation & Braces

Statements should be indented (using tabs) into blocks that show relative scope of execution. A consistent tab size should be used for all indentation in an application. Braces, when necessary, should be placed directly below and aligned with the statement that begins a new scope of execution. Visual Studio.NET includes a keyboard short-cut that will automatically apply this format to a selected block of code.

Example:

```
float CalculateDiscountedAmount(float amount, PurchaseMethod purchaseMethod)
{
    // Calculate the discount based on the purchase method
    float discount = 0.0f;
    switch(purchaseMethod)
    {
        case PurchaseMethod.Cash:
            // Calculate the cash discount
            discount = this.CalculateCashDiscount(amount);
            Trace.WriteLine("Cash discount of {0} applied.", discount);
            break;

        case PurchaseMethod.CreditCard:
            // Calculate the credit card discount
            discount = this.CalculateCreditCardDiscount(amount);
            Trace.WriteLine("Credit card discount of {0} applied.", discount);
            break;

        default:
            // No discount applied for other purchase methods
            Trace.WriteLine("No discount applied.");
            break;
    }

    // Compute the discounted amount, making sure not to give money away
    float discountedAmount = amount - discount;
    if(discountedAmount < 0.0f)
    {
        discountedAmount = 0.0f;
    }
    LogManager.Publish(discountedAmount.ToString());

    // Return the discounted amount
    return discountedAmount;
}
```

5.4 White space

Liberal use of white space is highly encouraged. This provides enhanced readability and is extremely helpful during debugging and code reviews. The indentation example above shows an example of the appropriate level of white space.

Guidelines:

- Blank lines should be used to separate logical blocks of code in much the way a writer separates prose using headings and paragraphs. Note the clean separation between logical sections in the previous code example via the leading comments and the blank lines immediately following.
- Single spaces should be used to separate logical elements within individual statements. This can be seen clearly in the `CalculateDiscountedAmount` and `switch` statements in the preceding example. Note the spaces immediately after opening '('s and before closing ')'s.

5.5 Long lines of code

Comments and statements that extend beyond 80 columns in a single line can be broken up and indented for readability. Care should be taken to ensure readability and proper representation of the scope of the information in the broken lines. When passing large numbers of parameters, it is acceptable to group related parameters on the same line.

Example:

```
string Win32FunctionWrapper(int arg1, string arg2, bool arg3)
{
    // Perform a PInvoke call to a win32 function,
    // providing default values for obscure parameters,
    // to hide the complexity from the caller
    if(Win32.InternalSystemCall(
        null,
        arg1, arg2,
        Win32.GlobalExceptionHandler,
        0, arg3,
        null)
    {
        return "Win32 system call succeeded.";
    }
    else
    {
        return "Win32 system call failed.";
    }
}
```

Guidelines:

- When breaking parameter lists into multiple lines, indent each additional line one tab further than the starting line that is being continued.
- Group similar parameters on the same line when appropriate.
- When breaking comments into multiple lines, match the indentation level of the code that is being

commented upon.

- Consider embedding large string constants in resources and retrieving them dynamically using the .NET ResourceManager class.

6 Commenting

6.1 Intellisense Comments

Use triple slash '///' comments for documenting the public interface of each class. This will allow Visual Studio.Net to pick up the method's information for Intellisense. These comments are required before each public, internal, and protected class member and optional for private members.

6.2 End-Of-Line Comments

Use End-Of-Line comments only with variable and member field declarations. Use them to document the purpose of the variable being declared.

Example:

```
private string name = string.Empty; // Name of control (defaults to blank)
```

6.3 Single Line Comments

Use single line comments above each block of code relating to a particular task within a method that performs a significant operation or when a significant condition is reached. Comments should always begin with two slashes, followed by a space.

Example:

```
// Compute total price including all taxes
float stateSalesTax = this.CalculateStateSalesTax(amount, Customer.State);
float citySalesTax = this.CalculateCitySalesTax(amount, Customer.City);
float localSalesTax = this.CalculateLocalSalesTax(amount, Customer.Zipcode);
float totalPrice = amount + stateSalesTax + citySalesTax + localSalesTax;
Console.WriteLine("Total Price: {0}", totalPrice);
```

6.4 // TODO: Comments

Use the // TODO: comment to mark a section of code that needs further work before release. Source code should be searched for these comments before each release build.

6.5 C-Style Comments

Use c-style /*...*/ comments only for temporarily blocking out large sections of code during development

and debugging. Code should not be checked in with these sections commented out. If the code is no longer necessary, delete it. Leverage your source control tools to view changes and deletions from previous versions of the code. If code must be checked in with large sections commented out, include a `// TODO:` comment above the block commented out describing why it was checked in that way.

7 Capitalization

7.1 Capitalization

Follow the standard set by the .NET framework team by using only three capitalization styles: **Pascal**, **Camel**, and **Upper** casing.

Examples:

Identifier Type	Capitalization Style	Example(s)
Abbreviations	Upper	ID, REF
Namespaces	Pascal	AppDomain, System.IO
Classes & Structs	Pascal	AppView
Constants & Enums	Pascal	TextStyles
Interfaces	Pascal	IEditableObject
Enum values	Pascal	TextStyles.BoldText
Property	Pascal	BackColor
Variables, and Attributes	Pascal (public) Camel (private, protected, local)	WindowSize windowWidth, windowHeight
Methods	Pascal (public, private, protected) Camel (parameters)	ToString() SetFilter(string filterValue)
Local Variables	Camel	recordCount

Guidelines:

- In **Pascal** casing, the first letter of an identifier is capitalized as well as the first letter of each concatenated word. This style is used for all public identifiers within a class library, including namespaces, classes and structures, properties, and methods.
- In **Camel** casing, the first letter of an identifier is lowercase but the first letter of each concatenated word is capitalized. This style is used for private and protected identifiers within the class library, parameters passed to methods, and local variables within a method.
- **Upper** casing is used only for abbreviated identifiers and acronyms of four letters or less.

8 C# Golden Rules

The following guidelines are applicable to all aspects of C# development:

- Make code as simple and readable as possible. Assume that someone else will be reading your code.
- Prefer small cohesive classes and methods to large monolithic ones.
- Use a separate file for each class, struct, interface, enumeration, and delegate with the exception of those nested within another class.
- Write the comments first. When writing a new method, write the comments for each step the method will perform before coding a single statement. These comments will become the headings for each block of code that gets implemented.
- Use liberal, meaningful comments within each class, method, and block of code to document the purpose of the code.
- Mark incomplete code with `// TODO:` comments. When working with many classes at once, it can be very easy to lose a train of thought.
- **Never hard code** “magic” values into code (strings or numbers). Instead, define constants, static read-only variables, and enumerations or read the values from configuration or resource files.
- Prefer `while` and `foreach` over other available looping constructs when applicable. They are logically simpler and easier to code and debug.
- Use the `StringBuilder` class and its `Append()`, `AppendFormat()`, and `ToString()` methods instead of the string concatenation operator (`+=`) for much more efficient use of memory.
- Be sure `Dispose()` gets called on `IDisposable` objects that you create locally within a method. This is most commonly done in the `finally` clause of a `try` block. It’s done automatically when a `using` statement is used.
- Never present debug information to yourself or the end user via the UI (e.g. `MessageBox`). Use tracing and logging facilities to output debug information.
- Gaps and exceptions to these guidelines should be discussed and resolved with your application architect.
- Follow the style of existing code. Strive to maintain consistency within the code base of an application. If further guidance is needed, look to these guidelines and the .NET framework for clarification and examples.

9 Database Standards

9.1 General Rules for all database objects

- Try to limit the name to 30 characters (shorter is better).
- Avoid using underscores even if the system allows it, only special cases are allowed.
- Try to avoid numbers – and limit the use of underscores to meet standards for constraints.
- Limit the use of abbreviations (can lead to misinterpretation of names).
- Limit the use of acronyms (some acronyms have more than one meaning e.g. "ASP").
- Make the name readable.
- Avoid using spaces in names even if the system allows it.
- Ensure the name is unique and does not exist as a reserved keyword.

9.2 Tables

- Names should use PascalCase (AuditTransaction).
- Use singular or a collective name or, less ideally, a plural form. For example (in order of preference) staff and employees.
- Do not prefix with tbl or any other such descriptive prefix or Hungarian notation.
- Never give a table the same name as one of its columns and vice versa.

9.3 Columns

- Always use the singular name.
- Where possible avoid simply using id as the primary identifier for the table.
- Do not add a column with the same name as its table and vice versa.
- All column names should use PascalCase to distinguish them from SQL keywords (camelCase).
- Don't use prefixes.
- Field names should contain only letters and numbers. No special characters, underscores or spaces should be used.

9.4 Views

- While it is pointless to prefix tables, it can be helpful for views. Prefix your views with "vw", is a helpful reminder that you're dealing with a view, and not a table.
- Give a meaningful name to the view. For example, joining the "Customer" and "StateAndProvince" table to create a view of Customers and their respective geographical data should be given a name like "vwCustomerStateAndProvince".
- Names should use PascalCase after the prefix.

9.5 Stored Procedures

- The name must contain and start with a verb (Get, Create, Save, Insert, Update, Delete, Validate, etc.)
- Do not prefix with sp_ or any other such descriptive prefix or Hungarian notation.
- Names should use camelCase (getCustomerInformation).
- SQL reserved words should be written in UPPER Case (SELECT, FROM, WHERE, etc)

9.6 Functions

- Prefix the name with "fn" as a helpful reminder that you're dealing with a function, and not a stored procedure.
- Names should use camelCase (fnGetOpenDate).
- Functions should be named as a verb, because they will always return a value.
- SQL reserved words should be written in UPPER Case (SELECT, FROM, WHERE, etc)

9.7 Variables

- Variable names should be meaningful and natural.
- Variable names should describe its purpose and not exceed 30 characters in length.
- All variables must begin with the "@" symbol. Do NOT use "@@" to prefix a variable as this signifies a SQL Server system global variable and will affect performance.
- All variables should be written in camelCase, e.g. "@firstName" or "@city" or "@siteId".
- Variable names should contain only letters and numbers. No special characters or spaces should be

used.

9.8 DB Coding Conventions

- SQL statements should be arranged in an easy to read manner, refer to the following example:

```
SELECT
    dui.DealUnitInvoiceID,
    dui.UnitInventoryID,
    ui.UnitID,
    ui.StockNumber [Stock Number],
    ut.UnitType AS [Unit Type],
    COALESCE(mk.Description, '') Make,
    COALESCE(ml.Description, '') Model,
    DATEPART(YEAR,u.ProductionYear) [Year],
    ut.UnitTypeID,
    mt.Description AS MeterType,
    ui.MeterReading,
    ui.ECMReading,
    '$' + LTRIM(CONVERT(nvarchar(18),CONVERT(decimal(18, 2),dui.Price))) Price,
    '$' + LTRIM(CONVERT(nvarchar(18),CONVERT(decimal(18, 2),dui.Cost))) Cost,
    dui.IsTradeIn,
    COALESCE(u.Vin, '') Vin,
    COALESCE(u.SerialNumber, '') SerialNumber,
    ui.AvailabilityStatusID,
    ui.SellingStatusID,
    ui.IsNew,
    ui.UnitPurchaseOrderID,
    ui.BaseCost,
    dui.DealPacketInvoiceID
FROM    dbo.DealUnitInvoice dui
INNER JOIN  dbo.UnitInventory ui
    ON dui.UnitInventoryID = ui.UnitInventoryID
INNER JOIN  dbo.Unit u
    ON ui.UnitID = u.UnitID
LEFT JOIN  dbo.MeterType mt
    ON u.MeterTypeID = mt.MeterTypeID
LEFT JOIN  dbo.UnitType ut
    ON ui.UnitTypeID = ut.UnitTypeID
    AND ut.InActive = 0
LEFT JOIN  dbo.Make mk
    ON u.MakeID = mk.MakeID
    AND mk.InActive = 0
LEFT JOIN  dbo.Model ml
    ON u.ModelID = ml.ModelID
    AND ml.InActive = 0
WHERE ut.ui.IsNew = 1
```

Note how the tables are aliased and joins are clearly laid out in an organized manner.



- When developing a stored procedure consider the following example to use **TRY – CATCH** and **BEGIN – END** blocks as well as the indentation:

```
CREATE PROCEDURE DoStuff
    @var1 int
AS
SET NOCOUNT ON

BEGIN TRY
    BEGIN TRAN
        DELETE FROM MyTable
        WHERE Col1 = @var1
        INSERT INTO MyOtherTable(Col1)
        SELECT @var1
    COMMIT TRANSACTION
END TRY
GO

BEGIN CATCH
    SELECT
        ERROR_NUMBER() as ErrorNumber,
        ERROR_MESSAGE() as ErrorMessage
    -- Test XACT_STATE for 1 or -1.
    -- XACT_STATE = 0 means there is no transaction and
    -- a commit or rollback operation would generate an error.

    -- Test whether the transaction is uncommittable.
    IF (XACT_STATE()) = -1
    BEGIN
        PRINT
            N'The transaction is in an uncommittable state. ' +
            'Rolling back transaction.'
        ROLLBACK TRAN
    END
    -- Test whether the transaction is active and valid.
    IF (XACT_STATE()) = 1
    BEGIN
        PRINT
            N'The transaction is committable. ' +
            'Committing transaction.'
        COMMIT TRAN
    END
END CATCH
```

- **Code Commenting:** Important code blocks within stored procedures and user defined functions should be commented. Brief functionality descriptions should be included where important or complicated processing is taking place.
- **Code Headers:** Stored procedures, triggers and user-defined functions should have a code header. The header can be created using the comment syntax above. This header should give a brief description of the functionality of the procedure as well as any special execution instructions. Also contained in this header there should be a brief definition of any parameters used. Refer to the example below. You may also include an execution example of the function or procedure in the header as well.

```
CREATE PROCEDURE [dbo].[validateConcurrency]
    @TableName varchar(255),
    @ID int,
    @LastUpdate datetime,
    @IsValid bit OUTPUT
AS
/*****
This procedure validates the concurrency of a record update by taking
the LastUpdate date passed in and checking it against the current
LastUpdate date of the record. If they do NOT match the record is not
updated because someone has updated the record out from under
the user.
-----
Parameter Definition:
-----
@TableName = Table to be validated.
@ID = Record ID of the current record to be validated.
@LastUpdate = The Last Update Date passed by app to compare with
current date value for the record.
@IsValid = Returns the following back to the calling app.
1 = Record is valid. No concurrency issues.
0 = Record is NOT concurrent.
*****/
```


10 Secure Website

10.1 Authentication and Authentication

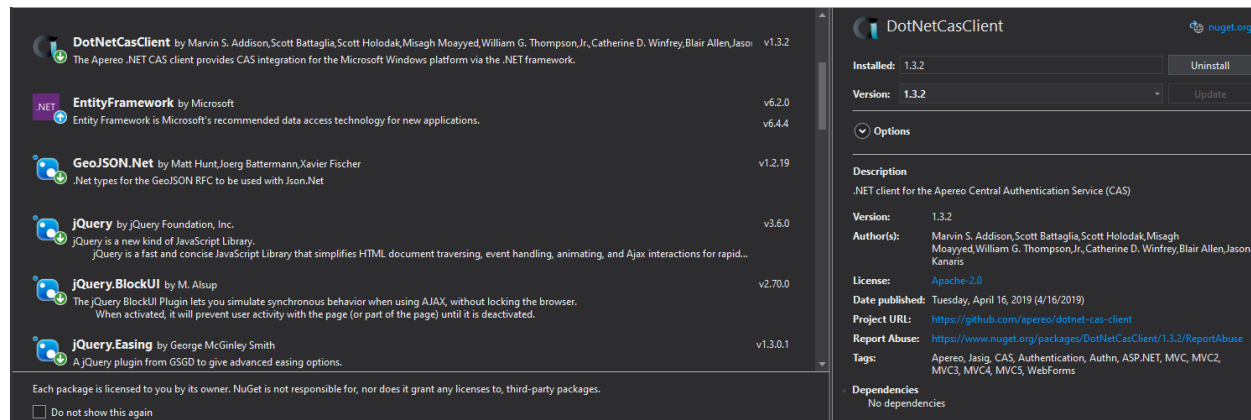
What is Authentication?

Authentication is a process to ensure and confirms a user's identity and whether the user is registered or not to access particular data or web pages. In other words, we can say that it is a process to validate someone against some data source.

What is Authorization?

Authorization is a security mechanism which is used to determine whether the user has access to a particular resource or not. The main point that you need to remember is, authentication happens first, then only authorization.

10.1.1 Install DotNetCasClient using NuGet



10.1.2 Forms Authentication

Add the following line to implement CAS Form Authentication

```

<system.web>
  <authentication mode="Forms">
    <forms name="TP_Cookie_" loginUrl="https://cas5.elpaso.ttuhsoc.edu/cas/login" timeout="90"
      defaultUrl="/~/Default.aspx" cookieless="UseCookies" path="/" enableCrossAppRedirects="true" />
  </authentication>
</system.web>

<configuration>

```



```
<casClientConfig casServerLoginUrl="https://cas5.elpaso.ttuhsoc.edu/cas/login"  
casServerUrlPrefix="https://cas5.elpaso.ttuhsoc.edu/cas/" serverName="https://localhost:44305/"  
notAuthorizedUrl="~/shared/AccessDenied.cshtml" cookiesRequiredUrl="~/CookiesRequired.aspx"  
redirectAfterValidation="true"  
renew="false" singleSignOut="true" ticketValidatorName="Cas20"  
serviceTicketManager="CacheServiceTicketManager" />  
</configuration>
```

10.1.3 Using Authorize and AllowAnonymous Attributes

The **Authorize** Attribute is the built-in attribute provided by MVC which is basically used to authenticate a user. The action methods which any anonymous user won't have to is decorated with Authorize Attribute.

The **AllowAnonymous** attribute helps secure website or Controller while providing convenient means of allowing anonymous user access to certain controller actions just like login and register actions.

- Create an Authorization Controller

[Authorize]

```
public class AuthorizationController : Controller  
{  
    private readonly FormsAuthenticationTicket _ticket =  
        DotNetCasClient.CasAuthentication.GetFormsAuthenticationTicket();  
  
    public ActionResult WelcomeScreen()  
    {  
        return View();  
    }  
}
```

[AllowAnonymous]

```
public ActionResult Login()  
{  
    try  
    {  
        DisableCacheHistory();  
  
        if (_ticket == null || !User.Identity.IsAuthenticated)  
        {  
            return WelcomeScreen();  
        }  
        return RedirectToAction("Home", "Home");  
    }  
    catch (Exception)  
    {  
        RemoveSession(false);  
        return WelcomeScreen();  
    }  
}  
  
public ActionResult Logout()  
{
```



```
    return WelcomeScreen();
}

private void DisableCacheHistory()
{
    Response.Cache.SetCacheability(HttpCacheability.NoCache);
    Response.Cache.SetAllowResponseInBrowserHistory(false);
    Response.CacheControl = "no-cache";
    Response.Cache.SetNoServerCaching();
}

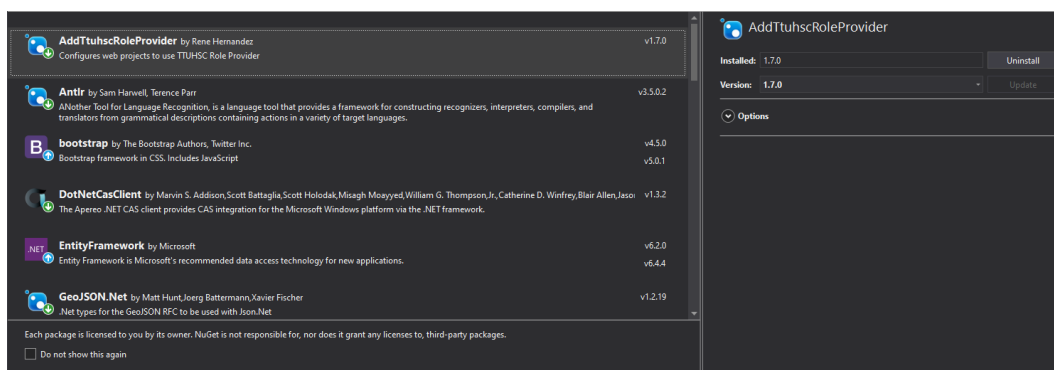
private void RemoveSession(bool signOutCas)
{
    DisableCacheHistory();
    Session.RemoveAll();
    Session.Abandon();

    if (signOutCas)
    {
        DotNetCasClient.CasAuthentication.SingleSignOut();
    }
}
}
```

10.1.4 Role base authentication

To add a role base authentication the steps are:

- Import NuGet package AddTtuhsRoleProvider



- Add the following lines on the Web.config file

```
</system.web>
<roleManager enabled="true" defaultProvider="RoleProvider">
  <providers>
    <clear />
    <add name="RoleProvider" type="TTUHSC.Web.Security.RoleProvider"
      connectionStringName="RoleProviderConnectionString" applicationName="ApplicationName" />
  </providers>
</roleManager>
</system.web>
```

- Using the EraiderAuthorize Attribute

```
[HttpGet]
[Authorize]
[EraiderAuthorize(Roles = "Admin")]
public ActionResult HomeView()
{
    try
    {
    }
    catch (Exception ex)
    {
    }
    return View(model);
}
```

10.2 Cross-Site Request Forgery (CSRF)

An attacker acts as a trusted source and sends some forged data to a site. The site processes the forged data because it believes it is coming from a trusted source.

We can use the HTML tag helper **asp-antiforgery** in an HTML attribute and set its value as true. By default, this value will be false. If we set this value as true, it will generate an anti-forgery token. Then, we need to add the **[ValidateAntiForgeryToken]** attribute to the form post action method to check whether a valid token is generated.

Example:

Setting the asp-antiforgery tag helper to true.

```
<form class="col-12" id="form" asp-antiforgery="true">
  <div class="text-center">
    <h1>Business</h1>
    <div class="separator"></div>
  </div>
</form>
```



Adding [ValidateAntiForgeryToken] Attribute to the [HttpPost] Method

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult View(model model)
{
    try
    {
    }
    catch (Exception ex)
    {
    }
    return View(model);
}
```

The anti-forgery token can also be added using the following razor line

```
<form class="col-12" id="form">
    @Html.AntiForgeryToken()
    <div class="text-center">
        <h1>Business</h1>
        <div class="separator"></div>
    </div>
</form>
```

When using JavaScript or jQuery to post using Ajax we can get the Anti-forgery token and then proceed to add it to the data attribute.

Example:

Adding the AntiForgeryToken to the data attribute

```
$.ajax({
    url: call,
    type: 'post',
    data: {'input[name="__RequestVerificationToken"]', $form).val()},
    dataType: 'json',
    contentType: 'application/x-www-form-urlencoded; charset=utf-8',
    success: function (data) {},
    error: function (err) {}
});
```

Adding [ValidateAntiForgeryToken] Attribute to the [HttpPost] Method

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<JsonResult> GetGeoLocation(string address)
{
    try
    {
        return Json(new { success = true });
    }
    catch (Exception exception)
    {
    }
}
```

```

        var innerException = exception.InnerException != null ? exception.InnerException.Message :
        "N/A";
        return Json(new { success = false, text = $"Message:{exception.Message} InnerMessage:
        {innerException}" });
    }
}

```

10.3 Cross Site Scripting (XSS)

10.3.1 Regular Expression Attribute and object model

Use of regular expressions to validate the user's form inputs. Deny malicious characters, symbols, or allow only acceptable required characters in the input field before allowing the user to proceed further.

Examples:

Regular expression attribute

```
[RegularExpression(@"^[a-zA-Z"]-\s{1,40}$", ErrorMessage = "Characters are not allowed.")]
public object FirstName;
```

```
[RegularExpression(@"^[a-zA-Z"]-\s{1,40}$")]
public object LastName;
```

Regular expression object model

```

public static void Main()
{
    string[] values = { "111-22-3333", "111-2-3333"};
    string pattern = @"^\d{3}-\d{2}-\d{4}$";

    foreach (string value in values)
    {
        if (Regex.IsMatch(value, pattern))
            Console.WriteLine("{0} is a valid SSN.", value);
        else
            Console.WriteLine("{0}: Invalid", value);
    }
}

```

10.3.2 URL Encoding

We use plain text in URL query strings, which can be used to launch XSS attacks. So, we should encode the query parameter input in the URL.

Example:

NuGet built-in library

```
string encodedValue = System.Net.WebUtility.UrlEncode("raw-string-text");  
string decodedValue = System.Net.WebUtility.UrlDecode(encodedValue);
```

10.3.3 SQL Injection

It's a dangerous attack where in unauthorized users inject malicious SQL code that then runs in your database, allowing the attackers to access confidential information stored in it.

The SQL injection attack can be prevented in the following ways:

a) Validate inputs

Defending against SQL injection by validating inputs requires the following actions:

- Validate the user inputs on both the client side and server side.
- Do not allow special characters that are involved in SQL scripts.
- Use regular expressions and data annotations to validate inputs.

b) Use stored procedures

Using stored procedures will prevent SQL injection, but still validate the input parameters passed to the stored procedures.

c) Use parameterized queries

Parameterized queries to prevent SQL injection.

Example:

```
SqlConnection sqlConnection = new SqlConnection();  
SqlCommand sqlCommand = new SqlCommand("SELECT * FROM Users WHERE id = @id", sqlConnection);  
SqlParameter sqlParameter = new SqlParameter();
```

```
sqlParameter.ParameterName = "@id";  
sqlParameter.Value = id;  
sqlCommand.Parameters.Add(sqlParameter);
```

d) Use Entity Framework or any other ORM

ORM stands for object-relational mapper, which maps SQL objects to your application class object.

Using Entity framework properly, makes the application not prone to SQL injection attacks because Entity Framework internally uses parameterized queries.

e) Use least-privileged DB access

Limiting DB user permissions for tables that have confidential data. For example, we should restrict the insert, update, and delete permissions for tables related to payments and transactions, and we should also limit permissions for tables that store a user's personal information.

If a user only works with Select queries, then we must give permission only to the Select statement, and should not provide permissions for Insert, Update, and Delete statements.

f) Store encrypted data

Confidential information like email addresses and passwords as plain text in a database should be stored in an encrypted format.

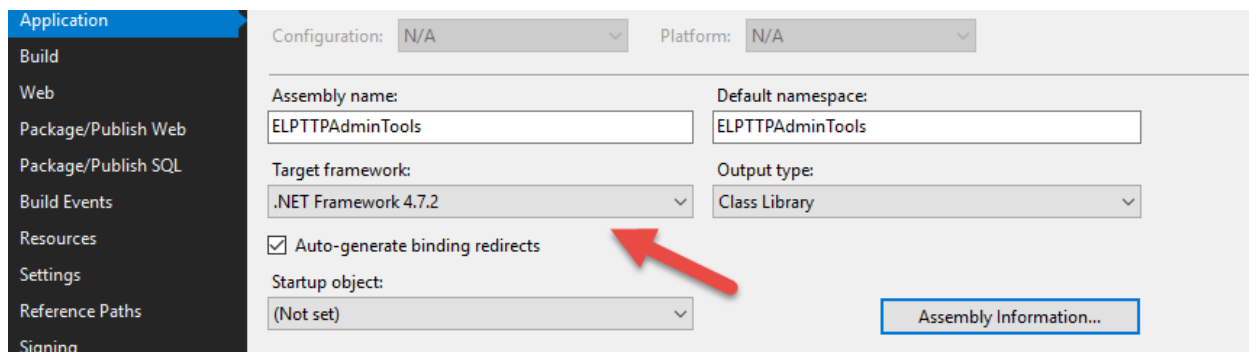
10.3.4 Secure Cookies

Securing cookies will help you avoid a hacker gets a secure cookie and hijack a session. There's a series of steps that we need to do in order to secure cookies.

The application must target at least .Net Framework version 4.7.2, the target framework can be verified by checking on the project properties.

Example:

Project Settings



Next step would be to add the following lines on the **global.asax**

Global.asax

```
protected void Application_Start()
{
    MvcHandler.DisableMvcResponseHeader = true;
}
```



```
protected void Application_PreSendRequestHeaders()
{
    if (HttpContext.Current != null)
    {
        HttpContext.Current.Response.Headers.Remove("Server");
    }
}
```

And finally add the following lines on the **web.config** file.

```
<system.web>
  <httpRuntime targetFramework="4.7.2" enableVersionHeader="false"/>
  <compilation debug="true" targetFramework="4.7.2">
  </compilation>
  <sessionState cookieSameSite="Strict" />
  <httpCookies httpOnlyCookies="true" requireSSL="true"/>
  <authentication mode="Forms">
    <forms name="appname" loginUrl="https://cas5.elpaso.ttuhscc.edu/cas/login" timeout="30"
      defaultUrl="~/Authorization/Login" cookieless="UseCookies" path="/"
      enableCrossAppRedirects="true" cookieSameSite="Strict" requireSSL="true"/>
  </authentication>
</system.web>
```

Under **system.webserver** add the following lines and modify accordingly

```
<system.webServer>
  <httpProtocol>
    <customHeaders>
      <add name="X-Frame-Options" value="SAMEORIGIN" />
      <add name="X-Xss-Protection" value="1; mode=block" />
      <add name="X-Content-Type-Options" value="nosniff" />
      <add name="Referrer-Policy" value="no-referrer" />
      <add name="X-Permitted-Cross-Domain-Policies" value="none" />
      <remove name="X-Powered-By" />
    </customHeaders>
  </httpProtocol>
</system.webServer>
```

As an additional step if you need to create a cookie on javascript the following code can be used,

```
function setCookie(name, value, expirationDays) {
    var d = new Date();
    d.setTime(d.getTime() + (expirationDays * 24 * 60 * 60 * 1000));
    var expires = d.toUTCString();
    document.cookie = name + "=" + value + ";expires=" + expires + ";path=/;secure;";
}
```

```
function setCookieHttpOnly(name, value, expirationDays) {
    var d = new Date();
    d.setTime(d.getTime() + (expirationDays * 24 * 60 * 60 * 1000));
    var expires = d.toUTCString();
    document.cookie = name + "=" + value + ";expires=" + expires + ";path=/;secure;httpOnly;";
}
```

10.4 Custom Error Page for Error Handling

Sometimes you may not properly write error-handling code in your web application. This can lead us to expose sensitive information such as database configuration info, table names, stored procedures, data structures, and programming coding structure to users.

Example:

Adding the bellow lines to the web.config file will enable custom error handling:

```
<customErrors mode="On" defaultRedirect="~/Error/GeneralError">
    <error statusCode="403" redirect="~/Error/Forbidden" />
    <error statusCode="404" redirect="~/Error/PageNotFound" />
    <error statusCode="500" redirect="~/Error/InternalError" />
</customErrors>
```

Next step would be to create a controller to handle the views:

```
public class ErrorController : Controller
{
    public ActionResult GeneralError()
    {
        return View();
    }

    public ActionResult Forbidden()
    {
        return View();
    }

    public ActionResult PageNotFound()
    {
        return View();
    }

    public ActionResult InternalError()
    {
        return View();
    }
}
```

And finally create your own custom error html forms:

