.NET CODING STANDARDS AND BEST PRACTICES

Revision History

Version	Date	Reason for Change
1.0	01- Aug -2016	Baseline and Approved By Management

TatvaSoft
sculpting thoughts...

Table of Contents

1. Naming Conventions and Standards

1.1. Camel Case

"Camel case" is the practice of writing identifiers in which no underscores are used to separate words, the first letter is lowercase, and the first letter of each subsequent word is capitalized. Examples: fileName, voterAddress. Use camel case to name private and protected fields, parameters, and local variables.

1.2. Pascal Case

In "Pascal case", no underscores are used to separate words, and the first letter of each word is capitalized. Examples: GetFileName, MainForm. Use Pascal case for public fields and properties, class names including enumerated types and structures), and namespaces.

1.3. Naming Constants

All constants should be named in uppercase with underscores to separate words. Examples: ROW_COUNT, COLUMN_NAME, URL

1.4. Other Guideline

1. Use Pascal casing for Class names

```
public class HelloWorld
 {

 }
```

2. Use Pascal casing for Method names

```
public void PrintMessage(string name)
 {

 }
```

3. Use Camel casing for variables and method parameters

```
int totalNumber;
public void PrintMessage(string firstName, string lastName)
{

}
```

4. Use the prefix "I" with Camel Casing for interfaces (Example: **IEntity**)

5. Use Meaningful, descriptive words to name variables. Do not use abbreviations.

   Good:

```
string address;
int salray;
string firstName;
```

   Not Good:

```
string fn;
string strAddress;
int sal;
```

6. Do not use single character variable names like i, n, s etc.

One exception in this case would be variables used for iterations in loops:

```
for ( int i = 0; i < count; i++ )
{

}
```

If the variable is used only as a counter for iteration and is not used anywhere else in the loop, many people still like to use a single char variable (i) instead of inventing a different suitable name.

7. Do not use underscores (_) for local variable names.

8. All member variables must be prefixed with underscore (_) so that they can be identified from other local variables.

9. Do not use variable names that resemble keywords.

10. Prefix boolean variables, properties and methods with "is".

    Ex: private bool _isFinished;   ("_" for member)
    private bool isFinished; (local variable)

11. Namespace names should follow the standard pattern

    <project name>.<module name>.<folder>

12. Use appropriate prefix for the UI elements so that you can identify them from the rest.

A brief list is given below. Since .NET has given several controls, you may have to arrive at a complete list of standard prefixes for each of the controls (including third party controls) you are using.

An exception to this is control names. We use Hungarian Notation for these:

Label –	lbl
CheckBox –	chk
LinkLabel –	lnk
CheckedListBox –	chklListbox
ComboBox –	cbo
ProgressBar –	pgb
Control –	ctrl
RadioButton –	rad
Menus –	mnu
RichTextBox –	rtb
Panel –	pnl

Splitter –	spl
TextBox –	txt
PictureBox –	pic
StatusBar –	sba
Datagrid –	dgr
DatagridColumn –	dgrc
ToolBar –	tba
Dialog Controls –	dlg
ToolTip –	tip
Form –	frm
GroupBox –	gbx
DataTable –	dt
DataSet –	ds
DataRow -	row
Image –	img

13. File name should match with class name.

For example, for the class HelloWorld, the file name should be HelloWorld.cs (or, HelloWorld.vb)

2. Indentation and Spacing

1. Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.

2. Comments should be in the same level as the code (use the same level of indentation).

Good:

```
// This declares and assigns these variables all in one step.
short levelNumber = 3;
int score = 0;
```

```
// This declares a variable, and assigns it at a later point in time.
long aBigNumber;
aBigNumber = -17;

// Here is a byte, which contains any value between 0 and 255
byte aSingleByte = 55;
```

Not Good:

```
// This declares and assigns these variables all in one step.
short levelNumber = 3;
int score = 0;

// This declares a variable, and
//assigns it at a later point in time.
long aBigNumber;
aBigNumber = -17;

// Here is a byte, which contains any value between 0 and 255
byte aSingleByte = 55;
```

3. Curly braces ({}) should be in the same level as the code outside the braces.

Good:

```
private bool SayHello(string name)
{
  DateTime currentTime = DateTime.Now;
  if (string.IsNullOrEmpty(name))
    {
          string message = string.Format("Hello {0}", name);
          Console.WriteLine(message);
          Console.WriteLine(currentTime.ToShortDateString());
   }
  else
   {
          Console.WriteLine("Name is empty");
   }
  return true;
}
```

Bad:

```
private bool SayHello(string name){
   DateTime currentTime = DateTime.Now;

if (string.IsNullOrEmpty(name)){
   string message = string.Format("Hello {0}", name);
  Console.WriteLine(message);
 Console.WriteLine(currentTime.ToShortDateString());}
}
```

4. Use one blank line to separate logical groups of code.

Good:

```
private bool SayHello(string name)
{
    DateTime currentTime = DateTime.Now;

    if (string.IsNullOrEmpty(name))
    {
        string message = string.Format("Hello {0}", name);
        Console.WriteLine(message);
        Console.WriteLine(currentTime.ToShortDateString());

    }
     return true;

}
```

Not Good:

```
private bool SayHello(string name)
{
   DateTime currentTime = DateTime.Now;
   if (string.IsNullOrEmpty(name))
      {
          string message = string.Format("Hello {0}", name);
          Console.WriteLine(message);
          Console.WriteLine(currentTime.ToShortDateString());
      }
   return true;
}
```

5. There should be one and only one single blank line between each method inside the class.

6. The curly braces should be on a separate line and not in the same line as if, for etc.

   Good:

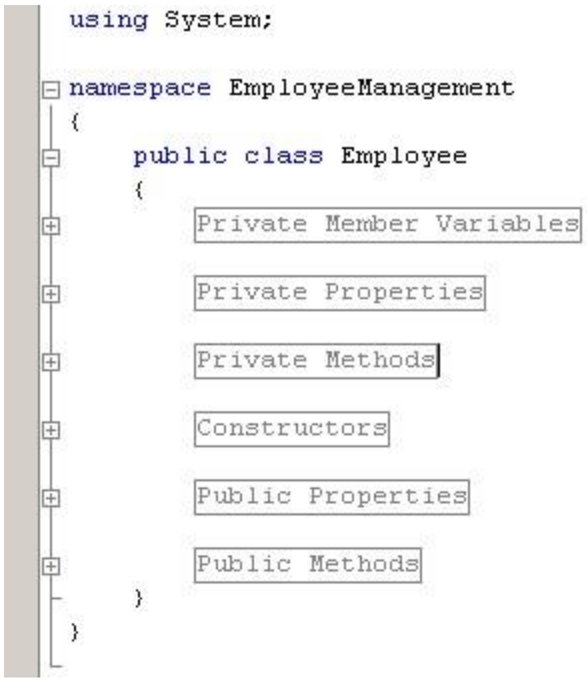
```
if (isResult == true)
{
      //Do something
}
```

7. Use a single space before and after each operator and brackets.

   Good:

```
if (isResult == true)
{
    for (int i = 0; i < 10; i++)
    {
       //Do something
    }
}
Bad:
if (isResult==true)
{
   for (int i=0;i<10;i++)
   {
      //Do Something
   }
 }
```

8. Use #region to group related pieces of code together. If you use proper grouping using
#region, the page should like this when all definitions are collapsed.
Example:



```
using System;

namespace EmployeeManagement
{
    public class Employee
    {
        Private Member Variables

        Private Properties

        Private Methods

        Constructors

        Public Properties

        Public Methods
    }
}
```

9. Keep private member variables, properties and methods in the top of the file and public members in the bottom.

3. Good Programming practices

1. Avoid writing very long methods / functions. A method should typically have 1~25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.

2. Method name should tell what it does. Do not use miss-leading names. If the method name is obvious, there is no need of documentation explaining what the method does.

   Good:
   ```
   public void SavePhoneNumber(string phoneNumber)
   {
         //Logic to save phone number
   }
   ```

Not Good:

```
public void SaveDetail(string phoneNumber)
{
    //Logic to save phone number
}
```

3. Always watch for unexpected values. For example, if you are using a parameter with 2 possible values, never assume that if one is not matching then the only possibility is the other value.

Good:

```
if (memberType == MemberType.Registered)
{
   //Do something
}
else if(memberType==MemberType.Guest)
{
     // Do something if memeber is guest
}     else
{
  //Unexpected Member type throw an exception;
  throw new Exception(string.Format("Unexpected Value :
                                        {0}",memberType.ToString()));
    //If we introduce a new type in future, can easily find problem here
}
```

Not Good:

```
if (memberType == MemberType.Registered)
{
  //Do something
}     else
{
  //Do something if guest is here
   //It will create a problem when we introduce a new type
 }
```

4. Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code.

   However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.

5. Convert strings to lowercase or upper case before comparing (Applicable only when comparison is not case sensitive). This will ensure the string will match even if the string being compared has a different case.

```
  string name = "John";
  if (name.ToLower() == "john")
      {
          //Do something
      }
```

   Or use string.Compare function in built

```
string.Compare(name,"john",false);
```

   Here first and second argument is for comparing string third option is for case ignorance if you pass false then it will ignore case while comparing string and if you put true it will compare string.

6. Use String.Empty instead of “”

   Good:

```
if (name == string.Empty)
{
   //Do something
}
```

   Or use string.IsNullOrEmpty function

```
if (string.IsNullOrEmpty(name))
{
   //Do something
}
```

Not Good:

```
if (name=="")
{
   //Do something
}
```

7. Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.

8. Use Enum whenever required do not use number or string hardcoded.

Good:

```
private enum MailType
{
    HTML,
    PlainText,
    Attachment
}
private void SendEmail(string message, MailType mailType)
{
    switch (mailType)
        {
         case MailType.HTML:
         //Do something
         break;

         case MailType.Attachment:
         //Do something
         break;

         case MailType.PlainText:
         //Do something
         break;

         default:
         //Do something
         break;

       }
 }
```

Not Good:

```
private void SendEmail(string message, string mailType)
{
       switch (mailType)
       {
              case "HTML":
              //Do something
              break;

              case "Attachment":
              //Do something
              break;

              case "PlainText":
              //Do something
              break;

              default:
              //Do
              something
              break;

      }
}
```

9. Do not make the member variables public or protected. Keep them private and expose public/protected Properties.

10. Do not programmatically click a button to execute the same action you have written in the button click event. Rather, call the same method which is called by the button click event handler.

11. Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.

12. Never assume that your code will run from drive "C:". You may never know, some users may run it from network or from a "Z:".

13. Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."

14. Every project we should have to generalized error track system (in global.asax), from that we can get appropriate error message and we can store in database as error log for future reference and we can sent mail to site administrator also that can help you trouble shoot a problem. We don't have to use try- catch exception handling for each and every functions, but for some specific case like file uploading and sending mail methods we will have to use try- catch exception handling.

15. Do not have more than one class in a single file.

16. Avoid having very large files. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Split them logically into two or more classes.

17. Avoid public methods and properties, unless they really need to be accessed from outside the Class. Use “internal” if they are accessed only within the same assembly.

18. If you have a method returning a collection, return an empty collection instead of null, if you have no data to return. For example, if you have a method returning a List, always return a valid List. If you have no items to return, then return a valid list with 0 items. This will make it easy for the calling application to just check for the “count” rather than doing an additional check for “null”.

19. Use the Assembly Info file to fill version number and description.

20. If you are opening database connections, sockets, file stream etc., always close them in the finally block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the finally block.

21. Use StringBuilder class instead of String when you have to manipulate string objects in a loop. The String object works in weird way in .NET. Each time you append a string, it is actually discarding the old string object and recreating a new object, which is a relatively expensive operations.

Consider the following example:

```
public string ComposeMessage(string[] lines)
{
     string message = String.Empty;

     for (int i = 0; i < lines.Length; i++)
     {
            message += lines[i];
        }
        return message;
}
```

In the above example, it may look like we are just appending to the string object 'message'. But what is happening in reality is, the string object is discarded in each iteration and recreated and appending the line to it.

If your loop has several iterations, then it is a good idea to use StringBuilder class instead of String object.

See the example where the String object is replaced with StringBuilder.

```
public string ComposeMessage(string[] lines)
{
     StringBuilder message = new StringBuilder();

     for (int i = 0; i < lines.Length; i++)
     {
         message.Append(lines[i]);
     }
     return message.ToString();
   }
```

4. Architecture

- Always use multi-layer (N-Tier) architecture.

- Never access database from the UI pages. Always have a data layer class which performs all the database related tasks. This will help you support or migrate to another database back end easily.

- Separate your application into multiple assemblies. Group all independent utility classes into a separate class library. All your database related files can be in another class library.

5. ASP.NET

- Do not use session variables throughout the code. Use session variables only within the classes and expose methods to access the value stored in the session variables. A class can access the session using System.Web.HttpCOntext.Current.Session

- Do not store large objects in session. Storing large objects in session may consume lot of server memory depending on the number of users. You can use Cache object for storing large objects which you need frequently it is much efficient than this.

- Always use style sheet to control the look and feel of the pages. Never specify font name and font size in any of the pages. Use appropriate style class. This will help you to change the UI of your application easily in future. Also, if you like to support customizing the UI for each customer, it is just a matter of developing another style sheet for them

6. Comments

Good and meaningful comments make code more maintainable. However,

1. Do not write comments for every line of code and every variable declared.

2. Use **//** or **///** for comments. Avoid using **/* … */**

3. Write comments wherever required. But good readable code will require very less comments. If all variables and method names are meaningful, that would make the code very readable and will not need many comments.

4. Do not write comments if the code is easily understandable without comment. The drawback of having lot of comments is, if you change the code and forget to change the comment, it will lead to more confusion.

5. Fewer lines of comments will make the code more elegant. But if the code is not clean/readable and there are less comments, that is worse.

6. If you have to use some complex or weird logic for any reason, document it very well with sufficient comments.

7. If you initialize a numeric variable to a special number other than 0, -1 etc, document the reason for choosing that value.

8. Perform spelling check on comments and also make sure proper grammar and punctuation is used.

7. Exception Handling

1. Never do a 'catch exception and do nothing'. If you hide an exception, you will never know if the exception happened or not. Lot of developers uses this handy method to ignore no significant errors. You should always try to avoid exceptions by checking all the error conditions programmatically. In any case, catching an exception and doing nothing is not allowed. In the worst case, you should log the exception and proceed.

2. In case of exceptions, give a friendly message to the user, but log the actual error with all possible details about the error, including the time it occurred, method and class name etc.

3. Always catch only the specific exception, not generic exception.

   Good:

```
void ReadFromFile(string fileName)
{
try
    {
       // read from file.
    }
    catch (FileNotFoundException fileNotFoundException)
    {
       // log error.
       // re-throw exception depending on your case.
      throw;
    }
}
```

Not Good:

```
void ReadFromFile(string fileName)
{
try
    {
        // read from file.
    }
    catch (Exception exception)
    {
        // log error.
    }
}
```

Here we don't know what type of exception is there whether it's a file not found exception or we are having problem with opening file for reading.

4. When you re throw an exception, use the throw statement without specifying the original exception. This way, the original call stack is preserved.

Good:

```
 try
{

}
Catch (Exception exception)
{
    //Write code for handling exception
    throw;
}
```

Not Good:

```
 try
{

}
catch (Exception exception)
{
    //Write code for handling exception
    throw exception;
}
```

5. Do not write try-catch in all your methods. Use it only if there is a possibility that a specific exception may occur and it cannot be prevented by any other means. For example, if you want to insert a record if it does not already exists in database, you should try to select record using the key. Some developers try to insert a record without checking if it already exists. If an exception occurs, they will assume that the record already exists. **This is strictly not allowed. You should always explicitly check for errors rather than waiting for exceptions to occur.**

6. Write your own custom exception classes if required in your application. Do not derive your custom exceptions from the base class SystemException. Instead, inherit from ApplicationException.

7. If you are using multiple catch then catch order should be specific to general like following.

```
 try
{
 }
catch (FileNotFoundException fileNotFoundException)
{
      //Write code for handling exception
      throw fileNotFoundException;
}
catch (Exception exception)
{
    //Write code for handling exception
    throw exception;
}
```

8. Do's for C#

Take care of following.

- Implement IDisposable Interface whenever required.
- Use Switch statement instead of If-else whenever possible.
- Name type name using Noun phases or adjective phrases.
- Use Constant and readonly whenever required.
- Finish every if-elseif statement with else part.
- Always write default in switch statement.
- Avoid nested loops as much as possible.
- Use Inexplicit comparison i.e. use if(IsValid) instead of if(IsValid==true).
- Declare and initialize variable as late as possible.
- Use using statement as much as possible.
- Always use resource files for the text of label, button and other controls.

9. Don'ts for C#:

- Avoid using static classes and static variables.
- Don't hide inherited member with new keyword.
- Don't right methods with hundreds of line. As thumb of rule if methods contains more than 25 lines of code then it should be sub divided into another methods.
- Do not pass parameters of methods as null.
- Properties, Methods and Argument representing string should never be NULL.
- Do not put multiple classes in single file.

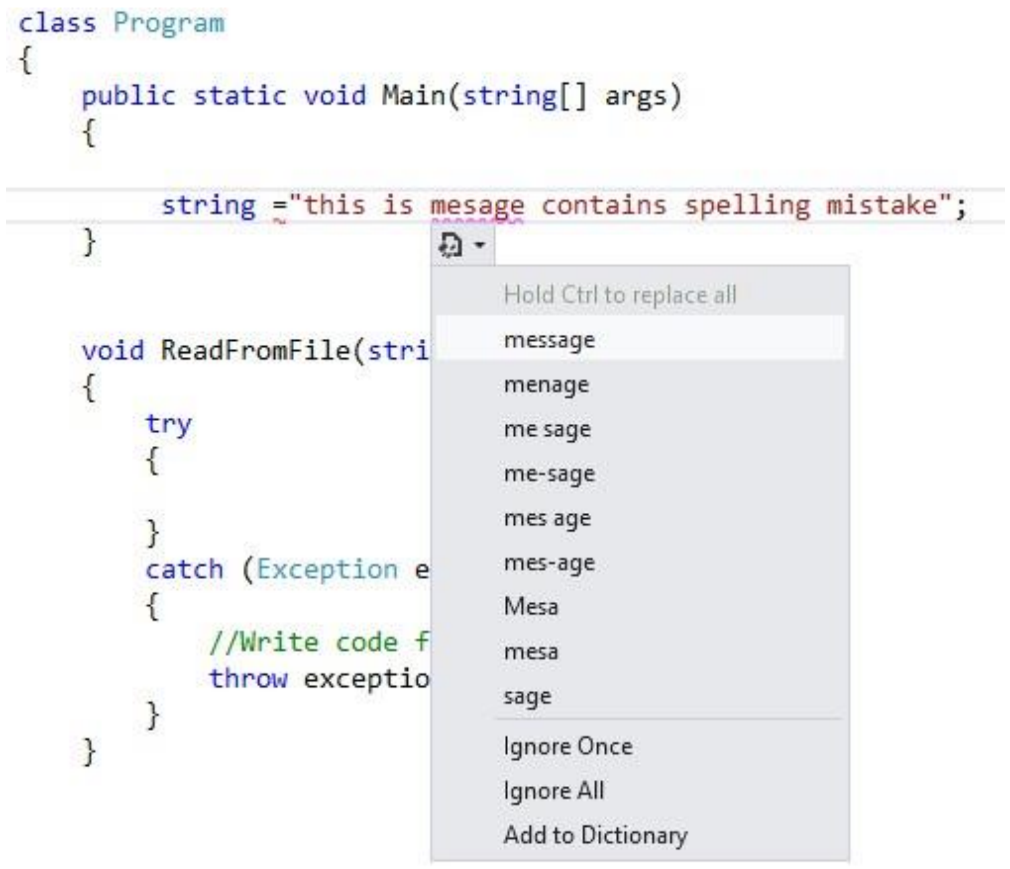
- Files/Class should not have 1000 lines of code it's better to have that class divide in to multiple classes.
- No line should exceeds 200 hundred characters.
- Do not make spelling mistakes in code its leaves a bad impression.

10. Spell checking in code files, html files

- There is a very good spell check available for Visual Studio which take care of spelling mistake made by developers. You can download that from following link.

  http://visualstudiogallery.msdn.microsoft.com/a23de100-31a1-405c-b4b7-d6be40c3dfff

- It helps correcting spelling mistakes from the code and if there is mistake it will highlight as underline.



```
class Program
{
    public static void Main(string[] args)
    {

        string ="this is mesage contains spelling mistake";
    }


    void ReadFromFile(stri
    {
        try
        {

        }
        catch (Exception e
        {
            //Write code f
            throw exceptio
        }
    }
```

11. Code quality

Based on requirement and with suggestion of customer developer has to use below tools to maintain the code quality.

11.1. Code Analyses

The developer should use Code Analyses to check their code. The Microsoft Managed Recommended Rules should be used and all code except auto-generated code should comply with these rule.

11.2. Style Cop

The developer should use the latest Style Cop available from https://stylecop.codeplex.com/) and should make sure that all code complies with the supplied rule set.

Auto generated code are exempt from this rule.

11.3. Re-Sharper

Visual studio plugin which helps developers with a great deal of frequent software development and maintenance tasks, such as finding unused code, complying with naming guidelines, detecting possible runtime exceptions.

11.4. Exceptions to the rules

- In some cases it can be that the coding rules can't be followed. The developer is not allowed to disable those rules by themselves.

  A developer can suggest an exception by putting following comment:

  // QualityCheck: Reason why you want the exception.

- Avoid writing very long methods / functions. A method should typically have 1~25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.

- Method name should tell what it does. Do not use miss-leading names. If the method name is obvious, there is no need of documentation explaining what the method does.

  Example:

```
 public void SavePhoneNumber(string phoneNumber)
{
      //Logic to save phone number
 }
```

- Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code.

  However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.

- Convert strings to lowercase or upper case before comparing (Applicable only when comparison is not case sensitive). This will ensure the string will match even if the string being compared has a different case.

```
string name = "John";
if (name.ToLower() == "john")
 {
              //Do something
 }
```

  Or use string.Compare function in built

```
string.Compare(name,"john",false);
```

Here first and second argument is for comparing string third option is for case ignorance if you pass false then it will ignore case while comparing string and if you put true it will compare string.

- Use String.Empty instead of “”
- Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.

- Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.
- Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."
- Every project we should have to generalized error track system (in global.asax), from that we can get appropriate error message and we can store in database as error log for future reference and we can sent mail to site administrator also that can help you trouble shoot a problem. We don’t have to use try- catch exception handling for each and every functions, but for some specific case like file uploading and sending mail methods we will have to use try- catch exception handling.
- Use the Assembly Info file to fill version number and description.
- If you are opening database connections, sockets, file stream etc., always close them in the finally block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the finally block.
- Use StringBuilder class instead of String when you have to manipulate string objects in a loop. The String object works in weird way in .NET. Each time you append a string, it is actually discarding the old string object and recreating a new object, which is a relatively expensive operations.

12. Best practices

- MVC pattern
  http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

- Bootstrap
  http://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)

- AngularJS
  http://en.wikipedia.org/wiki/AngularJS

- Entity Framework
  https://msdn.microsoft.com/en-us/data/jj591621.aspx