Best Practices while implementing Projects by

Srikanth RallabandiTable of Contents

[1. Introduction 3](#_Toc335669056)

[2. Summary 3](#_Toc335669057)

[3. Definitions 3](#_Toc335669058)

[4. Application Coding Standards 4](#_Toc335669059)

[Naming Conventions 4](#_Toc335669060)

[General Naming Guidelines 4](#_Toc335669061)

[Namespaces 4](#_Toc335669062)

[Classes and Structs 4](#_Toc335669063)

[Properties 5](#_Toc335669064)

[Methods 5](#_Toc335669065)

[Parameters 5](#_Toc335669066)

[Local variables 5](#_Toc335669067)

[Fields (class variables) 5](#_Toc335669068)

[Enums 5](#_Toc335669069)

[Interfaces 6](#_Toc335669070)

[Events, Delegates, and Callbacks 6](#_Toc335669071)

[Attributes 7](#_Toc335669072)

[Assemblies 7](#_Toc335669073)

[Source Files 7](#_Toc335669074)

[Region Usage 7](#_Toc335669075)

[Code Commenting 7](#_Toc335669076)

[General 7](#_Toc335669077)

[Function Headers 8](#_Toc335669078)

[General Programming Guidelines 8](#_Toc335669079)

[Database Coding Standards 10](#_Toc335669080)

[General Database Programming Guidelines 11](#_Toc335669081)

# Introduction

The purpose of this document is to define standard coding style to be used by Microsoft project teams for development of software projects using Microsoft .NET technologies.

# Summary

These guidelines are intended to help engineers understand the trade-offs between different solutions. There might be situations where good design requires that you overstep these design guidelines. Such cases should be rare and vetted out with the Lead Architect.  If such cases are introduced, it is important that you provide a solid justification for your decision. The sections below provide naming and usage guidelines for types in the .NET Framework as well as guidelines for implementing common design patterns.

This document also serves as the guidelines for Database development for Procedures, Functions, Views and Tables.

Source Control Standards will be found at the bottom of this document

# Definitions

1. Few examples for **Pascal casing** and **camel casing**:

|  |  |
| --- | --- |
| **Pascal Case** | **Camel Case** |
| NewImage | newImage |
| UIEntry | uiEntry |
| PmsMR | pmsMR |

1. Hungarian notation: Hungarian notation is a naming convention in which the name of an object indicates its type or intended use.  Examples:
   1. intCount: Prefixed with int to indicate that this variable is of type integer.
   2. txtEmployeeID: Prefixed with txt to indicate that this is a TextBox control.

**Hungarian notation for some commonly used UI controls:**

|  |  |
| --- | --- |
| **Control Name** | **Prefix** |
| Button | btn |
| Check box | chk |
| DataGrid | grd |
| Select control | slt |
| HyperLink | lnk |
| Image | Img |
| Label | Lbl |
| Radio button | Rad |
| Table | Tbl |
| Text box | Txt |
| Any user control | Uc |

# Application Coding Standards

The following sections cover most of the important elements to consider when developing an application.

## Naming Conventions

A consistent naming pattern is one of the most important elements of predictability. Widespread use and understanding of these naming guidelines should eliminate many of the most common user questions.

### General Naming Guidelines

1. Do not use Hungarian notation or add any other type identification to identifiers.
2. Do not prefix member fields
3. Do not use casing to differentiate identifiers.  For example, do not define employeeid and employeeID as two different variables in the same scope.
4. Use abbreviations with care.  For example, use numberinstead ofnum.
5. Do not use an underscore in identifiers
   1. Exception to this rule:  Prefixing the private members of classes and structs.
6. Name an identifier according to its meaning and not its type.

### Namespaces

1. When creating a namespace use the following pattern:

<company>.<technology>.<top-level component>.<bottom-level component>

*e.g.,Srikanth.Framework*

*Srikanth.Framework.WinSvc*

*Srikanth.Enterprise.ApplicationBlocks*

1. If the namespace is product specific, use the product name after the company name (example: Srikanth.HRMS) or if  it is generic which can be used in any project within the enterprise use the word enterprise after the company name

(Example: Srikanth.Enterprise)

1. A nested namespace should have a dependency on the types in the containing namespace. For example, *Srikanth*.Enterprise.Tools.UI depends on *Srikanth*.Enterprise.Tools, but *Srikanth*.Enterprise.Tools does not depend on the nested name.
2. Use a noun or noun phrase to name a namespace.
3. Use Pascal case
4. Always keep your assembly name, project name, and namespaces in Sync.
5. Allow subfolder names to act as an extension of the namespace hierarchy
6. To resolve the conflict in namespace try to modify the object model or namespace or alias the less frequently referenced namespace with a temporary name (especially with 3rd party assembly)

### Classes and Structs

1. Use a noun or noun phrase to name a class or struct.

 C#

public class SurveyQuestion

1. Use Pascal case
2. Do not use a type prefix in the name.  Use FileStream rather than CFileStream.
3. When appropriate, use a compound word to name a derived class.

 C#

public class ApplicationException (which is a kind of Exception)

### Properties

1. Use a noun or noun phrase to name a property
2. Use Pascal case

### Methods

1. Name methods using verb or verb-object pair, such as ShowDialog()
2. Use Pascal case

 GetEmployee()

DeletePerson()

Invoke()

### Parameters

1. Use descriptive names.  Make sure the name describes the “what” rather than the “how”
2. Use camel case
3. Do not use prefix and do not use underscores

 C#

public EmployeeGetEmployee(int employeeID)

### Local variables

1. Use descriptive names.  Make sure the name describes the “what” rather than the “how”
2. Use camel case
3. Do not use prefix and do not use underscores

 C#

int employeeID

### Fields (class variables)

1. Prefix private members with “\_”

 C#

private int \_employeeID;

1. Use descriptive names.  Make sure the name describes the “what” rather than the “how”
2. Use camel case for private fields, but use all capitals for constand read-only static fields

 C#

const int KEYSIZE = 256;

private const int \_CODE = 123;

1. Use this/Me keyword on class variables to avoid conflicts.

 C#

this.employeeID = 10001;

1. Use properties for public and protected fields

### Enums

1. Use descriptive names.  Make sure the name describes the “what” rather than the “how”
2. Use Pascal case for enum types and enum values.
3. Do not add an Enum suffix to an enumeration type.
4. Use singular names for enumeration types.

 C#

public enumColor{Red = 1, Green = 2, Blue = 3 }

1. Use a plural name for enumerations representing bit fields.  The following code snippet is a good example of an enumeration that allows combining multiple options.

 C#

publicenumSearchOptions

{

        CaseInsensitive = 0x01,

        WholeWordOnly = 0x02,

        AllDocuments = 0x04,

        AllowWildcards = 0x08

}

### Interfaces

1. Prefix interfaces with the letter I.  UseIEnumeratorinstead of  Enumerator
2. Use similar names for the default implementation of an interface.  If you provide a default implementation for a particular interface, use a similar name for the implementing class.  Notice that this only applies to classes that only implement that interface.  For example, a class implementing theIComponentinterface could be calledComponent or DefaultComponent.

### Events, Delegates, and Callbacks

1. Use a verb (gerund) for naming an event.  Good examples of eventsare Closing, Minimizing, and Arriving.  For example, the declaration for the Closing event may look like this:

 C#

public event ClosingEventHandler Closing;

1. Prefix an event handler with On.  It is good practice to prefix the method that is registered as an event handler with On.  For example, a method that handles theClosingevent should be namedOnClosing().
2. Add EventHandler to delegates related to events.  Delegates that are used to define an event handler for an event must be suffixedwith EventHandler.  For example, the following declaration is correct for a Close event.

 C#

public delegate CloseEventHandler(object sender, EventArgs arguments)

1. Add Callback to delegates related to callback methods.  Delegates that are used to pass a reference to a callback method (so**not** an event) must be suffixed with Callback. For example:

 C#

public delegate AsyncIOFinishedCallback(IpcClient client, string message);

1. Do not add a Callback or similar suffix to callback methods.  You cannot make assumptions on whether methods will be called through a delegate or not. An end-user may decide to use Asynchronous Delegate Invocation to execute the method.
2. Do not add an Event suffix (or any other type-related suffix) to the name of an event.  Use an –ing and –ed form to express pre-events and post-events.  Do not use a pattern like BeginXxx and EndXxx or a pattern like BeforeValidation and AfterValidation. Instead, use aValidat*ing*andValidat*ed*pattern.

### Attributes

1. Suffix names of custom attributes with Attribute. Example: AuthorAttribute

### Assemblies

1. Name DLL assemblies after their containing namespace.  To allow storing assemblies in the GAC, their names must be unique.  Therefore, use the namespace name as a prefix of the name of the assembly. As an example, consider a group of classes organized under the namespace Srikanth.Framework.ApplicationBlocks.  In that case, the assembly generated from those classes will be called Srikanth.Framework. ApplicationBlocks.dll. If multiple assemblies are built from the same namespace, it is allowed to append a unique postfix to the namespace name.

### Source Files

1. Use Pascal casing for naming source files.
2. Do not use the underscore character and do not use casing to differentiate names of files.

### Region Usage

1. All classes should have the following regions when code exists:
   1. Variables – use for enums and constants as well
   2. Methods – add sub-regions for Private, Public, Abstract as needed
   3. Events/Delegates
   4. Properties
2. Only create a region with code, e.g. no empty regions.

### Code Commenting

Most engineers, when looking back at code they wrote just a few months ago, have a very tough time understanding what, exactly, they were doing. Part of the reason old code is tough to read is because it usually lacks sufficient commenting. The question often arises, "When should I comment?" Well, there is no such thing as "over-commenting," so if you think you should comment at a specific place, do not hesitate to do so! By commenting your code thoroughly, you'll greatly increase the ease of maintainability of your applications.

Since much of the time comments become invalid and/or inaccurate as soon as changes are made, the engineer should always update comments when changing the code.

### General

As a standard good programming practice, all code should be thoroughly commented.  This is not to say that each line of code needs a comment.  Well named variables can usually be self-documenting. Take the following example of variable declarations:

  C#

int level; // Indentation level

int size; // Size of table

Using variable names that are more meaningful as in the following example helps reduce the need for some comments:

  C#

int indentationLevel;

int tableSize;

Try to organize and comment your code into segments as in the following example:

  C#

// Create the message and set the base properties

System.Messaging.Message myMsg = new System.Messaging.Message();

myMsg.Body = messageText;

myMsg.Label = "Created from MSMQUtility";

Use the TODO: comment so incomplete code shows up in the Visual Studio Task List as in the following example:

C#

// TODO: Create a helper function to...

### Function Headers

All functions should be commented using the .Net XML comments as in the following example:

C#

///<summary>

/// <para>Posts a message to a MSMQ queue... </para>

/// <para>This function should only be used to... </para>

///</summary>

///<param name="messageText">The body of the message.</param>

///<param name="queueName">The name of the destination queue.</param>

///<param name="isTransactional">The type of queue.</param>

///<param name="serverName">The name of MSMQ server.</param>

///<returns>Sucess or failure</returns>

///<exception cref="System.Exception">Throws an exception...</exception>

publicstaticbool PostMessage(string messageText, string queueName, bool   isTransactional, string serverName)

### General Programming Guidelines

* 1. When using abstract classes, offer an interface as well.
  2. Avoid events as interface members.
  3. All member variables should be declared at the top, with one line separating them from the properties or methods.
  4. Methods/functions should be limited to one page in length, or shorter.
  5. Avoid fully qualified type names. Use the using statement instead.
  6. Avoid putting ausing<namespace>statement inside a namespace.
  7. Group all framework namespaces together and put custom or third-party namespaces underneath.

C#

using System;  
using System.Collections.Generic;  
usingSrikanth.Framework;  
using Srikanth.WinSvc;

* 1. Maintain strict indentation.  Do not use nonstandard indentation such as one space.  Use tabs instead.
  2. Add a space on both sides of an operator.

C#

int DaysInWeek = 7

if(shepherd != null)

* 1. While writing a complicated expression (or condition)
     1. Use parentheses to separate each logical expression
     2. Use spacing between the operators
     3. Place logical expressions on multiple lines to increase readability
     4. Keep the line continuation operator at the end of the line (not at the beginning of next line)

C#

if (((args.Length == 1) && (employeeID > 0)) ||

          ((firstName != "Ben") && (lastName != "Jon")))

* 1. While using curly braces keep it on the second line.  (TBD)
  2. Avoid comments that explain the obvious.  Code should be self-explanatory. Good code with readable variable and method names should not require comments.
  3. Document only operational assumptions, algorithm insights, tips for other engineers, and so on.
  4. With the exception of zero and one, never hard -code a numeric value; always declare a constant instead.
  5. Use blank lines to separate different code blocks inside a method.
  6. Catch only exceptions for which you have explicit handling.
  7. In a catch statement that throws an exception, always throw the original exception (or another exception constructed from the original exception) to maintain the stack location of the original error:

C#

catch(Exception exception)

{

MessageBox.Show(exception.Message);  
        **throw**; //Same as throw exception;

}

* 1. Avoid error codes as method return values.
  2. Define custom exceptions only if those are really required by the design.
  3. When defining custom exceptions:
     1. Derive the custom exception from ApplicationException.
     2. Provide custom serialization.
  4. Minimize code in application assemblies (EXE client assemblies). Use class libraries instead to contain business logic.
  5. Avoid using the new inheritance qualifier. Use override/Overrides instead.
  6. Avoid explicit casting. Use the as operator to defensively cast to a type.

C#

Dog dog = new GermanShepherd();  
GermanShepherd shepherd = dog **as** GermanShepherd;  
if(shepherd != null)  
{...}

* 1. Always check a delegate for null before invoking it.
  2. Use String.Empty instead of ""

C#

//Avoid

string name = "";

//Correct

string name = String.Empty;

* 1. When building a long string, use StringBuilder or string.Format()
  2. Avoid providing methods on structures.
     1. Parameterized constructors are encouraged.
     2. Can overload operators.
  3. Use application blocks for all purposes where an application block exists (list once they’re complete).
  4. Do not use the base word to access base class members unless you wish to resolve a conflict with a subclasses member of the same name or when invoking a base class constructor.

C#

//Example of proper use of ’base’

public class Dog

{

public Dog(string name) {}

virtual public void Bark(int howLong) {}

}

 public class GermanShepherd : Dog

{

     public GermanShepherd(string name): **base(name)** {}  
     override public void Bark(int howLong)   
     {

**base.**Bark(howLong);

}

}

* 1. When creating a new class, the format should be consistent.  Use #Region to organize the class file into different sections (See the section *Region Usage Guidelines* for details).  Generally, classes should be laid out in this order:
     1. Header comments and Class summaries
     2. Class data fields
     3. Constructors
     4. Properties
     5. Public methods
     6. Private methods
  2. Do not alter the Visual Studio Editor defaults for Tabs (i.e. don’t change the tab/indent size or enable the keep tabs option).

### Database Coding Standards

* 1. Avoid using a select statement to create a new table (by supplying an “into table” that does not exist). Instead, build the table and then insert into it.
  2. ***For Example SELECT ID, address INTO NewTable FROM ExisitingTable***

1. When returning a variable or computed expression, always supply a friendly alias to the client.
2. Always list column names within an insert statement. Never perform inserts based on column position alone as your code could break if table structure is changed.
3. Do not use column numbers in the ORDER BY clause.
4. Do not use SELECT \* in your queries. Always write out the required column names after the SELECT statement. This technique results in reduced disk I/O and better performance.
5. Name all tables in the singular form.

***For Example: Activity not Activities***

1. Do not prefix your stored procedure names with "sp\_". Prefix them with “spPhy\_”.  The prefix sp\_ is reserved for system stored procedures that come with SQL Server.  After the initial prefix, tack on the name of the product the procedure supports like “spPhy\_PQRS” or “spPhy\_Transition” and then the procedure name.  This helps us better identify the procedures.  For functions, do the same except prefix with “fnPhy\_”.
2. Keep database object names less than 50 characters. Very long names sometimes can’t be displayed in Management Studio or other utilities. Other systems often don’t support very long table names.
3. Avoid creating heaps (tables without indexes). If not sure what to do, consult your Technical Architect or just create a clustered index on the primary key – which is the default
4. No spaces or special characters as those will require special coding to work with them.

***For example: ‘Phone List’ would need to be coded as [Phone List]***

1. Avoid using system reserved names or keyword as an object name

***For example: Bulk, check, first, second, move, name, key, status, etc.***

1. Never use the ***recompile*** option in stored procedure development.
2. Use SET NOCOUNT ON at the beginning of your SQL batches, stored procedures and triggers in production environments, as this suppresses messages like '(1 row(s) affected)' after executing INSERT, UPDATE, DELETE and SELECT statements. This improves the performance of stored procedures by reducing network traffic.
3. Use single-line comment markers where needed (--). Reserve multi-line comments (/\*..\*/) for blocking out sections of code.
4. Error reporting
   * 1. Avoid abbreviations other than the specified prefixes and postfixes in error messages.
     2. Try using system messages stored in syscomments.
5. Error messages should be added to the system using the following outline:

**sp\_addmessage** [**@msgnum =**] *msg\_id***,**[**@severity =**] *severity***,** [**@msgtext =**]**'***msg***'**  
                                                [**,** [**@lang =**] **'***language***'**]   
                                                [**,** [**@with\_log =**] **'***with\_log***'**]  
                                                [**,** [**@replace =**] **'***replace***'**]

* + 1. Assign error message numbers based in the following

**Reserved                                               50000 thru 50999**

**General errors                                     51000 thru 51099**

**Import errors                                       52000 thru 52099**

**Export errors                                        53000 thru 53099**

**Services errors                     58000 thru 58099**

**DbChangeControl Message     59000 thru 59099**

### General Database Programming Guidelines

* + - 1. Make sure you normalize your data at least to the 3rd normal form. At the same time, do not compromise on query performance. A little bit of denormalization helps queries perform faster.
      2. Write comments in your stored procedures, triggers and SQL batches generously, whenever something is not very obvious. This helps other programmers understand your code clearly. Don't worry about the length of the comments, as it won't impact the performance, unlike interpreted languages like ASP 2.0.
      3. Try to avoid server side cursors as much as possible. Always stick to a 'set-based approach' instead of a 'procedural approach' for accessing and manipulating data. Cursors can often be avoided by using SELECT statements instead.
      4. If a cursor is unavoidable, use a WHILE loop instead. A WHILE loop is always faster than a cursor. But for a WHILE loop to replace a cursor you need a column (primary key or unique key) to identify each row uniquely. However, every table must have a primary or unique key.
      5. Avoid the creation of temporary tables while processing data as much as possible, as creating a temporary table means more disk I/O. Consider using advanced SQL, views, SQL Server 2000 table variable, or derived tables, instead of temporary tables.
      6. Try to avoid wildcard characters at the beginning of a word while searching using the LIKE keyword, as this results in an index scan, which defeats the purpose of an index. The following statement results in an index scan, while the second statement results in an index seek:  
         *SELECT LocationID FROM Locations WHERE Specialities LIKE '%pples'  
                         SELECT LocationID FROM Locations WHERE Specialities LIKE 'A%s'*  
         Also avoid searching using not equals operators (<> and NOT) as they result in table and index scans.

* + - 1. Use 'Derived tables' wherever possible, as they perform better. Consider the following query to find the second highest salary from the Employees table:

SELECTMIN(Salary)   
FROM Employees   
WHERE EmpID IN (SELECT TOP 2 EmpID   
                FROM Employees   
                ORDER BY Salary Desc)   
The same query can be re-written using a derived table, as shown below, and it performs twice as fast as the above query:

SELECTMIN(Salary)   
FROM (SELECT TOP 2 Salary   
      FROM Employees   
      ORDER BY Salary DESC) AS A

* + - 1. While designing your database, design it keeping "performance" in mind. You can't really tune performance later, when your database is in production, as it involves rebuilding tables and indexes, re-writing queries, etc. Use the graphical execution plan in Query Analyzer or SHOWPLAN\_TEXT or SHOWPLAN\_ALL commands to analyze your queries. Make sure your queries do an "Index seek" instead of an "Index scan" or a "Table scan." A table scan or an index scan is a very bad thing and should be avoided where possible. Choose the right indexes on the right columns.
      2. Use the more readable ANSI-Standard Join clauses instead of the old style joins. With ANSI joins, the WHERE clause is used only for filtering data. Whereas with older style joins, the WHERE clause handles both the join condition and filtering data. The first of the following two queries shows the old style join, while the second one shows the new ANSI join syntax:

SELECT a.au\_id, t.title   
FROM titles t, authors a, titleauthor ta  
WHERE a.au\_id = ta.au\_id AND  
    ta.title\_id = t.title\_id AND   
    t.title LIKE'%Computer%'  
  
SELECT a.au\_id, t.title  
FROM authors a   
    INNER JOIN titleauthor ta ON a.au\_id = ta.au\_id  
    INNER JOIN titles t ON ta.title\_id = t.title\_id  
WHERE t.title LIKE'%Computer%'

* + - 1. Use User Defined Data types if a particular column repeats in a lot of your tables, so that the data type of that column is consistent across all your tables.
      2. Do not let your front-end applications query/manipulate the data directly using SELECT or INSERT/UPDATE/DELETE statements. Instead, create stored procedures, and let your applications access these stored procedures. This keeps the data access clean and consistent across all the modules of your application, and at the same time centralizing the business logic within the database.
      3. Use the CHAR data type for a column only when the column is non-nullable. If a CHAR column is nullable, it is treated as a fixed length column in SQL Server 7.0+. So, a CHAR(100), when NULL, will eat up 100 bytes, resulting in space wastage. So, use VARCHAR(100) in this situation. Of course, variable length columns do have a little processing overhead over fixed length columns. Carefully choose between CHAR and VARCHAR depending up on the length of the data you are going to store.
      4. Avoid dynamic SQL statements as much as possible. Dynamic SQL tends to be slower than static SQL, as SQL Server must generate an execution plan every time at runtime. IF and CASE statements come in handy to avoid dynamic SQL.
      5. Minimize the use of NULLs, as they often confuse the front-end applications, unless the applications are coded intelligently to eliminate NULLs or convert the NULLs into some other form. Any expression that deals with NULL results in a NULL output. ISNULL and COALESCE functions are helpful in dealing with NULL values.
      6. Perform all your referential integrity checks and data validations using constraints (foreign key and check constraints) instead of triggers, as they are faster. Limit the use triggers only for auditing, custom tasks and validations that cannot be performed using constraints.
      7. Keep your transactions as short as possible. Touch as few data as possible during a transaction. Never, ever wait for user input in the middle of a transaction. Do not use higher level locking hints or restrictive isolation levels unless they are absolutely needed. Make your front-end applications deadlock-intelligent, that is, these applications should be able to resubmit the transaction incase the previous transaction fails with error 1205. In your applications, process all the results returned by SQL Server immediately so that the locks on the processed rows are released, hence no blocking.
      8. Offload tasks, like string manipulations, concatenations, row numbering, case conversions, type conversions etc., to the front-end applications if these operations are going to consume more CPU cycles on the database server. Also try to do basic validations in the front-end itself during data entry.
      9. Do not call functions repeatedly within your stored procedures, triggers, functions and batches. For example, you might need the length of a string variable in many places of your procedure, but don't call the LEN function whenever it's needed, instead, call the LEN function once, and store the result in a variable for later use.
      10. Make sure your stored procedures always return a value indicating their status. Standardize on the return values of stored procedures for success and failures. The RETURN statement is meant for returning the execution status only, not data. If you need to return data, use OUTPUT parameters. If your stored procedure always returns a single row result set, consider returning the result set using OUTPUT parameters instead of a SELECT statement, as ADO handles output parameters faster than result sets returned by SELECT statements.
      11. To make SQL Statements more readable, start each clause on a new line and indent when needed.

***Following is an example:***  
                SELECT t.TitleID, t.title, p.PurchaseNo

                FROM Titles t

                INNERJOIN Purchases p ON p.TitleID = t.TitleID

                WHERE t.Title LIKE'%Computer%'

                AND t.Title LIKE'%cook%'

* + - 1. Do not use GOTO, or use it sparingly. Excessive usage of GOTO can lead to hard-to-read-and-understand code.
      2. Do not forget to enforce unique constraints on your alternate keys.
      3. Use variables instead of constant values within your queries to improve the readability and maintainability of your code.
      4. Use isolation level hints to minimize locking. For example, use “with (nolock)” when retrieving data if you are not concerned about retrieving records that we in the processes of being changed.
      5. Avoid the use of index hints; Database optimization engine knows how to identify the best index to use.
      6. Avoid direct use of system tables as system tables could change in future releases, use INFORMATION\_SCHEMA views instead.
      7. Avoid the use of UNICODE data types such as nvarchar, or ntext unless you are storing special characters such as the ones found in some languages besides English.
      8. Place all declare statements before any other code in the procedure to give the query optimizer the best shot at reusing query plans.
      9. Use parenthesis to increase readability, especially when working with branch conditions or complicated expressions.

## RESTful servicesGuidelines

The concept of REST is to separate the API structure into logical resources. There are used the HTTP methods GET, DELETE, POST and PUT to operate with the resource

1. GET method and query parameters should not alter the state

Use **PUT, POST** and **DELETE** methods instead of the **GET** method to alter the state.  
Does not use **GET** for state changes?

1. Do not mix up singular and plural nouns. Keep it simple and use only plural nouns for all resources.
2. Use HTTP headers for serialization formats

Both, client and server need to know which format is used for the communication. The format has to be specified in the HTTP-Header.

***Content-Typ****e* defines the request format.  
***Accept***defines a list of acceptable response formats.

1. Use HATEOAS

**H**ypermedia **a**s **t**he **E**ngine **o**f **A**pplication **S**tate is a principle that hypertext links should be used to create a better navigation through the API.

1. Use nounsin the URL but no verbs
2. It's highly recommended to use a query string toreduce the complexity of a URL that has several different associations

The following shows how to use a query string that's somewhat complicated and unintuitive to the user:

GET /trainers/12/training/11/zip/75080/city/richardson/state/tx

The following shows a better way to write a query string:

GET /trainers/12/training/11?zip=75080&city=richardson&state=tx

1. Allow overriding HTTP method

Some proxies support only **POST** and **GET** methods. To support a RESTful API with these limitations, the API needs a way to override the HTTP method.

Use the custom HTTP Header **X-HTTP-Method-Override** to over rider the POST Method.

## MVC applicationsGuidelines

1. Disable Request Validation

Request Validation is a feature that prevents potentially dangerous content from being submitted. This feature is enabled by default. However, at times you might need your application to post HTML mark-up tags to the server. You would then need this feature to be disabled. Here is how you can do it:

[ValidateInput(false)]

[ValidateInput(HttpVerbs.Post)]

public ActionResultCreate([Bind(Exclude="Id")]EmployeeempObj)

{

}

1. Use Strongly Typed Models

A strongly typed view is a view that defines its data model as a CLR type instead of a weakly typed dictionary that may contain potentially anything. To create a strongly typed view, check the "Create a strongly-typed view" checkbox while you are creating the view. If you plan to create a strongly typed view manually later, ensure that your view "Inherits"

System.Web.Mvc.<Your Namespace>.<YourClass>

1. Cache Pages that Contain Shared Data or are Public and don't Require Authorization

You should not cache pages that need authorization in ASP.NET MVC. You should not cache pages that contain private data or need authorization. Caching pages in ASP.NET MVC is simple - just specify the OutputCache directive as shown

[OutputCache(Duration = 60)]

public ActionResult Index()

{

  return View("Index", somedata);

}