.Net Coding Guidelines

v1.0 (23rd June 2017)

Architecture

- Lightweight your controller and heavy your services: Avoid writing excessive code in controller, instead perform the business logic and data operation work in service classes.
- Use dependency injection to create instances of repository and service classes.
- Avoid using multiple if condition in a code block, instead divide that in multiple functions.
- Avoid adding more stress on classes by assigning more responsibilities on them, this means that every class in your code should only have one job to do and everything in this class should be related with single purpose.
- Always try to design your classes in such a way that the new functionality can be added only when new requirements are generated and it should not be altered until you find any bugs into it.
- Always ensure that new derived classes extend the base classes without changing their behavior and if it’s doing so make the changes in the code.
- Classes should not be forced to implement interfaces they don't use. Instead of one bulky interface, many small interfaces are preferred based on groups of methods, each one containing one sub module.
- Always try to decouple your classes by using abstraction, if a class “A” is directly dependent on a different class “B” then any change in the “B” class will also force to change in “A” class, so to avoid this use abstraction to define those dependencies.
- Follow DRY principle means write once and runs everywhere
- Keep code simple and straight

Reference: S.O.L.I.D Principles
https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
http://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

C#

- DO use Pascal Casing for all public member (method and property), type (class, interface, enum and struct) and namespace names consisting of multiple words.

Sr. No.	Type	Case	Example
1	Method	Pascal	public Company **GetCompany**(int id){ }
2	Property	Pascal	public byte **Salutation** { get; set; }
3	Class	Pascal	public class **EmployeeService** { }
4	interface	Pascal	public interface **IEmployeeService**{ }

5	enum	Pascal	public enum **AbsenceTypeEnum** : byte
			{
			Planned=1,
			Unplanned=2
			}

6	struct	Pascal	public struct EmployeeCalendar
			{
			public long EmployeeId { get; set; }
			public DateTime StartDate { get; set; }
			
			}

- DO use camel Casing for parameter and variable names.
- DO choose easily readable identifier names.
  For example, a property named HorizontalAlignment(✓) is more English-readable than AlignmentHorizontal(**X**).
- DO favor readability over brevity.
  The property name CanScrollHorizontally(✓) is better than ScrollableX(**X**) (an obscure reference to the X-axis).
- AVOID using identifiers that conflict with keywords of widely used programming languages.
- DO NOT use abbreviations or contractions as part of identifier names.
  For example, use GetWindow(✓) rather than GetWin(**X**) .
- DO choose names for your assembly DLLs that suggest large chunks of functionality. CONSIDER naming DLLs according to the following pattern:
  <ProjectName>.<Component>.dll (like DS.Web)
- DO NOT use the same name for a namespace and a type in that namespace.
- DO name classes and structs with nouns or noun phrases, using PascalCasing.

```
public struct EmployeeCalendar
{
    public long EmployeeId { get; set; }
    public DateTime StartDate { get; set; }
    public int? CalendarYear { get; set; }
}
```

- DO ensure that the names differ only by the "I" prefix on the interface name when you are defining a class–interface pair where the class is a standard implementation of the interface.
- DO add the suffix "Attribute" to names of custom attribute classes. Add the suffix "Attribute" to names of custom attribute classes.
- DO NOT explicitly specify a type of an enum or values of enums (except bit fields).
- DO name properties using a noun, noun phrase, or adjective.

```
public class CalendarDateViewModel{
    public long RotaScheduleId { get; set; }
    public string Start { get; set; }
    public string ScheduleDate { get; set; }
    public string OvertimeStartTime { get; set; }
    public bool IsLeaving { get; set; }
    public string LeaveColor { get; set; }
}
```

- DO name collection properties with a plural phrase describing the items in the collection instead of using a singular phrase followed by "List" or "Collection."

- DO name Boolean properties with an affirmative phrase (CanSeek(✓) instead of CantSeek(X)). Optionally, you can also prefix Boolean properties with "Is," "Can," or "Has," but only where it adds value.
- DO use nested types when the relationship between the nested type and its outer type is such that member-accessibility semantics are desirable. If there are no nested type then create type in separate file.
  Example: Suppose a class which is used with parent class then we create nested class such EmployeeLogin class is used with Employee.

```
public class Employee {
    public string Email { get; set; }
    class EmployeeLogin {
           public string Password { get; set; }
   }
}
```

- DO NOT declare or override instance members in static classes.
- DO NOT call method in conditional statements.
- DO use read-only repository in Service project's classes like below -

```
public class RotaScheduleService : IRotaScheduleService {
    private readonly IRepository<RotaSchedule> repoRotaSchedule;
    public RotaScheduleService(IRepository<RotaSchedule> repoRotaSchedule)
    {       this.repoRotaSchedule = repoRotaSchedule;      }
    public void Dispose()
    {
       if (repoRotaSchedule != null)
       {          repoRotaSchedule.Dispose();         }
    }
}
```

- Do use read-only services in Web.

```
public class RotaScheduleController : BaseController{
       private readonly IRotaScheduleService rotaScheduleService;
       public RotaScheduleController(IRotaScheduleService rotaScheduleService)
       {   this.rotaScheduleService = rotaScheduleService;   }
      protected override void Dispose(bool disposing)
       {
          if (rotaScheduleService != null)
          { rotaScheduleService.Dispose(); }
             base.Dispose(disposing);
       }
 }
```

- Do use interface only from web and create instance using DI.

```
//Do
public class UserCoverController : BaseController{
         private readonly IScheduleCoverService scheduleCoverService;
         public UserCoverController(IScheduleCoverService scheduleCoverService)
```

```
        {   this.scheduleCoverService = scheduleCoverService;    }
}
//Don't
public class UserCoverController : BaseController{
        private ScheduleCoverService scheduleCoverService;
        public UserCoverController()
        {    scheduleCoverService = new ScheduleCoverService();   }
}
```

- Do use const string rather than static variables.
- DO NOT use unused usings.
- DO use comments on public method and types. Always include<summary> comments. Include <param>, <return>, and <exception>comment sections where applicable
- All comments should be written in the same language, be grammatically correct, and contain appropriate punctuation. DO Use // or /// but never /* … */.
- DO use dispose in controller and class if required.
- DO inherit IDisposable on interfaces.
- DO use exception handling in complex method.
- DO refactor (Split in small block) large method and classes.
- DO name source files according to their main classes.
- DO use If statement with curly brackets.
- DO use .Net Framework namespaces on top in using.
- DO NOT include the parent class name within a property name.
- DO use #region filed for combined methods of class. DO NOT use for #region in the method.
- Do not omit access modifiers. Explicitly declare all identifiers with the appropriate access modifier instead of allowing the default.
- Try to initialize variables where you declare them.
- Always use the built-in C# data type aliases, not the .NET common type system (CTS). like String vs string

```
public  class  Job{
//Do
 public int  JobId  {  get;  set;  }

//Don't
  public  Int32  JobId  {  get;  set;  }

  }
```

- Avoid direct casts. Instead, use the “as” operator and check for null.

```
//Do
var  model  =  (ViewData["NotificationModel"]  ??
TempData["NotificatonModel"])  as  LD.Web.Models.Others.Notification;
//Don't
var  model  =  (ViewData["NotificationModel"]  ??
(LD.Web.Models.Others.Notification)TempData["NotificationModel"]);
```

- Try to use the “@” prefix for string literals instead of escaped strings. Prefer String.Format() or StringBuilder over string concatenation. Never concatenate strings inside a loop.
- Avoid using foreach to iterate over immutable value-type collections. E.g. String arrays.
  ```
  int[] days = Enumerable.Range(1, 10).ToArray();
  //Do
  for (int i = 0; i < days.Length;i++ )
  {    //operation goes here    }
  //Don't
  foreach (int day in days)
  {    //operation goes here }
  ```
- Do not modify enumerated items within a foreach statement.
- Use the ternary conditional operator only for trivial conditions. Avoid complex or compound ternary operations. Example: int result = isValid ? 9 : 4;
- Avoid evaluating Boolean conditions against true or false.
  ```
  //Don't
  bool  isValid  =  ModelState.IsValid  ?  true  :  false;
  //Don't
  if(ModelState.IsValid  ==  true)
  {      }
  //Do
  if(ModelState.IsValid)
  {    }
  ```
- Only use switch/case statements for simple operations with parallel conditional logic.
- Prefer nested if/else over switch/case for short conditional sequences and complex conditions.
- Never declare an empty catch block and Avoid nesting a try/catch within a catch block.
- Only use the finally block to release resources from a try statement.
- Do use extension method for commonly used functionality on type.
- Constant - Use uppercase for constant variables with words separated by underscores. It is recommended to use a grouping naming schema.
  Example (for group AP_WIN): AP_WIN_MIN_WIDTH, AP_WIN_MAX_WIDTH
- Use inline-comments to explain assumptions, known issues, and algorithm insights.

MVC

- **Use BaseController**
  It is recommended to use Base Controller with our Controller and this Base Controller will inherit Controller class directly. Using Base Controller, we can write common logic like access authentication users information which could be shared by all Controllers.

- **Use Entity Model instead of ViewModel (Dto) wherever possible.**
  ViewModel is a class just like Entity Model class but it contains some extra properties required only for business process. If there are extra properties then we can use ViewModel.

- **Strongly Type View**
  When working with View and passing data from Controller, We should use Strongly Typed View and map with a ViewModel directly. Avoid overuse of ViewBag, ViewData and TempData.

- **Use HTTP Verbs**
  Always use HTTP verbs suitable to action method such as HttpPost or HttpGet.

- **Use Fluent Validation**
  There should be separate validation classes in specific folder or might be separate class library in your application. Use validation factory. We prefer fluent validations.

- **Shared Folder**
  In Shared folder we should keep views/partial views which used globally across the application like _Layout.cshtml as a master page or partial view which need to use from various controller actions. We recommend that strongly related view/partial view must be placed in their specific View folders. Just like if we are using a product add/edit view and its using only from product controller then it should be placed in Views' >> Product.

- **Use proper pre-fix with partial view**
  If you are using partial view than use underscore (_) as prefix like _ Delete.cshtml

- **DO NOT place database calls in your Html Helpers**
  Confirm the controller sends ALL of the data needed to make a complete View.

- **Use meaningful view names**
  To create view for respective action result follows naming conventions. Suppose you want to create CRUD operation then use Add.cshtml, Edit.cshtml, Delete.cshtml, Index.cshtml, List.cshtml.

- **Use Bundling and Minification (for CSS and Scripts)**
  we should always use Bundling and Minification. Using Bundling and Minification, we can make fewer requests on server and decrease the file size, which will download faster. Minification is achieved to use .min file. We can do Bundling in BundleConfig class which is located in App_Start folder.

- **Routing**
  Try to use pattern matching routing

- **Exception Email**
  Global exception email should be implemented with application so that email can be trigger to developer if application generate any exception.
- **Action Filter**
  If you need to perform some checks on most of controller or actions then you use Action Filter.

- **Don't use scripting code** on views directly

LINQ

- Don’t write all query in one line of code. Write like below :

```
repoPerson.Query()
      .Filter(predicate)
      .OrderBy(o => o.OrderBy(oo => oo.Surname))
      .Get()
      .ToList()
```

- Avoiding FOREACH on Collections to Filter for Data
- Try to use stored procedures when LINQ statement become complex and may have speed issues
- Always use below type of code:

```
DB.Events.Where(E => E.EventId == SomeId).FirstOrDefault();
Instead use
DB.Events.FirstOrDefault(E => E.EventId == SomeId);
```

- Use IEnumerable<T> instead on List<T> if you don’t want this collection for further manipulation.
- Avoid fetching all the fields if not required
- Retrieve only required number of records like use paging
- Always use Any() method instead of count to check the record exists

Entity Framework

- Keep context open for as long as necessary
- Use Pre-Generating Views to reduce response time for first request
- Avoid to put all the DB Objects into One Single Entity Model
- Disable change tracking for entity if not needed
- Avoid using Contains
- Use Eager Loading when relations in a table are so much while searching or selecting the relationship data
- Debug and Optimize LINQ Query
- Don’t change anything in EDMX file like Navigation Properties Name, Create Custom Fields etc.

JavaScript/jQuery

Loading JS external file

- Try to use CDN url if available for external JS file like as below for jQuery library:
  `<script src="//ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>`
- Include below line after CDN url in case of fallback.
  `<script>window.jQuery || document.write('<script src="/js/jquery-3.2.1.min.js"><\/script>')</script>`

- Always add JavaScript/jQuery/plugin script files at the bottom of the page
- Use protocol independent URL (without http or https).
- Use only one version of JQuery (use latest version if not dependent on any plugin)
- Use minified file.

JQuery Selectors & DOM manipulation

- Use ID selector whenever possible. It is faster because they are handled using document.getElementById(). Ex. $("#products")
- When using class selectors, don't use the element type in your selector.
- Performance Comparison
  ```
  var $products = $("div.products"); // SLOW
  var $products = $(".products"); // FAST
  ```

- Use find to select child object of a ID selector
  ```
  var $productIds = $("#products").find("div.id");
  ```

- Give your Selectors a Context.
  ```
  $('.class');        // SLOWER because it has to traverse the whole DOM for .class
  $('.class', '#class-container'); // FASTER because now it only looks under class-container.
  ```

- Don't Descend Multiple IDs or nest when selecting an ID. ID-only selections are handled using document.getElementById() so don't mix them with other selectors.
  ```
  $('#outer #inner'); // BAD
  $('div#inner'); // BAD
  $('.outer-container #inner'); // BAD
  $('#inner'); // GOOD, only calls document.getElementById()
  ```

- Always use $ with a variable which you want to cache or you need to perform any action on it.
  ```
  $myDiv    = $("#myDiv");
  ```
  Now perform action on:
  ```
  $myDiv.click(function(){});
  ```

- Use string concatenation or array.join() over .append()
  ```
  // BAD
  var $myList = $("#list");
  for(var i = 0; i < 10000; i++){
      $myList.append("<li>"+i+"</li>");
  }

  // GOOD
  var $myList = $("#list");
  var list = "";
  for(var i = 0; i < 10000; i++){
      list += "<li>"+i+"</li>";
  ```

```
}
$myList.html(list);

//EVEN FASTER
var array = [];
for(var i = 0; i < 10000; i++){
        array[i] = "<li>"+i+"</li>";
}
$myList.html(array.join(''));
```

- Use only one Document Ready handler per page
- DO NOT put request parameters in the URL, send them using data object setting.

```
// Less readable...
$.ajax({
        url: "something.php?param1=test1&param2=test2",
       ....
});

// More readable...
$.ajax({
        url: "something.php",
        data: { param1: test1, param2: test2 }
});
```

- Always choose a plugin with good support, documentation, testing and community support.
- Check the compatibility of plugin with the version of jQuery that you are using.
- Check free plugins LICENCE for commercial use before integration

SQL

- Do not use SELECT * FROM <Table Name>
- When naming tables, express the name in the singular form. For example, use `Employee` instead of `Employees`.
- When naming columns of tables, do not repeat the table name; for example, avoid having a field called `EmployeeLastName` in a table called `Employee`.
- Avoid unnecessary use of temporary tables
  - Use 'Derived tables' or CTE (Common Table Expressions) wherever possible, as they perform better
- Avoid using <> as a comparison operator
  - Use ID IN(1,3,4,5) instead of ID <> 2
- Do not define default values for fields. If a default is needed, the front end will supply the value.
- Define all constraints, other than defaults, at the table level.
- Minimize the use of NULLs, as they often confuse 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.
  - The ISNULL and COALESCE functions are helpful in dealing with NULL values.

- When executing an UPDATE or DELETE statement, use the primary key in the WHERE condition, if possible. This reduces error possibilities.
- Avoid dynamic SQL statements as much as possible.
- Avoid multiple Joins in a single SQL Query
- Consider VIEWS for complex queries which used multiple times in the project so that we can put the repetitive logic at same place.
- Always create the foreign key constraint if you use relationship in two tables
- Do not prefix stored procedures with sp_, because this prefix is reserved for identifying system-stored procedures.
- Always use ‘datetime2’ data type in SQL 2008 instead of the classic ‘datetime’
- Create indexes on tables that have high querying pressure using select statements. Be careful not to create an index on tables that are subject to real-time changes using CRUD operations.
- If you need to verify the existence of a record in a table, don’t use SELECT COUNT (*) in your Transact-SQL code to identify it, which is very inefficient and wastes server resources. Instead, use the Transact-SQL IF EXITS to determine if the record in question exits, which is much more efficient.

  DO NOT USE: IF (SELECT COUNT(*) FROM table_name WHERE column_name = 'xxx')
  LINQ counterpart: **db**.table.count()
  USE: IF EXISTS (SELECT * FROM table_name WHERE column_name = 'xxx')
  LINQ counterpart: db.table.Any()

- Always name your constraints and start the name with proper prefix. For example: PK_ for for primary key, FK_ for foreign key etc.
- Try to write sql query to create/alter table instead of design, It will be helpful to manage the version to upload on server