

# **Development Standards** Version 1.0

Contents

[**Coding Best Practices** Version 1.0 1](#_Toc316475379)

[Contents 2](#_Toc316475380)

[Acknowledgements 4](#_Toc316475381)

[Status 4](#_Toc316475382)

[Core Standard 5](#_Toc316475383)

[Naming conventions 5](#_Toc316475384)

[Code layout 7](#_Toc316475385)

[Code style 9](#_Toc316475386)

[Exceptions 12](#_Toc316475387)

[Framework Guidelines 13](#_Toc316475388)

[Best Practice Guidelines 14](#_Toc316475389)

[Golden Rules 19](#_Toc316475390)

[Refactoring 19](#_Toc316475391)

[Patterns 19](#_Toc316475392)

[Pragmatic practices 19](#_Toc316475393)

[Language analysis 19](#_Toc316475394)

[JavaScript Coding Standards 20](#_Toc316475395)

[JavaScript Best Practices 20](#_Toc316475396)

[Always Use 'var' 20](#_Toc316475397)

[Feature-Detect Rather Than Browser-Detect 20](#_Toc316475398)

[Use Square Bracket Notation 20](#_Toc316475399)

[Avoid 'eval' 20](#_Toc316475400)

[Avoid 'with' Statements 20](#_Toc316475401)

[Avoid Cluttering The Global Namespace 20](#_Toc316475402)

[Avoid sync "Ajax" calls 20](#_Toc316475403)

[Use JSON 20](#_Toc316475404)

[Use Correct <script> Tags 20](#_Toc316475405)

[JQuery Best Practices 20](#_Toc316475406)

[Component Development Best Practices 20](#_Toc316475407)

[Client Template Best Practices 20](#_Toc316475408)

[Useful Tools 20](#_Toc316475409)

[Suggested Reading 20](#_Toc316475410)

[References 20](#_Toc316475411)

[Books 20](#_Toc316475412)

[CSS Coding Standards 21](#_Toc316475413)

[TBD – Ron and team 21](#_Toc316475414)

Preface

A straight forward coding standard is essential for successful product delivery. This document aims to provide clear and concise instruction in conventions and practices that lead to first-class production code.

Developers are merely asked to comply with this standard; an understanding of the rationale behind the directives will follow with experience.

The elements of this standard are classified as follows:

1. **Core Standard** – everyone must adhere to these rules in all new code.
   1. Naming Conventions
   2. Code Layout
   3. Code Style
   4. Exceptions
2. **Framework Guidelines** – specific coding practices
3. **Best Practice Guidelines** – developers should consider these rules as mentoring guidance, to be followed in most cases, unless there is a business case to do otherwise.
4. **Golden Rules** – good ideas that make for better software.
5. **Suggested reading** – a selection of articles that provide insight into related areas of best practice.

## Acknowledgements

Portions of this standard are ©2008 IDesign, Inc.

## Status

This is a living document. The organization of the rules and general content are subject to change; it is suggested you check for changes on a regular basis.

DO NOT edit this standard without consultation. Please forward comments, suggestions and criticism to ARCH for inclusion in the next revision.

/\*

"Naming conventions cannot guarantee good code ... only the skill of the programmer can."

-- Charles Simonyi

\*/

## C# Standards

*Items in this section are* ***mandatory*** *for all new code.*

### Naming conventions

1. Use Pascal casing for types, properties, method names and constants.
   1. The abbreviation ID should only be used as an identifier suffix and must appear uppercase at all times.

public class SomeClass

{

private const int DefaultSize = 100;

public int CurrentUserID { get; set; }   
 public void GetHTML()   
 {…}

}

1. Use camel casing for local variable names and method arguments.
   1. Acronyms should be all lowercase.
   2. Combination words such as ILot should be all lowercase.
   3. The abbreviation ID should only be used as an identifier suffix and must appear uppercase at all times.

private void MyMethod(int itemID)

{

int number;

}

1. **Never** prepend variables with type information (i.e. Hungarian notation)

public class SomeClass

{

// Correct:

private int \_number;

// Avoid:

private int i\_number;

}

1. Prefix private member variables with an underscore. Use camel case after the underscore.

public class SomeClass

{

private int \_number;

}

1. Prefix interface names with I.

interface IMyInterface

{…}

1. Do not use scope indicators; rely instead on identifier casing and leading underscores.

// Correct:

private int \_componentID;

// Avoid:

private int ThisComponentID;

private int m\_componentID;

1. Explicitly state visibility with access modifiers, even if this restates the default visibility.

// Correct:

private Chart GetChart(…)

{…}

// Avoid:

Chart GetChart(…)

{…}

1. Never suffix any identifier with a reserved word.

// Correct:

IPermissionManager PermissionManager { get; }

// Avoid:

IPermissionManager PermissionManagerInterface { get; }

1. Suffix custom attribute classes with Attribute.
2. Suffix custom exception classes with Exception.
3. Name extension classes with the type they extend, suffixed with Extensions.

public static class StringExtensions

1. Name methods using verb-object pair, such as ShowDialog().
2. Methods with return values should describe the type returned, such as GetComponent().
3. Use descriptive variable names.
   1. Avoid using Hungarian notation – even when using it the right way.
   2. Avoid single character variable names, such as i or t. Use index or temp instead.
   3. Emphasize semantics, not structure: prefer the name components to items.
   4. Do not abbreviate words (such as num instead of number).
4. Always use C# keywords rather than the aliases in the System namespace.

// Correct:

object item1;

string item2;

int item3;

// Avoid:

Object item1;

String item2;

Int32 item3;

1. Prefix generics with T
   1. Reserve suffixing Type when dealing with the .NET type Type.

// Correct:

public class LinkedList<TKey,TEntity>   
{…}

// Avoid:

public class LinkedList<KeyType,EntityType>   
{…}

### Code layout

1. Classes should follow this general pattern. Comments are for illustration only.

public class SomeClass

{

// Static variables first

private static object \_expensiveObject;

// Constants and readonly variables

private const SomeConstant;

private readonly int \_someReadonlyNumber;

// Member variables

private int \_number;

// Properties (backing fields should be directly below the property)

public int Property

{

get { return \_property; }

}

private int \_property;

public string AnotherProperty { get; set; }

// Events

public event DelegateName EventName;

// Constructors

public SomeClass()

{…}

// Methods (in functional order and/or alphabetical order)

public void DoSomething()

protected void DoSomethingElse()

private void SomeHelper()

// Static Methods

public static void CreateSomeClass()

{…}

}

1. Maintain strict indentation using tabs. Do not use spaces for formatting.
2. Indent comments at the same level of indentation as the code you are documenting.
3. Declare a local variable as close as possible to its first use.
4. File names should match the name of the class/artifacts within
5. When multiple classes inherit from the same base class without any new implementation, they may be located in the same file. The file name should be the plural for the base class:

// In CountAttributes.cs

public class OneAttribute : CountAttribute {}

public class TwoAttribute : CountAttribute {}

public class ThreeAttribute : CountAttribute {}

1. When using partial types and allocating a part per file, name each file after the logical role that part plays.  
   For example:

// In MyClass.cs

public partial class MyClass   
{...}

// In MyClass.Generated.cs

public partial class MyClass   
{...}

1. Avoid putting multiple classes in a single file.
2. A single file should contribute types to only a single namespace. Avoid having multiple namespaces in the same file.
3. As a general rule:
   1. avoid files with more than 500 lines.
   2. avoid methods with more than 30 lines.
4. Avoid methods with more than 5 arguments. Use structures for passing multiple arguments.
5. Lines should not require excessive scrolling to read.

### Code style

1. Use meaningful namespaces, such as:

namespace DowJones.Web.Mvc.UI.Components

1. Avoid fully qualified type names. Prefer the using statement instead.
2. Group all framework namespaces together, followed by third-party and custom namespaces.

using System;

using System.Collections.Generic;

using ThirdParty.Library;

using DowJones.Web.Mvc;  
using DowJones.Web.Mvc.UI.Components;

1. Remove unused using statements.
2. Use Action<> or Func<> over delegates
3. All comments should pass spell checking
4. All humorous identifier names should be funny. Obscure humor should be commented.

StringBuilder bob = new StringBuilder();

StreamWriter paulBunyan = new StreamWriter(logfile); // get it? log... Paul Bunyan?

1. Prefer Automatic Properties

// Correct:

public string Name { get; set; }

public ClassName()

{

Name = DefaultName;

}

// Avoid:

public string Name

{

get { return \_name; }

set { \_name = value; }

}

private string \_name = DefaultName;

1. Avoid accessing property backing fields directly:

public string Name

{

get { return \_name; }

set { \_name = value; }

}

private string \_name;

// Correct:

Name = "Bob";

// Avoid:

\_name = "Bob";

1. Limit type and member accessibility to the most private possible level.
2. Avoid Inner classes
3. Always place an open curly brace ({) in a new line, except for single line property accessors and auto-implemented properties.
4. The only acceptable use of multiple semicolons on a single line is when initializing a for loop.
5. With Lambda expressions, mimic the code layout of a regular method, aligned with the delegate declaration. Omit the variable type and rely on type inference, yet use parentheses:

public delegate void SomeDelegate(string someString);

SomeDelegate someDelegate = (name) =>   
 {

Trace.WriteLine(name);   
 MessageBox.Show(name);   
 };

1. Only use in-line Lambda expressions when they contain a single simple statement. Avoid multiple statements that require a curly brace or a return statement with in-line expressions. Omit parentheses:

public delegate void SomeDelegate(string someString);

public void MyMethod(SomeDelegate someDelegate)   
{…}

// Correct:

MyMethod(name => MessageBox.Show(name));

// Avoid:

MyMethod((name) => {Trace.WriteLine(name);MessageBox.Show(name);});

1. Always use a curly brace scope in an if statement, even if it conditions a single statement.
   1. Flow-control statements are exempt from this rule; specifically the keywords return, break and continue.
   2. You may throw on the same line, assuming you’re not creating a new Exception.
   3. You may return on the same line, as long as you don’t need multiple line initialization:

// Correct:

if (isFoo) return new Bar();

// Avoid:

if (isFoo) return new Bar

{

Thing = "Something",

Something = "Something else"

};

1. Favor the traditional C == equality comparison over .Equals(a,b).
2. When building a long string (more than two concatenations), use StringBuilder, not string, except in the explicit case when you are joining only string literals.
3. Avoid excessive use of StringBuilder; use an external template instead.
4. Never hard-code strings that will be presented to end users. Use tokens.
5. Never hard-code strings that might change based on deployment environment. Use settings.
6. Use string.Empty instead of "":

string name;

/\* Some conditional code to initialize name, then: \*/

if (isUnknown)

{

// Correct:

name = String.Empty;

// Avoid:

name = "";

}

1. Avoid providing methods on structures.
2. Do not “new up” service instances – prefer dependency injection.

### Exceptions

1. Catch only exceptions for which you have explicit handling.
2. 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:

catch(Exception exception)   
{

MessageBox.Show(exception.Message);   
 throw;

}

1. Avoid error codes as method return values.
2. Avoid ‘Expection Handling’. Exceptions should not be the rule.
3. When defining custom exceptions:
   1. Derive the custom exception from Exception.
   2. Ensure serialization is possible.

## Framework Guidelines

1. Data access layer methods may return IQueryable<T>. These methods should be “finders”.

public IQueryable<Entity> FindEntitiesByName(string name)

1. Core layer methods may return IEnumerable<T>. These methods should be “getters”.

public IEnumerable<Entity> GetEntitiesByName(string name)

1. Advertise dependencies using constructor injection:

public class SomeService : ISomeService   
{

private readonly IOtherService \_otherService;

public SomeService(IOtherService otherService)

{

\_otherService = otherService;

}

1. Only in classes where constructors *are not* accessible (WCF services, etc.), advertise dependencies using property injection:

[Inject]

public IOtherService OtherService { get; set; }

## Best Practice Guidelines

1. Become familiar with the architecture of the project – avoid re-inventing the wheel.
2. Avoid clipboard inheritance. Prefer refactoring.
3. Do not manually edit any machine-generated code.
4. If modifying machine generated code, modify the format and style to match this coding standard.
5. Use partial classes whenever possible to factor out the maintained portions.
6. Avoid comments that explain the obvious. Code should be self-explanatory. Good code with readable variable and method names should not require comments.
7. Document only operational assumptions, algorithm insights and so on. Comments describing program flow indicate the need for ‘Extract Method’ refactoring.
8. Encapsulate business rules within a class:

Thing currentThing = new Thing();

// Correct:

if (currentThing.IsValid)

{

currentThing.Save();

}

// Avoid:

if (currentThing.Value > 1 && currentThing.OtherValue == 2)

{

currentThing.Save();

}

1. Declare a constants where needed; do not use magic numbers.
2. Use the const directive only on natural constants such as the number of days of the week.
3. Avoid using const on read-only variables. For that, use the readonly directive.

public class MyClass

{

public const int DaysInWeek = 7;   
 public readonly int Number;   
 public MyClass(int someValue)   
 {

Number = someValue;

}

}

1. Do not initialize value types to their default value.
2. Prefer an enumerated type over a Boolean for method parameters:

// Correct:

public State PerformOperation(State currentState)

{…}

// Avoid:

public State PerformOperation(bool isReady)

1. Avoid multiple Main() methods in a single assembly.
2. Make only the most necessary types public, mark others as internal.
3. Avoid friend assemblies, as they increase inter-assembly coupling.
4. Avoid code that blindly relies on an assembly running from a particular location. Use of a structured settings mechanism to handle this is acceptable.
5. Minimize code in application assemblies. Use class libraries instead to contain business logic.
6. Avoid providing explicit values for enums unless they are bit flags or the enum is generated by a tool.
7. Avoid using the ternary conditional operator for complex logic; simple a or b assignments are acceptable.
8. Avoid using pre-processor directives. Use conditional attributes to exclude method calls:

[Conditional("MySpecialCondition")]

public void MyMethod()

{…}

1. Avoid function calls in Boolean conditional statements. Assign into local variables and check on them.

private bool IsEverythingOK()   
{…}

// Avoid:

if (IsEverythingOK())   
{…}

// Correct:

bool ok = IsEverythingOK();   
if (ok)

{…}

1. Always use zero-based arrays.
2. With indexed collection, use zero-based indexes
3. Do not provide public or protected member variables. Use properties instead.
4. Use automatically implemented properties when there is no business logic or validation for the property.
5. Avoid using the new inheritance qualifier. Use override instead (ensuring consistent operation for polymorphic use):
6. Never use unsafe code, except when using interop.
7. Avoid explicit casting. Use the as operator to defensively cast to a type.

Dog dog = new GermanShepherd();

GermanShepherd shepherd = dog as GermanShepherd;

if (shepherd != null)

{…}

1. Always check a delegate for null before invoking it.
2. Do not provide public event member variables. Use event accessors instead.

public class MyPublisher

{

MyDelegate \_someEvent;

public event MyDelegate SomeEvent   
 {

add

{

\_someEvent += value;

}

remove

{

\_someEvent -= value;

}

}

}

1. Avoid defining event-handling delegates. Use EventHandler<T> instead.
2. Favor the use of interfaces.
3. Classes and interfaces should have at least 2:1 ratio of methods to properties.
4. Avoid data classes (with nothing but properties). Use the ‘Move Method’ refactoring.
5. Avoid interfaces with one member. Interfaces as just a naming container aren’t good either.
6. Strive to have three to five members per interface.
7. Do not have more than 20 members per interface. Twelve is probably the practical limit.
8. Avoid events as interface members.
9. When using abstract classes, offer an interface as well.
10. Expose interfaces on class hierarchies.
11. Prefer using explicit interface implementation.
12. Never assume a type supports an interface. Defensively query for that interface.

SomeType obj1;

IMyInterface obj2;

/\* Some code to initialize obj1, then: \*/

obj2 = obj1 as IMyInterface;

if (obj2 != null)

{

obj2.Method1();

}

else

{

// Handle error in expected interface

}

1. Always code to the most abstract interface or type that supports the feature required:

private void RespondToButtonEvent(object sender, EventArgs e)

{

// Correct:

IButtonControl button = sender as IButtonControl;

if (button != null)

{

string argument = button.CommandArgument; // property is defined by IButtonControl

}

// Avoid:

Button button = sender as Button; // Explicit reference makes refactoring more complex

{…}

}

1. Use application logging and tracing.
2. Never use goto unless in a switch statement fall-through, or to exit a nested for loop.
3. Always have a default case in a switch statement that asserts.

int number = SomeMethod();

switch (number)

{

case 1:

Trace.WriteLine("Case 1:");   
 break;

case 2:

Trace.WriteLine("Case 2:");   
 break;

default:

Debug.Assert(false);   
 break;

}

1. Do not use the this reference unless invoking another constructor from within a constructor.

// Correct use of ‘this’:

public class MyClass

{

public MyClass(string message)   
 {}

public MyClass() : this("Hello")   
 {}

}

1. Do not use GC.AddMemoryPressure().
2. Do not rely on HandleCollector.
3. Always run code unchecked by default (for the sake of performance), but explicitly in checked mode for overflow- or underflow-prone operations:

private int CalcPower(int number, int power)

{

int result = 1;

for (int count = 1; count <= power; count++)   
 {

checked

{

result \*= number;

}

}

return result;

}

1. Avoid casting to and from System.Object in code that uses generics. Use constraints or the as operator instead:

private class SomeClass

{}

// Correct:

private class MyClass<T> where T : SomeClass   
{

private void SomeMethod(T t)   
 {

SomeClass obj = t;

}

}

// Avoid:

private class MyClass<T>   
{

private void SomeMethod(T t)   
 {

object temp = t;

SomeClass obj = (SomeClass)temp;

}

}

1. Do not define constraints in generic interfaces. Interface-level constraints can often be replaced by strong-typing.

public class Customer   
{…}

// Correct:

public interface ICustomerList : IList<Customer>   
{…}

// Avoid:

public interface IList<T> where T : Customer   
{…}

1. Do not define method-specific constraints in interfaces.
2. Do not define constraints in delegates.
3. If a class or a method offers both generic and non generic flavors, always prefer using the generics flavor.
4. When implementing a generic interface that derives from an equivalent non-generic interface (such as IEnumerable<T>), use explicit interface implementation on all methods, and implement the non-generic methods by delegating to the generic ones:

public class MyCollection<T> : IEnumerable<T>

{

IEnumerator<T> IEnumerable<T>.GetEnumerator()   
 {…}

IEnumerator IEnumerable.GetEnumerator()   
 {

IEnumerable<T> enumerable = this;

return enumerable.GetEnumerator();   
 }

}

1. Prefer generic methods to entire generic classes.
2. Always throw semantically useful exceptions from public methods.
3. Avoid parameters that control method flow; ensure each parameter is necessary for computation.

## Golden Rules

1. Favor Convention over Configuration.
2. Favor Repeatable, Reusable Patterns
3. Follow the Principal of Least Astonishment.
4. Automated tests are Good.
5. Share knowledge.

Suggested Reading

### Refactoring

* Refactor mercilessly: <http://www.extremeprogramming.org/rules/refactor.html>
* Smells to refactorings: <http://wiki.java.net/bin/view/People/SmellsToRefactorings>
* Index of refactorings: <http://www.refactoring.com/catalog/index.html>

### Patterns

* Anti-Patterns: <http://en.wikipedia.org/wiki/Anti-pattern#Programming_anti-patterns>
* Design Patterns: <http://www.dofactory.com/Patterns/Patterns.aspx>

### Pragmatic practices

* Ten Commandments of Egoless Programming:  
  <http://articles.techrepublic.com.com/5100-10878_11-1045782.html>
* Convention Over Configuration: <http://en.wikipedia.org/wiki/Convention_over_Configuration>
* Don't Repeat Yourself: [http://en.wikipedia.org/wiki/Don%27t\_repeat\_yourself](http://en.wikipedia.org/wiki/Don't_repeat_yourself)
* You ain’t gonna need it: [http://en.wikipedia.org/wiki/You\_Ain%27t\_Gonna\_Need\_It](http://en.wikipedia.org/wiki/You_Ain't_Gonna_Need_It)
* The Principal of Least Astonishment: <http://andywibbels.com/2006/10/the-principle-of-least-astonishment/>
* When to optimize: <http://en.wikipedia.org/wiki/Optimization_(computer_science)#When_to_optimize>
* The Art of Separation of Concerns: <http://ctrl-shift-b.com/2008/01/art-of-separation-of-concerns.html>
* Now you have two problems: <http://blogs.msdn.com/oldnewthing/archive/2006/03/22/558007.aspx>

### Language analysis

* Cyclomatic complexity: <http://javaboutique.internet.com/tutorials/metrics/>
* The difference between override and new: <http://blogs.msdn.com/csharpfaq/archive/2004/03/12/88453.aspx>
* Cleaner, more elegant, and wrong: <http://blogs.msdn.com/oldnewthing/archive/2004/04/22/118161.aspx>

# JavaScript Coding Standards

## JavaScript Best Practices

### Always Use 'var'

### Feature-Detect Rather Than Browser-Detect

### Use Square Bracket Notation

### Avoid 'eval'

### Avoid 'with' Statements

### Avoid Cluttering The Global Namespace

### Avoid sync "Ajax" calls

### Use JSON

### Use Correct <script> Tags

## JQuery Best Practices

* jQuery Variables
* DOM Manipulation
* Events
* Page Style and Layout Changes
* Effects and Animation
* Selectors
* Plugins
* Chaining

## Component Development Best Practices

## Client Template Best Practices

## Useful Tools

* jsHint
* JSLint Visual Studio Plugin (supports jsHint engine)

## Suggested Reading

### References

### Books

* Javascript: The Good Parts - <http://shop.oreilly.com/product/9780596517748.do>

# CSS Coding Standards

## TBD – Ron and team

# ASP.NET MVC Best Practices

### Inherit from Important base classes (DowJonesHttpApplication, ControllerBase, etc.)

Some key base classes in the framework offer powerful and relatively universal functionality. In some cases, many parts of the Framework will simply not work unless you inherit from these classes. Two of the most important base classes are *DowJones.Web.Mvc.****DowJonesHttpApplication*** and *DowJones.Web.Mvc.Infrastructure.****ControllerBase.***

### Depend on abstractions

Abstractions encourage loosely-coupled systems with a healthy separation of contracts and implementations. Abstractions are easily interchanged which not only provides easier maintenance, but is also crucial to unit testing.

#### Avoid the New Keyword

Any time you employ the **new** keyword to create a new instance of a concrete type you are – by definition – *not* depending on an abstraction. Though this is often not a problem at all (e.g. new StringBuilder(), new List<T>(), etc.), take a moment any time you use the **new** keyword to consider if the object you are creating might be better expressed as a dependency to be injected. Let another component create it!

#### Avoid referring to HttpContext directly (use HttpContextBase)

ASP.NET MVC (and later, .NET 4) introduced System.Web.Abstractions, a set of abstractions over many of the core parts of the ASP.NET framework. The “depend on abstractions” advice extends to these classes as well. In particular, one of the most often referenced objects in ASP.NET development is HttpContext – prefer using the HttpContextBase abstraction instead.

### Avoid “magic strings”

“Magic strings” – crucial, yet arbitrary string values – may be convenient and in many situations even required, however they have many issues. Some of the biggest issues with magic strings are that they:

1. don’t have any intrinsic meaning (e.g. it’s difficult to tell how or if one “ID” relates to another “ID”)
2. are easily broken with misspelling or case sensitivity
3. don’t react well to refactoring
4. promote rampant, pervasive duplication

Here are two examples, the first using magic strings to access data in a ViewData dictionary, and the second refactored example with that same data in a strongly-typed model:

**Using magic strings:**

<p>

<label for="FirstName">First Name:</label>

<span id="FirstName">@ViewData["FirstName"]</span>

</p>

**Using a strongly-typed model:**

<p>

<label for="FirstName">First Name:</label>

<span id="FirstName">@Model.FirstName</span>

</p>

Magic strings carry the allure of being very simple to use when you introduce them, but that ease of use often comes back to bite you later when it comes time to maintain them.

#### Prefer View.Model over View.ViewData

As the preceding example shows, the ViewData dictionary is one of the most tempting places to leverage magic strings in an ASP.NET MVC application. However, strongly-typed Presentation Models can be a handy tool to avoid assigning and retrieving data directly to and from the ViewData dictionary.

### Do not write HTML in “back end” code

Follow the practice of separation of concerns: it is not the responsibility of controllers and other “back-end code” to render HTML.

The exceptions here, of course, are UI helper methods and classes whose only job is to help the views render code. These classes should be considered part of the view, not “back-end” classes.

### Do not perform business logic in views

The inverse of the previous practice is true as well: views should not contain any business logic. In fact, views should contain as little logic as possible! Views should concentrate on how to display data that they have been provided, not take action on that data.

### Consolidate commonly-Used View Snippets with Helper Methods

The notion of “user controls,” “server controls,” and simply “controls” in general is very widespread… and for good reason. These concepts help consolidate commonly-used code and logic in a central location to make it easier to reuse and maintain. ASP.NET MVC is not control-driven, however – instead, it relies on the “helper method” paradigm in which methods do the work that controls once did. This can pertain to an entire section of HTML (what we’re used to calling a “control”), or even as simple as strongly-typed access to a commonly-referred URL.

For example, you may notice many of the same references to the “Membership Page” (~/membership) like so:

@Html.ActionLink(“Membership”, “Index”, “Membership”, […])

You can instead consolidate this call (and eliminate the [magic strings](#_Avoid_“magic_strings”)!) by turning it into a helper method:

@Html.MembershipLink()

### prefer Presentation Models Over Direct Usage of business objects

In general, try to avoid allowing changes to the business model to directly affect the view. [Presentation Models](#_Presentation_Models) help with this.

### Encapsulate View “if” statements with Html Helpers

Integrating code and markup is quite powerful; however, it can get quite messy. Consider the following (relatively simple) if-else statement:

**Index.cshtml**

@if(Model.IsAnonymousUser) {

<img src="@Url.Content("~/content/images/anonymous.jpg")" />

} else if(Model.IsAdministrator) {

<img src="@Url.Content("~/content/images/administrator.jpg")" />

} else if(Model.Membership == Membership.Standard) {

<img src="@Url.Content("~/content/images/member.jpg")" />

} else if(Model.Membership == Membership.Preferred) {

<img src="@Url.Content("~/content/images/preferred\_member.jpg")" />

}

That’s quite obscure code for rendering out essentially the same markup with the exception of one part (the URL). Consider this approach instead:

**UserHtmlHelperExtensions.cs**

public static string UserAvatar(this HtmlHelper<User> helper)

{

var user = helper.ViewData.Model;

string avatarFilename = "anonymous.jpg";

if (user.IsAnonymousUser)

{

avatarFilename = "anonymous.jpg";

}

else if (user.IsAdministrator)

{

avatarFilename = "administrator.jpg";

}

else if (user.Membership == Membership.Standard)

{

avatarFilename = "member.jpg";

}

else if (user.Membership == Membership.Preferred)

{

avatarFilename = "preferred\_member.jpg";

}

var urlHelper = new UrlHelper(helper.ViewContext.RequestContext);

var contentPath = string.Format("~/content/images/{0}", avatarFilename);

string imageUrl = urlHelper.Content(contentPath);

return string.Format("<img src='{0}' />", imageUrl);

}

**Index.cshtml** (and everywhere else you need the user’s avatar)

@Html.UserAvatar()

Not only is this cleaner, it’s also more declarative and moves this logic into a central location so that it may be more easily maintained. For instance, if the requirements change and the site needs to support custom avatars, the Html.UserAvatar helper method can be modified in one place.

### Use Script Registry and Stylesheet Registry instead of direct includes (even for snippets)

The Dow Jones MVC Framework introduces the Script Registry and Stylesheet Registry concepts to better manage script and style includes. These Registries provide you with a central location to manage these resources, as well as “behind the scenes” optimizations such as script combining, minifying, etc. To best take advantage of these optimizations, leverage these Registries as opposed to implementing these things yourself.

### Prefer explicit View names

A majority of the ASP.NET MVC controller action code samples call the View() method without specifying a view name. This is suitable for simple demo code, however when tests or other action methods begin calling each other, the detriments to this approach become clear. When no view name is specified, the ASP.NET MVC framework defaults to the name of the action that was *originally* *called*. Thus, calling the Index action in the following example will attempt to locate a view named “Index.cshtml” – a view that probably doesn’t exist (but “List.cshtml” certainly does!):

**EmployeeController.cs**

public ActionResult Index()

{

return List();

}

public ActionResult List()

{

var employees = Employee.GetAll();

return View(employees);

}

If the List action is modified to call the View() method with a specific view name (as shown below), everything works fine.

public ActionResult List( )

{

var employees = Employee.GetAll();

return View(**"List",** employees);

}

### Prefer objects over long lists of parameters

This advice is not specific to ASP.NET MVC - long parameter lists are commonly considered a “code smell” and should be avoided whenever possible. Additionally, ASP.NET MVC’s powerful Model Binders make following this advice incredibly easy. Consider the two contrasting snippets:

**Long Parameter List**

public ActionResult Create(

string firstName, string lastName, DateTime? birthday,

string addressLine1, string addressLine2,

string city, string region, string regionCode, string country

[... and many, many more]

)

{

var employee = new Employee( [Long list of parameters...] )

employee.Save();

return View("Details", employee);

}

**Parameter Object**

public ActionResult Create(Employee employee)

{

employee.Save();

return View("Details", employee);

}

The Parameter Object example is much more straight-forward, and leverages the ASP.NET MVC framework’s powerful Model Binders and model validation to make this code much safer and easier to maintain.

### Encapsulate shared/common functionality, logic, and data with Action Filters or Child Actions (Html.RenderAction)

Every website of any significant complexity will have common elements across multiple (or perhaps all) pages in the application. A global website navigation menu – the kind that appears on every single page in the site - is a canonical example of this type of globally-applied logic and content. The data for these common elements needs to come from *somewhere*, yet explicitly retrieving the data in every controller action would create a maintenance nightmare. Action Filters and/or child actions (via the **Html.RenderAction** method) provide a central location to hold this kind of logic.

Consider the following layout snippet (cut from the larger layout page) which renders navigation items in a list:

<ul id="global-menu">

@foreach (var menuItem in ViewData.SingleOrDefault<NavigationMenu>()) {

<li class="@(menuItem.IsSelected ? "selected" : null)">

@Html.RouteLink(menuItem.DisplayName, menuItem.RouteData)

</li>

}

</ul>

The **NavigationMenu** **ViewData** object needs to come from *somewhere*. Since they can be configured to execute prior to every controller request, Action Filters make an excellent candidate to populate View Data with globally-required data like this. Here is the Action Filter that populates the **NavigationMenu** data required in the previous example:

public class NavigationMenuPopulationFilter : ActionFilterAttribute

{

private readonly INavigationDataSource \_dataSource;

public NavigationMenuPopulationFilter(INavigationDataSource dataSource)

{

\_dataSource = dataSource;

}

public override void OnActionExecuting(ActionExecutingContext filterContext)

{

NavigationMenu mainMenu = \_dataSource.GetNavigationMenu("factiva-main-menu");

filterContext.Controller.ViewData["MainNavigationMenu"] = mainMenu;

}

}

This Filter is pretty straight-forward – it gets the correct navigation menu data model from some data source and adds it to the View Data collection prior to executing the requested action. From this point on, any component that requires it can retrieve the navigation menu from View Data. As with most Action Filters, there are several ways to apply the **NavigationMenuPopulationFilter**: either apply the attribute to your controllers or actions, or register the filter in the Global Action Filters collection. Below is a bootstrapper task for adding the Action Filter to the list of global filters:

public class GlobalFilterRegistrationTask : IBootstrapperTask

{

[Inject]

public NavigationMenuPopulationFilter NavigationFilter { get; set; }

public void Execute()

{

GlobalFilters.Filters.Add(NavigationFilter, 4);

}

}

### Prefer placing Action Filters at the “highest appropriate level”

Most Action Filter attributes can be applied at either the method (Action) or class (Controller) level. When an attribute applies to all actions in a controller, prefer placing that attribute on the controller itself rather than on each individual class. Also consider whether or not the attribute may be appropriate further up the controller’s dependency chain (i.e. on one of its base classes) instead.

### Prefer partial views (or entirely separate views) over complex if-then-else logic that shows and hides sections

The Page Controller pattern of Web Forms encourages posting back to the same page, possibly showing or hiding certain sections of the page depending on the request. Due to ASP.NET MVC’s separation of concerns, this can often be avoided by creating separate views for each of these situations, lowering or eliminating entirely the need for complex view logic. Consider the following example:

**Wizard.cshtml**

@if(Model.WizardStep == WizardStep.First) {

<!-- The first step of the wizard -->

} else if(Model.WizardStep == WizardStep.Second) {

<!-- The second step of the wizard -->

} else if(Model.WizardStep == WizardStep.Third) {

<!-- The third step of the wizard -->

}

Here the view is deciding which step of the Wizard to display, which is dangerously close to business logic! Let’s move this logic to the Controller where it belongs and split this view into multiple views:

**WizardController.cs**

public ActionResult Step(WizardStep currentStep)

{

// This is simple logic, but could be MUCH more complex!

string view = currentStep.ToString();

return View(view);

}

**First.cshtml**

<!-- The first step of the wizard -->

**Second.cshtml**

<!-- The second step of the wizard -->

**Third.cshtml**

<!-- The third step of the wizard -->

### Prefer the Post-Redirect-Get pattern When Posting Form Data

The [Post/Redirect/Get (PRG)](http://en.wikipedia.org/wiki/Post/Redirect/Get)[[1]](#footnote-1) pattern is a common design pattern for web developers to help avoid certain duplicate form submissions and allow user agents to behave more intuitively with bookmarks and the refresh button. Due to the Page Controller nature of Web Forms in which developers are usually required to post back to the same page for all actions in a particular context (e.g. display employee data so that it may be edited and re-submitted), the PRG pattern is not used as much in Web Forms environments.

Because ASP.NET MVC separates actions into separate URLs it is easy to run into trouble with update scenarios. For instance,

**EmployeeController.cs**

public class EmployeeController : Controller

{

public ActionResult Edit(int id)

{

var employee = Employee.Get(id);

return View("Edit ", employee);

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Update(int id)

{

var employee = Employee.Get(id);

UpdateModel(employee);

return View(“Edit”, id);

}

}

In this example when a user posts to the Update action, though the user will be looking at the “Edit” view as desired, the resulting URL in their browser will be “/employees/**update**/1”. If the user refreshes the page or bookmarks a link to that URL, etc. subsequent visits would update the employee information again or even not work at all.

What we really want to happen in the Update action is to update the Employee information and then redirect the user back to the Edit page so that they are back to the original “Edit” location. In this scenario, the PRG pattern may be applied thusly:

**EmployeeController.cs** (partial, showing only the changed Update action)

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Update(int id)

{

var employee = Employee.Get(id);

UpdateModel(employee);

return RedirectToAction("Edit", new { id });

}

Though it’s a subtle change, switching from the View() method to the RedirectToAction() method will produce a client-side redirect (as opposed to a “server-side redirect” in the original example) after the Update method has finished updating the employee, landing the user on the proper URL: “/employees/edit/1”.

### Prefer Bootstrapper tasks over logic placed in Application\_Start (Global.asax)

Most ASP.NET MVC demos will advise modifying the Application\_Start method in the Global.asax file in order to introduce logic that will execute when the application starts. While this is certainly the easiest and most straight-forward approach, the Dow Jones MVC Framework provides the concept of “bootstrapper tasks”. These tasks are easy to implement (simply implement the IBootstrapperTask interface – the Execute() method) and are automatically discovered (merely by existing in referenced assembly) and executed during the Application\_Start phase of the application. These Tasks help provide cleaner code and encourage proper adherence of the Single Responsibility principle.

### Prefer Authorize attribute over imperative security checks

Traditionally, authorization control resembles the following:

public ActionResult Details(int id)

{

if (!User.IsInRole("EmployeeViewer"))

return new HttpUnauthorizedResult();

This is a very imperative approach, and makes it difficult to implement application-wide changes. The ASP.NET MVC AuthorizeAttribute provides a simple and declarative way to authorize access to actions. This same code may be rewritten as:

[Authorize(Roles = "EmployeeViewer")]

public ActionResult Details(int id)

{

### Prefer creating a custom layout for mobile devices

Mobile device browsers operate under a completely different environment than desktop browsers. It is difficult enough to worry about the intricacies of various desktop browsers, let alone adding mobile support to the mix. Instead of flooding your “default” (desktop-focused) views with mobile-conditional logic, separate the mobile views into their own files. In fact, the Dow Jones MVC Framework even offers support for targeting specific mobile devices. Take advantage of these capabilities whenever possible!

### Prefer using the Route attribute over More Generic Global Routes

Routing is one of the most powerful aspects of ASP.NET 4.0. The default route that ships with the ASP.NET MVC template (“{controller}/{action}/{id}”) is derived from what is considered the most common routing pattern for “standard” MVC applications. This pattern is not universal, however, and is commonly augmented or replaced with additional custom routes. When determining new routes for an application, the more specific you can be the better. Otherwise, your application may experience matching on unintended routes.

Of course, the most specific route is one that maps directly to one action and one action only. The Dow Jones MVC Framework provides you with this ability via [Custom Routing](#_Custom_Routes) (with the RouteAttribute), allowing you to specify a custom route for a particular action. Prefer applying this attribute when you desire non-standard routing logic over adding a more generic custom global route.

### Consider using an Anti-Forgery Token to avoid CSRF attacks

For form posts where security is a concern, ASP.NET MVC provides measures to help deter certain kinds of common attacks. One of these measures is the Anti-forgery Token. The Token has both server- and client-side components.

**Client Side:** Simply call the Html.AntiForgeryToken helper method at some point inside your form

@using(Html.Form("Update", "Employee”)) {

@Html.AntiForgeryToken()

<!-- rest of form goes here -->

}

**Server Side:** Apply the ValidateAntiForgeryTokenAttribute to the destination action (the action that is receiving the form post data)

[ValidateAntiForgeryToken]

[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Put)]

public ActionResult Update(int id)

{

This code will insert a user-specific token in a hidden field on your form and validate that token on the server side prior to executing any further processing of the data being posted.

For more detailed information, please see the blog posts by [Phil Haack](http://haacked.com/archive/2009/04/02/anatomy-of-csrf-attack.aspx)[[2]](#footnote-2) and [Steve Sanderson](http://blog.stevensanderson.com/2008/09/01/prevent-cross-site-request-forgery-csrf-using-aspnet-mvcs-antiforgerytoken-helper/)[[3]](#footnote-3)

### Consider the AcceptVerbs attribute to restrict how actions may be called

Many Actions rest on a number of assumptions about how and when they will be called in the context of an application. For instance, one assumption might be that an Employee.Update action will be called from some kind of Employee Edit page containing a form with the Employee properties to post to the Employee.Update action in order to update an Employee record. If this action is called in an unexpected way (e.g. via a GET request with no form posts), the action will probably not work, and in fact may produce unforeseen problems.

The ASP.NET MVC framework offers the AcceptVerbs attribute to help restrict action calls to specific Http Methods. Thus, the answer to the aforementioned Employee.Update scenario would be:

[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Put)]

public ActionResult Update(int id)

Applying the AcceptVerbs attribute in this way will restrict requests to this action only to those made specifying the POST or PUT Http Methods. All others (e.g. GET requests) will be ignored.

### Consider output caching

Output caching is one of the easiest ways to get additional performance from a web application. Caching requests in which little or no content has changed since the previous request is a quick way to speed up your request times. The ASP.NET MVC framework offers the OutputCacheAttribute to accomplish this task. This attribute mirrors the Web Forms output caching functionality and accepts many of the same properties.

### Consider custom ActionResults for unique scenarios

The ASP.NET MVC Request Pipeline has a deliberate separation of concerns in which each step in the process completes its task and no more. Each step does merely enough to provide the subsequent tasks with enough information to do what they need to do. For instance, a controller action that decides a view should be rendered to the client does not load up a view engine and order it to execute the view, it merely returns a ViewResult object with the information that the Framework needs to take the next steps (most likely loading a view engine and executing the view!).

When it comes to results of controller actions, declarative is the name of the game. For instance, the ASP.NET MVC Framework provides an HttpStatusCodeResult with a StatusCode property, but it also goes one step further to define a custom HttpStatusCodeResult named HttpUnauthorizedResult. Though the following two lines are effectively the same, the latter provides a more declarative and strongly-typed expression of the controller’s intent.

return new HttpStatusCodeResult(HttpStatusCode.Unauthorized);

return new HttpUnauthorizedResult();

When your actions produce results that don’t fit the “normal results”take a moment to consider whether returning a custom Action Result may be more appropriate. Some common examples include things like RSS feeds, Word documents, Excel spreadsheets, etc.

### Consider Async Controllers for controller tasks that can happen in parallel

Parallel execution of multiple tasks can offer significant opportunities to enhance the performance of your site. To this end, ASP.NET MVC offers the **AsyncController** base class to help make processing parallelizable requests easier. When creating an action with processor-intensive logic, consider whether that action has any elements that may be safely run in parallel.

See the [ASP.NET MVC Asynchronous Controller documentation](http://msdn.microsoft.com/en-us/library/ee728598.aspx)[[4]](#footnote-4) for more information.

1. Post/Redirect/Get; Wikipedia <http://en.wikipedia.org/wiki/Post/Redirect/Get> [↑](#footnote-ref-1)
2. Anatomy of a Cross-site Request Forgery Attack; Phil Haack  
    <http://haacked.com/archive/2009/04/02/anatomy-of-csrf-attack.aspx> [↑](#footnote-ref-2)
3. Prevent Cross-Site Request Forgery (CSRF) using ASP.NET MVC’s AntiForgeryToken() helper; Steve Sanderson  
    <http://blog.stevensanderson.com/2008/09/01/prevent-cross-site-request-forgery-csrf-using-aspnet-mvcs-antiforgerytoken-helper/> [↑](#footnote-ref-3)
4. Using an Asynchronous Controller in ASP.NET MVC; MSDN  
    <http://msdn.microsoft.com/en-us/library/ee728598.aspx> [↑](#footnote-ref-4)