***Best Practices for ASP.NET MVC***

# Model Recommendations

The model is where the domain-specific objects are defined. These definitions should include business logic (how objects behave and relate), validation logic (what is a valid value for a given object), data logic (how data objects are persisted) and session logic (tracking user state for the application).

## **DO separate the model its own project with a distinct assembly.**

For applications with a large complex model, it's a good idea to create a separate assembly for the model to avoid accidently mixing concerns. You can then reference the model assembly in your ASP.NET MVC project.

## **DO put all business logic in the model.**

If you put all business logic in the model, you shield the view and controller from making business decisions concerning data. You also reap the following benefits:

* Less duplicated business logic.
* The view is easier to read when there is no business logic present.
* Testing business rules is isolated to the model.

For example, if you have a business requirement to display a user's name with the last name first, you could put the logic in the view as follows:

<% if (String.Compare((string)TempData["displayLastNameFirst"], "on") == 0)

{ %>

Welcome, <%= Model.lastName%>, <%= Model.firstName%>

<% }

else

{ %>

Welcome, <%= Model.firstName%> <%= Model.lastName%>

<% } %>

However, you would have to duplicate this logic in every place this business requirement was needed. Instead, you could put the business logic the the "display last name first" rule in the model by adding a property to the model that encapsulates the business logic as follows:

public string combinedName

{

get

{

return (displayLastNameFirst ? lastName + " " + firstName : firstName + " " + lastName);

}

private set

{

;

}

}

This would greatly simplify the view as shown:

<% Welcome, <%= Model.combinedName %> %>

## **DO put all validation logic in the model.**

All input validation should occur in the model layer. This includes client side validation, which is essential to performance. However, client side validation can be circumvented (with, for example, tools like Fiddler).

You can use ModelState to add validation checking. The following example shows how to add validation checks to ModelState explicitly:

if (String.IsNullOrEmpty(userName))

{

ModelState.AddModelError("username", Resources.SignUp.UserNameError);

}

However, given the advances in .NET Framework, the **System.ComponentModel.DataAnnotations** should be the preferred method for validation. These annotations are added as attributes to the properties of a model class, as the following example shows:

public class User

{

[Required(ErrorMessageResourceName = "nameRequired", ErrorMessageResourceType = typeof(Resources.User))]

public String userName { get; set; }

...

}

## **DO define interfaces for data access**

.

It is preferred that interfaces be used to expose methods on a provider for data access. This reinforces the loosely coupled component design of ASP.NET MVC.

Consider using the Entity Framework or LINQ to SQL as the means of creating wrappers around calls to a database. Both Entity Framework and LINQ to SQL allow the use of stored procedures as well.

## **DO put all session logic in the model.**

It is beyond the scope of this document to explore in depth the various mechanisms for storing session state in the model. As a starting point, here are a few of possibilities of session state storage:

|  |  |  |
| --- | --- | --- |
| **Technique** | **Strengths** | **Weaknesses** |
| In Process | No additional setup required. | Does not work if the web site needs to scale. |
| Session State Service | Lightweight sertvice runs on each machine in a web farm.  Faster than database session storage. | Session data is lost if the service goes down. |
| Database | Session data is persisted. | Slower than session state.  Management cost is relatively high. |

# View Recommendations

The primary concern of a view is presentation of the model. The view is chosen by the controller. Business logic does not belong in the view, because business logic is the concern of the model layer. The view mechanism can be extremely flexible. For example, a view of the model for a web page can be rendered using HTML. Another view of the same model (same data), can be presented in XML and act as a web service.

## **DO put HTML in Views and Partial Views (not in a controller).**

A strength of the view pattern is the readability of view template files. For the default view engine, ASP.NET offers the following types of view files: full HTML view (.aspx), partial HTML view (.ascx), and master page (.master). A master page enables you to specify an overall layout for a view. Master pages can be nested several times to create a hierarchy of available layout types.

The following example, shows a view which calls a partial view:

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage" %>

…

Below is a list of items submitted by <b>

<%= Html.Encode(ViewData["name"]) %></b>.

<p>

...

<div id="items">

<% Html.RenderPartial("ItemsByName");%>

</div>

</asp:content>

The partial view (ItemsByName.ascx) is shown here:

<%@ Control Language="C#" %>

…

<% foreach (Seller.Controllers.Items item in (IEnumerable)ViewData.Model)

{ %>

<tr>

<td>

<%= Html.Encode(item.title)%>

</td>

<td>

<%= Html.Encode(item.price)%>

</td>

</tr>

<% } %>

</table>

<% } %>

The partial view is a powerful extensibility and reuse mechanism. For example, we can include the same view in an admin view, without writing another line of code.

## **DO access data in views using ViewData.**

ASP.NET provides the following mechanisms to enable you to access data from view templates:

* **ViewData.Model** object, which is setup in a controller’s action method by passing a model object in the action method's return statement (return View(myModelObject)).
* **ViewData** dictionary (property bag), which enables you to enter any data into the dictionary  in an action method (ViewData[“key”] = value), and then access that same dictionary from within the view.

Whenever possible, you should use **ViewData Model** instead of the **ViewData** dictionary because it provides better type safety. Additionally, you should use either data access mechanism rather than accessing the Request/Session state directly in the view template.

If you have an object for which you are going to display multiple properties, you should use **ViewData.Model** and create a strongly typed view for that object type. If you have a seller’s details page, for example, and the seller class has the name, phone, address, email, etc. properties. Then you would assign a seller instance to **ViewData.Model** in the controller before rendering the view. If you have disparate data such as page #, a user’s name, and the current time, use **ViewData** dictionary.

Avoid data access in the view when using model binding. In other words, do the data retrieval from the database in the controller. Then populate lightweight view model objects from the retrieved data in the controller before executing the view. So the lightweight view model objects do not retrieve data during view execution.

## **DO enable (auto-generated) client side validation.**

Previously, web developers were faced with the dilemma of keeping client and server validation synchronized. Starting with ASP.NET MVC 2, it has become easy to add client side validation.

To add client side validation:

1. Add data validation in the model layer as described in the section titled DO put all business login in the model.
2. Ensure the following javascript files are in Scripts directory in your project:
   * MicrosoftAjax.js
   * MicrosoftMvcValidation.js
3. Add the following lines to your form submission page:
4. <script src="<%= Url.Content("~/Scripts/MicrosoftAjax.js") %>" type="text/javascript"></script>

<script src="<%= Url.Content("~/Scripts/MicrosoftMvcValidation.js") %>" type="text/javascript"></script>

1. Put the following line before the form tag:

<% Html.EnableClientValidation(); %>

Now if you try to edit a form field which does not match the data annotation, the client side validation will fail and immediate feedback is given to the user.

## **DO insert server-side comments in templates.**

Use server side comments in the view templates for commenting. These comments are stripped out before the server returns the HTML representation.

The following line demonstrates a server side comment:

<%-- This is a server side template comment --%>

Do not use HTML comments in the view template because they will be rendered to the web browser and could be viewed by an advanced (and potentially malicious) user.

## **DO use HTMLHelper extension methods.**

The **System.Web.Mvc.Html** class contains useful HTML extension methods. These extension methods include helpers for:

* Form generation (BeginForm)
* Input field generation (checkbox, hidden, radio button, textbox)
* Link generation (ActionLink)
* XSS protection (Encode)

These HTML helpers should be utilized as much as possible. For example, the following code creates a link using the route table back to the default action on the controller from which the view is called.

<%= Html.ActionLink(“Home page”, “Default”) %>

# Controller Recommendations

The controller (and a specified action method) is invoked by the routing system given a pattern match on the URL. The controller receives input from the routing system, which includes the HTTP request context (session, cookies, browser, etc).

## **DO use model binding instead of manually parsing the request.**

ASP.NET MVC abstracts much of the rote code of object deserialization by using model binding. Model binding is a mechanism by which request context data is marshaled through reflection into the object type defined on the action method.

The following example shows a **Seller** class that defines the data that might be submitted from a form for signing up sellers:

public class Seller

{

public Int64 ID { get; set; }

public string Name { get; set; }

public string Phone { get; set; }

public string Address { get; set; }

}

The form form submitting the sellers data could be contained in a Register view with code similar to the following:

<% using (Html.BeginForm()) { %>

<legend>Account Information</legend>

<p>

<%= Html.TextBox("Name") %>

</p>

<p>

<%= Html.TextBox("Phone") %>

</p>

<p>

<%= Html.TextBox("Address") %>

</p>

<p>

<input value="Register" type="submit" />

</p>

<% } %>

The controller would need a Register action method that would provide the model binding as shown:

public ActionResult Register([Bind(Exclude="ID")] Seller newSeller)

{

...

}

The default model binder will look for each property in the class given the following order (using Name as example):

1. Request.Form["Name"], if it exists
2. RouteData.Values["Name"], if it exists
3. Request.QueryString["Namel"], if it exists
4. Otherwise, null

As you can see from the Register action method, there are several attributes which can be placed on an object that will be invoked using the default model binder.

The model binding system also runs the validation logic that has been applied to the model object, such as the data annotations attributes.

The model binding system has a rich extensibility mechanism that allows full customization of how objects are created, populated, and validated.

## **DO explicitly name views in action methods.**

After you have setup the context in the action method for HTML generation, you will return a **ViewResult** or a **PartialViewResult** object. If you do not pass a view name to the result class, the view file will be chosen based upon the receiving action name. For example, given a controller named Products with an action named List. You can call “return View()” from within the List action method without any parameters. The framework will look for a view called /Views/Products/List.aspx. If that view is missing, it will try /Views/Products/List.ascx. If that is not present, it tries /Views/Shared/List.aspx and then /Views/Shared/List.ascx. So, you can use /Views/Shared for any views that are shared across multiple controllers.

To avoid confusion, explicitly name the view, such as "return View("*explicitViewName")",*in the action method. This allows you to call List from a different action, without the framework looking for a different view.

## **DO use Post/Redirect/Get (PRG) when submitting forms.**

According to the definition of the HTTP POST and GET verbs:

* HTTP GET is used for non-changing (idempotent) data to your model.
* HTTP POST is used for changing data to your model.

Given this clear delineation, when receiving form data in your post back action method, return RedirectToAction(<actionName>), which will result in a HTTP 302 (temporary redirect) and will generate a GET on the <actionName>. This results in Post-Redirect-Get pattern.

Therefore, do not use HTTP GET for posting form data, as this is a violation of the purpose of HTTP GET verb.

Additionally, classic ASP.NET postbacks can be the cause of a problematic feedback loop when posting forms.

For example, in the diagram below using a standard postback, you do a GET and POST against the same url (create.aspx). This is a problem when a user of the site gets impatient waiting on the form post to complete. If they hit the browser refresh button, there is the potential of submitting duplicate data which your web site will have to handle. You can solve this problem in MVC using the Post-Redirect-Get pattern.

However, this pattern does come with client side performance penalty, because the redirect causes further requests to the server. This performance cost has to weighed against the usability benefits of this pattern during the decision making process.

## **DO implement HandleUnknowAction and HandleError.**

The default response for an unknown action is 404 (Not Found) error. If you override the **HandleUnknownAction** class in a controller, you can implement a “default” view in this error. Additionally, put the **HandleError** attribute on an action and/or controller and you can provide a standard error view for when an uncaught exception is thrown.

# Routing Recommendations

Routing is used in ASP.NET MVC to map URLs directly to a controller, rather than a specific file. This is especially useful for readability, as the developer can focus on designing URLs that are human readable (for example, product support and search engine indexing).

The default routes are added to a RouteTable which is located inside of Application\_Start section of the Global.asax file. The table allows you to map a specific URL to a controller and action.

## **DO order routes from specific to general when using standard routing.**

The route table is ordered, therefore create routes from most specific to general.

Consider the following example. Suppose you have a product catalog for which you would like to create URLs of the following forms:

* http://sellmyproducts/
* http://sellmyproducts/Page#
* http://sellmyproducts/category
* http://sellmyproducts/category/Page#

Given the following signature of the List method (on ProductsController class):

public ViewResult List(string category, int page)

The following routing specification will correctly route the user to the correct views, given the previously specified schema:

routes.MapRoute(

null,

"",

new { controller = "Products", action = "List", category = (string)null, page = 1 }

);

routes.MapRoute(

null,

"Page{page}",

new { controller = "Products", action = "List", category = (string)null },

new { page = @"\d+" }

);

routes.MapRoute(

null,

"{category}",

new { controller = "Products", action = "List", page = 1}

);

routes.MapRoute(

null,

"{category}/Page{page}",

new { controller = "Products", action = "List"},

new { page = @"\d+" }

);

## **DO use named route mechanism to avoid route ambiguity.**

When you rely upon the ASP.NET routing mechanisms, you must know how the routing mechanisms work. Otherwise, you can create a lot of extra work tracking down misbehaving routes. One way to mitigate this problem is to explicitly name the routes.

For example, the following route mapping define named routes:

routes.MapRoute(

"Default",

"",

new { controller = "Products", action = "List", category = (string)null, page = 1 }

);

routes.MapRoute(

"PageRoute",

"Page{page}",

new { controller = "Products", action = "List", category = (string)null },

new { page = @"\d+" }

);

Using these route definitions, you can create a link which resolves to "PageRoute" as follows:

<%= Html.RouteLink("Next", "PageRoute",

new RouteValueDictionary( new { page = i + 1 } )); %>

# Extensibility Recommendations

Inside of the ASP.NET MVC framework there are many points for extension. You can replace any of the core components, a partial list of which includes the following:

* routing engine (MvcRouteHandler)
* controller factory (IControllerFactory)
* view engine (IViewEngine)

For example, you might want to write your own controller factory that uses an inversion of control container.

Rewriting core components is outside the scope of this topic. However, you could extend the framework by adding custom behaviors in the form of filters. Some of the standard filters included in the framework are: OutputCache, HandleError, and Authorize.

## **DO use filters for adding behaviors**

MVC has a mechanism to add behaviors (filters) through the use of attributes before and after action methods and action results. These filters allow for extensibility which is lightweight in the request handling pipeline.

Filters can be applied to a specific action method to alter its behavior. Alternatively, a filter can be applied to a controller class, in which case it will take effect on every action method in the controller class. Base classes can be used to define common behavior patterns by applying filters to the base controller class and then ensuring that other controllers derive from that base class.

For example, suppose that you want to add functionality to log information for each request to debug a problem with HTTP headers. The following code define class that derives from the **ActionFilterAttribute** class.

public class LogHeadersFilterAttribute : ActionFilterAttribute

{

public override void OnActionExecuting(ActionExecutingContext filterContext)

{

foreach (string header in

filterContext.HttpContext.Request.Headers.AllKeys)

{

Debug.WriteLine("Header " + header);

Debug.WriteLine("Value " +

filterContext.HttpContext.Request.Headers.Get(header));

}

base.OnActionExecuting(filterContext);

}

}

In order to add this filter to a given action method, you simply place the **LogHeadersFilter** attribute at the top of the action (or controller) that you want to filter.

# Testability Recommendations

One of the major strengths of the MVC pattern is the improved testability of the designs by keeping concerns separated and decoupled. In ASP.NET MVC you can cleanly separate and test the business logic in the model. For example, you could test the logic for adding a bid to the auction site without depending upon either the controller or the view.

## **DO write units tests.**

ASP.NET MVC provides many of the tools developers need to create testable applications. In addition, it is relatively easy add a third-party unit test framework, mocking framework, or dependency injection framework. It is beyond the scope of this topic to tell you how to create unit tests for your application. However, ASP.NET MVC provides a flexible architecture that allows for easy testing. Unit testing is easier because of features like pluggable view engines, controller factories, action result types, and wrappers around the ASP.NET context types. For more information about unit testing an ASP.NET MVC application, see [Unit Testing in MVC Applications](http://msdn.microsoft.com/en-us/library/ff936235.aspx).

# Security Recommendations

Security is an important aspect of any modern software development project. Although no framework can provide perfect security, there is much you can do to help safeguard your ASP.NET MVC application.

## **DO guard against common attack vectors.**

Website security needs to concern all web developers writing enterprise class websites and services. There are a host of well known attack vectors that you should know about. These attack vectors include (but are not limited to):

* Cross-site scripting (XSS) attacks
* SQL injection
* Cross-site Request Forgery (XSRF)
* Improperly implementing model binding

To prevent cross-site scripting (XSS) attacks:

* Disable request validation through use of the **ValidateInput** attribute. This attribute will falsely reject valid HTML input.
* Add Html.Encode for all user input data that is displayed, whether immediately rendered or the data is put into the database and then displayed later.
* Set the **HttpOnly** flag on cookies. This will prevent JavaScript from reading and sending cookies.

To prevent SQL injection:

* Always use parameterized SQL queries.
* Do not pass raw SQL to the database.
* Use an object-relational mapper (ORM) such as Entity Framework, which can completely eliminate the need to have SQL statements in the application code.

To prevent cross-site request forgery (XSRF):

* Use the **Html.AntiForgeryToken** class in forms to protect against cross-site forgery request. On the action which takes the post request, place the **ValidateAntiForgeryToken** attribute

To properly implement model binding:

* Explicitly call out with which members of a class you're model binding to, use the [Bind(Include=explicit members here)] attribute above the class declaration. For example, if there is an **IsAdmin** field on a class, automatic binding without **Bind** directives would allow it to bind to this class.
* Alternatively, create so-called view models, which are model objects specifically designed to accept input from external sources. View models contain only properties that are safe for external input. After model binding has created, populated, and validated these objects, the property values of these objects can be mapped to the application model or data model for further processing.

## **DO authenticate and authorize users to protect content.**

It is beyond the scope of these guidelines to provide a deep treatment of authentication and authorization. However, you must annotate restricted data views, through either writing your own **RoleProvider**, or judicious use of the Authorize filter attribute.

## **DO use <%: %> (.NET 4) to protect against XSS attacks.**

Prior to .NET 4.0 the developer would have to ensure there HTML was encoded by using code like the following:

<%= Html.Encode(ViewData["name"]) %>

This code was need to protect against XSS (cross-site scripting) attacks.

If you are using .NET 4, do not use the above syntax. Use the following syntax instead.

<%: ViewData["name"] %>

This new syntax automatically HTML encodes the string (if necessary), and is preferred.

# Localization and Globalization Recommendations

Globalization is the process of making a product multi-lingual, where localization is the process of adapting a global product for a particular language and country. To develop a web application that supports globalization and localization, keep at least one rule in mind. Do not use hard-code strings in views.

## **DO use ASP.NET special resource folders and resource files.**

While writing your web pages add an ASP.NET project folder for globalized content (App\_GlobalResources) and  for localized content for a given view (App\_LocalResources). In each of these folders, you should add a  resource (.resx) file that you should name according to the controller name. In other words, if your controller is named SubmissionPipeline, the resource file should be named SubmissionPipeline.resx.

Visual Studio converts this text mapping class into a global class that you can call using the following syntax:

Resources.<resource\_filename>.<string\_name>

Then you would access the resource in the view like this:

<%= Resources.SubmissionPipeline.continueButton %>

When the translated resource files become available, you should name each file using the following format: <filename>.<language>.resx.

For example, the German version of the resource file would be named: SubmissionPipeline.de.resx.

# Performance Recommendations

Performance is a multi-faceted problem for web-sites, as a myriad of bottlenecks can affect performance including:

* Database
  + Inefficient queries
  + Incorrectly placed indexes
  + Non-normalized design
* Bandwidth problems
  + Large request size (affected by individual large images, .css, .js, .html, etc.)
  + Content that references many other items, such as multiple script, CSS, or image files
  + Slow connection speed
* Processing power
  + Server: expensive operations
  + Client: poorly written javascript

This section will focus solely on server processing and request size.

## **DO consider partial page updates using AJAX for reducing bandwidth.**

One way to mitigate performance issues involving both server processing and request size is to use asynchronous Javascript (AJAX) to do partial page updates. ASP.NET MVC has built-in AJAX support, which helps to facilitates this pattern. The performance benefit occurs because the pattern reduces the amount of processing that the server must do to render a request and reduce the size of the HTML fragment.

The following example explains how to to partial page updates using AJAX:

1. Choose what portion of HTML you want to update, and mark it with an ID.

<div id="items"> to be updated dynamically </div>

1. Add the Javascript files that enable AJAX to the HTML (typically in the master view):

<script src="<%= Url.Content("~/Scripts/MicrosoftAjax.js") %>" type="text/javascript"></script>

<script src="<%= Url.Content("~/Scripts/MicrosoftMvcAjax.js") %>" type="text/javascript"></script>

1. In the view you want to update, add an AJAX link that references an action method (in the example the action method is named RetrieveItems).

<%= Ajax.ActionLink("Refresh All", "RetrieveItems", new { criteria = "all" } , new AjaxOptions { UpdateTargetId = "items" })%>

1. Implement the controller action, and have it return the partial view.

## **DON'T overuse session, instead use TempData for short lived (intra-request) storage.**

It is tempting when creating websites to add objects to the **Session** object, so that they are readily available. The problem with placing these objects in the **Session** object is that it can bog down the server by having to store extra information that may only be necessary across a redirect. The correct way to store these temporary variables across a redirect is by using the TempData dictionary.

For example, suppose you receive form data from a POST at login. The action method that procedure the POST might be something like the following:

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult LogIn(Seller person)

{

...

TempData["name"] = person.Name;

return RedirectToAction("ItemUpload");

}

In this example, before redirecting to the ItemUpload action, the name of the seller is placed in the TempData dictionary.  In the ItemUpload action method, the seller name is retrieved from the TempData dictionary and placed in the ViewData dictionary so it can be referenced in the view.

public ActionResult ItemUpload()

{

string name = TempData["name"] as string;

ViewData["name"] = name;

return View();

}

## **DO use an OutputCache filter for static pages.**

Use **OutputCache** attribute when you are returning less frequently updated data; a good candidate may be your home page. You can use this technique for both HTML and JSON data types. When using it, only specify the cache profile name; do not specify anything else. If you need to fine tune caching, use the output cache section of the Web.config file.

For example, the OutputCache attribute is attached to Dashboard action method in the following code.

[AcceptVerbs(HttpVerbs.Get), OutputCache(CacheProfile = "Dashboard")]

public ActionResult Dashboard(string userName, StoryListTab tab, OrderBy orderBy, int? page)

{

...

}

In the Web.config file, the duration is fine tuned to 15 seconds.

## **DO consider using asynchronous controllers for long running requests.**

ASP.NET’s threading pool has a default limit of 12 concurrent worker threads per CPU. When the requests overload the server’s ability to process these requests, a queue is built up of requests. For example, any request which takes a considerable amount of time waiting for external resources, such as database or large file operations. These external requests block the thread they occupy for the entire wait period. When this queue gets too large (5000 requests pending), the server starts responding with 503 (server too busy) errors.

In ASP.NET 4 the number of concurrent threads is set by default to 5000. While it is possible to increase the default limits, there is a better way to mitigate long running requests from tying up threads, modifying the long running requests to run asynchronously. ASP.NET MVC enables you to implement asynchronous controllers for this purpose. For more information about how to implement an asynchronous controller, see [Using an Asynchronous Controller in ASP.NET MVC](http://msdn.microsoft.com/en-us/library/ee728598.aspx).