FatModel

Version 1.1

Best Practices Framework
for
ASP.Net MVC

By
LomaCons

February 5, 2016

Table of Contents

Software License

LomaCons FatModel software is licensed, not sold. If you do not agree to these terms you cannot utilize this software and you must destroy all copies. Custom licensing agreements, and additional website licenses, can be made by contacting LomaCons.

This license governs use of the accompanying software. If you use the software, you accept this license. If you do not accept the license, do not use the software. This license agreement is valid without the licensor's signature or the licensee's signature. It becomes effective upon the licensee's installation or use of the software.

1. Definitions

The "licensor" is LomaCons.

The "licensee" is the user of this software.

A "website" is a single instance that can be identified by a unique hostname or fully qualified domain name.

"Software" are all assemblies, documentation, examples, and any other files included with this distribution.

The following grants are subject to the terms of this license, including the conditions and limitations in section 3.

2. Grant of Rights

(A) The licensor grants the licensee a non-exclusive, worldwide, royalty-free license to use the software.

(B) The licensee may install the software on no more than four websites. If the licensee has purchased the source code, the licensee may install the software on no more than twenty-five websites. The licensee may make copies of the software for backup purposes only.

(C) The licensee may not redistribute the source code.

(D) If the licensee has purchased the source code, derivatives of the LomaCons.FatModel.Mvc.dll assembly, in a similar compiled dll form, may be distributed under a license of the licensee's terms, only when the distributed assembly file has been renamed and obfuscated. All other files, source code, and other derivatives of the software may not be redistributed.

(E) The licensee may not use, or create derivatives of any portion of the software or source code with the intent to create a similar or competing product.

3. Conditions and Limitations

(A) The software is licensed "as-is." The licensee bears the risk of using it. The licensor gives no express warranties, guarantees or conditions. The licensee may have additional consumer rights under local laws which this license cannot change. To the extent permitted under local laws, the licensor excludes the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

(B) The licensor has the right to terminate this license agreement and licensee's right to use this software upon any breach by licensee.

(C) The licensee agrees to remove, uninstall and destroy all copies of the Software upon termination of the License.

(D) The licensee cannot rent or lease the software. The licensee may permanently transfer this license, provided that the licensee uninstalls and retains no copies, that the licensee transfers all the software, upgrades, derivatives, media and documentation and that the transferee agrees to be bound by the terms of this license.

(E) This license does not grant the licensee rights to use the licensor's name, logo, or trademarks.

(F) All right, title, copyrights and interest in the software are owned exclusively by the licensor. The software is protected by copyright laws and international treaty provisions. The software is licensed to the licensee, not sold to the licensee. The licensor reserves all rights not otherwise expressly and specifically granted to the licensee in this license.

(G) If for any reason a court of competent jurisdiction finds any provision of this license, or any portion thereof, to be unenforceable, that provision of this license will be enforced to the maximum extent

permissible so as to effect the intent of the parties, and the remainder of this license will continue in full force and effect.

Other Licenses

Portions of FatModel and related samples make use of components from Microsoft and other authors as published on NuGet. Some of these software and tools are included in the FatModel distribution and are redistributed in its unmodified form under one of the licenses below.

1. Product Overview

The LomaCons FatModel framework can be added to any ASP.Net MVC based applications to enhance coding best practices and add notifications to MVC models.

Out of the box, Microsoft provides a tremendous amount of guidance, and supporting material to get started working with MVC Routing, Controllers, Models and Views. No doubt these are important to know first, however very little consistent guidance is provided for how to build models that can be used to properly separate concerns while being the conduit to transfer data, via models, from the controller to the view.

FatModel fills this gap for both those web developers that are new to ASP.Net MVC, as well as for more seasoned developers that want a better methodology to develop web based applications. Even if you have years of experience with Microsoft software, tools and application development, there is always something new that can be learned.

The FatModel framework provides intuitive and unobtrusive base classes and interfaces for models and controllers that facilitate consistent sub classing and usage of MVC Models. The accompanying documentation (including this document) provide a solid set of best practices for ASP.Net MVC, which can be used with FatModel, or used independently with any ASP.Net application.

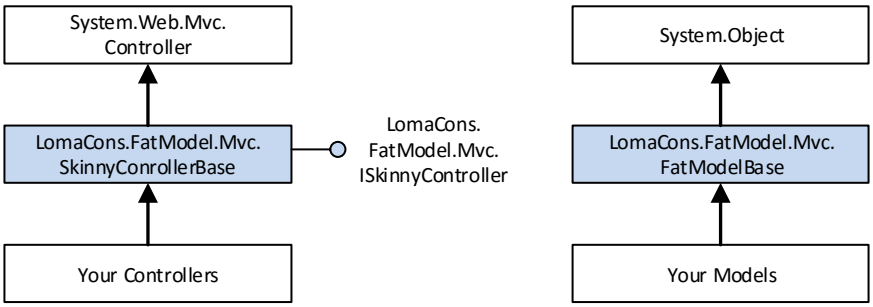
Features

FatModel has the following features.

- FatModel is based on a common best practice known as Skinny Controller, Fat Model, and Dumb View. The classes and interfaces provided make it easy to implement and conform to this best practice.
- Easy to use - Just add one method to your startup config code.
- Can be used as the basis for a new ASP.Net MVC application.
- Can be added to existing ASP.Net MVC applications. Your existing controllers and models will continue to function as they do now, without any code changes. Add and use FatModel where you want it.
- Provides additional View helpers for Ajax forms and controllers that allows a RedirectTo from a partial view controller. (coming soon)
- FatModel can be used with applications written in either C# or VB.Net. Works with any version of Visual Studio that supports ASP.Net MVC5, .Net 4.5.2 or newer - version 2012 or newer.
- Written in Visual Studio ASP.Net C# and based on the .Net Framework 4.5.2 and MVC 5.

FatModel Architecture

The diagram below shows the inheritance and placement of the various FatModel model and controller objects within an ASP.Net MVC based application that utilizes FatModel. FatModel sits in-between the Microsoft base classes and your classes.



FatModel Inheritance Architecture

How and where you use FatModel is up to you. It is only necessary to inherit from FatModel and SkinnyControllerBase when you want to add FatModel behaviors to your controllers and models.

Alternatively, you can add FatModelBase and SkinnyControllerBase as the base class for every controller and model in your project. Both are lightweight, and introduce very little overhead, even if the notifications are not used.

System.Web.Mvc.Controller
The Controller class is defined in the ASP.Net MVC framework assemblies and is the base class for all controllers.

LomaCons.FatModel.Mvc.SkinnyControllerBase
SkinnyControllerBase is derived from Controller and is the base class for your controllers. This base class provides the methods and properties for use with FatModels.

LomaCons.FatModel.Mvc.ISkinnyController
ISkinnyController is an interface that provides the ModelState and a minimum number of controller owned resources for the Model. This helps to keep the coupling low between the controller and model. You do not need to implement this interface. It has already been implemented on SkinnyControllerBase.

Your Controllers
Your controllers will work the same as any other ASP.Net controller you have seen or written before, however instead of deriving directly from Controller, your controllers will derive from SkinnyControllerBase.

System.Object
System.Object is the base class for all .Net objects. It is also the default base class for all models, because Microsoft does not provide a default model base class.

LomaCons.FatModel.Mvc.FatModelBase
FatModelBase is the base class for all models where you want to add instrumentation and notifications to your models.

Your Models
Your models will inherit from and override the virtual methods from FatModelBase. These methods and properties will provide the hooks for executing your server side code for populating and validating your model content.

2. FatModel Installation Prerequisites

FatModel requires the installation and configuration of Visual Studio 2012 or newer and ASP.Net MVC 5 or newer components and features. If you are already building applications with ASP.Net MVC, then you are ready to get started.

Visual Studio and Frameworks

The features below are typically installed along with Visual Studio or can be pulled in via NuGet packages. The versions listed are the current stable and minimum supported versions by Microsoft at the time that the provided assemblies and documentation was last updated.

- Visual Studio 2012, 2013 or 2015
- .Net Framework 4.5.2 or later
- ASP.Net MVC 5.2.3 or later

In general, if you are already developing applications with Visual Studio and the current release of ASP.Net MVC, these components are likely already installed.

Slightly earlier versions are supported with the source code edition of FatModel. The code for FatModel will work unchanged with any version of .Net Framework 4.x.x and ASP.Net MVC 5.x.x. To support these versions, you will need to rebuild the FatModel source code targeting the appropriate framework and referencing the proper assemblies.

Please note that support for .Net4.5.2 or later is not included with Visual Studio 2012, but it can be added. Please contact Microsoft for more information about how to install support for version of the .Net framework you are targeting.

Domain Knowledge

The biggest prerequisite for FatModel is a good understanding of Visual Studio, C# or VB.Net, ASP.Net MVC, HTML and web applications in general. In depth knowledge is not required, however, if you are a junior programmer, and have already worked though some of the basic samples provided by Microsoft on getting started with ASP.Net MVC, you should be ready for FatModel.

3. Adding FatModel to a Project

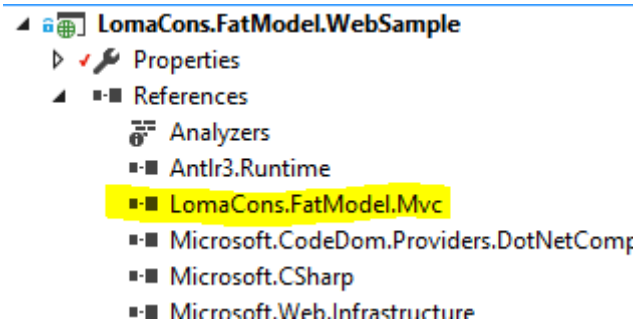
Follow the six steps below to add FatModel to any ASP.Net MVC application. FatModel can be added to an existing project or a new project.

1. Add a reference to the LomaCons.FatModel.Mvc assembly.
2. Initialize the FatModel binder in the ASP.Net startup code.
3. Derive a controller from SkinnyControllerBase.
4. Derive a model from FatModelBase.
5. Create a controller action
6. Override the FatModel notification methods.

The step-by-step examples that follow, will show how to use FatModel as the controller and model to back up a view. The view will contain a data entry form that is used to create a unique Person object. A complete working example of this code and additional examples can be found in the included FatModel Web Sample.

Reference the Assembly Dll

In your ASP.NetMVC web application project add a reference to the LomaCons.FatModel.Mvc.dll assembly.



Project References

Initialize the FatModel Binder

In your global.asax.cs file add a call to the FatModelConfig.RegisterBinder() method. The RegisterBinder() method adds the custom logic to the MVC framework that will call the notifications in your FatModels.

Your application start method should look similar to the one below.

```
protected void Application_Start() {
   // Standard ASP.Net Mvc Startup code
   AreaRegistration.RegisterAllAreas();
   FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
   RouteConfig.RegisterRoutes(RouteTable.Routes);
   BundleConfig.RegisterBundles(BundleTable.Bundles);

   // set up the Fat Model binder
   FatModelConfig.RegisterBinder();
   return;
}
```

Derive a Controller from SkinnyControllerBase

Create a controller and derive it from LomaCons.FatModel.MVC.SkinnyControllerBase. In the example below, PersonController is derived from SkinnyControllerBase. Person controller will contain our Create action methods.

```
public class PersonController : SkinnyControllerBase {
   public PersonController()
       : base() {
       return;
   }

. . .
}
```

Derive a Model from FatModelBase

Derive a Model to be loaded with content, bound, validated, and passed to the view. This model will look much like any other MVC model that you are already familiar with, where there are properties and attributes to support MVC validation and binding.

In the example below, the model will be used with both the Get and Post controller action methods. In addition to all the usual model content, the code below contains other methods and properties, which support the FatModel notifications and MVC best practices.

- The model constructor will be passed a reference to the controller as an ISkinnyController. This interface must be passed down to the base class constructor.
- The OnLoad() virtual method is overridden, such that the HTTP GET will be used to initialize the model properties with initial values, and the HTTP POST will save the changes to the data store repository. In the example that follows the IsPostBack property is evaluated, and if it is false (indicating the GET), the initial values of the model properties are set. If IsPostBack is true (indicating the POST), and the ModelState is valid, the business layer is called to persist the changes.
- The OnBound() virtual method is overridden, such that the HTTP POST will used to perform cleanup of posted content immediately after the model is bound but before the values are accessed for validation and business logic processing. In the example that follows, the string values that were posted to the server are checked for null values and whitespace is trimmed.
- The OnValidate() virtual method is overridden, such that server side validation can be done during the HTTP POST. In the example that follows, a business method is called to determine if the person already exists. If the person exists a model error is added to the ModelState.

The list above is not inclusive of all the properties, methods, and events on FatModelBase. See the following chapters for additional information on all methods, properties and events in the FatModel framework.

```
public class PersonCreateModel : FatModelBase {

  public PersonCreateModel(ISkinnyController controller)
      : base(controller) {
      return;
  }

  protected override void OnBound() {
      base.OnBound();

      if (IsPostBack == false)
          return;
      if (FirstName == null)
          FirstName = string.Empty;
      if (LastName == null)
          LastName = string.Empty;
      FirstName = FirstName.Trim();
      LastName = LastName.Trim();
      return;
  }

  protected override void OnValidate() {
      base.OnValidate();

      if (_business.GetPerson(FirstName, LastName) != null)
          ModelState.AddModelError("Failure",
                       "A Person with this First Name and Last Name already exisits.");
      return;
  }

  protected override void OnLoad() {
      base.OnLoad();

      if (IsPostBack == true){
          if (ModelState.IsValid){
              _business.CreatePerson(FirstName, LastName, Age);
          }
          return;
      }
      FirstName = string.Empty;
      LastName = string.Empty;
      return;
  }

  [Required]
  [StringLength(48)]
  [Display(Name = "First Name")]
  public string FirstName { get; set; }

  [Required]
  [StringLength(48)]
  [Display(Name = "Last Name")]
  public string LastName { get; set; }

  [Display(Name = "Age")]
  [Range(0, 150)]
  public int? Age { get; set; }

}
```

Add a GET Method Action

The next step is to add a GET method action to the controller. This will handle the HTTP GET action from the browser when the web page is first requested. In the example below, the CreateFatModel method is called to create an instance of the PersonCreateModel, and that model is then passed to the View.

This controller action method is much like any other action method that you have seen before. However, instead of creating the model directly with a new statement, the CreateFatModel method is used. This

method will create the fat model, set the IsPostBack property on the model to false, bind the model, and call the OnLoad override.

```
[HttpGet]
public ActionResult Create() {
    PersonCreateModel m;
    m = base.CreateFatModel<PersonCreateModel>();
    return (View(m));
}
```

Add a POST Method Action

The next step is to add a POST method action to the controller. This will handle the HTTP POST action from the browser when the form in the web page is posted back to the server. In the example below, the model is bound by the MVC model binder just prior to the execution of the action method. During the model binding the model IsPostBack property is set to true and the model binding and validation notification methods are called. If the model state is valid, the business logic is called from within the model to persist the person to the data store.

After the model binding and notification is complete, the controller action below is called, where the model state is checked. If the model state is not valid, the View is returned. Otherwise a Redirect action is returned to the browser.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(PersonCreateModel model) {
    if (ModelState.IsValid == false)
        return (View(model));
    return (RedirectToAction("Samples", "Home"));
}
```

View

The view in ASP.Net MVC is responsible for rendering the result and sending the content back to the client. The FatModel framework does not change anything about the way that you code your views and does not change anything about how you like your view to utilize the model. Therefore, the current way that you prefer to work with your views and your preferred view rendering engine, remains unchanged.

However, as a best practice, the view should be dumb. In this context, dumb means that the view should access only the property values defined on the model to determine how to render the view. Views should not call business logic; views should not modify the model; and views should never call public methods on the Model.

The client in this example is the browser and in the example below, the PersonCreateModel is used to render the view as an MVC form, utilizing the C# Razor view engine, complete with data entry control binding and validation.

```
@model PersonCreateModel
@using (Html.BeginForm()) {
  @Html.AntiForgeryToken()
  <fieldset>
      <div class="row">
          <div class="col-md-12">
              @Html.ValidationSummary(false, null, new { @class = "text-danger" })
          </div>
      </div>
      <ul class="nav nav-tabs">
          <li class="active"><a data-toggle="tab" href="#sectionA">Person</a></li>
      </ul>
      <div class="tab-content">
          <div id="sectionA" class="tab-pane fade  in active">
              <div class="form-group col-md-6">
                  @Html.LabelFor(model => model.FirstName,
                              htmlAttributes: new { @class = "control-label" })
                  @Html.EditorFor(model => model.FirstName,
                              new { htmlAttributes = new { @class = "form-control" } })
              </div>
              <div class="form-group col-md-6">
                  @Html.LabelFor(model => model.LastName,
                              htmlAttributes: new { @class = "control-label" })
                  @Html.EditorFor(model => model.LastName,
                              new { htmlAttributes = new { @class = "form-control" } })
              </div>
              <div class="form-group col-md-2">
                  @Html.LabelFor(model => model.Age,
                              htmlAttributes: new { @class = "control-label" })
                  @Html.EditorFor(model => model.Age,
                              new { htmlAttributes = new { @class = "form-control" } })
              </div>
              <div class="clearfix"></div>
          </div>
      </div>
      <div class="form-group">
          <div class="col-md-12">
              <div class="pull-right">
                  @Html.ActionLink("Cancel", "Samples", "Home",
                                          null, new { @class = "btn btn-link" })
                  <input type="submit" value="Create Person" class="btn btn-success" />
              </div>
          </div>
      </div>
  </fieldset>
}
```

Business Layer / Data Repository

In the example above, you might notice that all interaction with the business layer (the layer that reads and writes content to/from the data storage repository) occurs within the Model and no business layer calls are made from within the Controller.

It is neither right nor wrong to have all business layer calls in the model, all in the controller or in both. Where you call your business layer is entirely up to you and what you feel should be your best practice. Therefore, FatModel does not enforce where your business layer calls occur.

See the included WebSample project for examples of both methodologies.

Conclusion

Looking back at the example above, you can see how the controller is skinny, the model is fat, and the view is dumb. If you have been working with ASP.Net MVC for a while, you should quickly see what has happened.

The net effect of using FatModel is that the model state, validation and binding logic has moved into the model, where this logic belongs. The code to perform these tasks are put neatly in consistently named and overridden methods. As the application grows and additional pages are added, these common method names help enforce where this code logic happens and easily creates consistent patterns across the many models.

The controller has become skinny. The controller now contains only the action methods, and these actions methods are short and concise. For the most part, the action methods are now only responsible for creating the model and determining which action result to return.

The view continues to be dumb, as ASP.Net MVC Views should be. The views, and how you utilize them, remain unchanged from the patterns that you use currently.

4. HTTP, ASP.Net MVC and FatModel

There is a lot of material and information about how to get started with ASP.Net MVC, and FatModel is no exception. The guidelines in this chapter will provide the basics of how the HTTP Request and Response cycle interacts with the ASP.Net Framework, and how the FatModel notifications interact with the ASP.Net Framework.

HTTP Request and Response

The HTTP protocol defines the methodology for communicating between a client and a server. The client typically will be a web browser, and the server typically will be a web server. Regardless of the kind of browser or server, the data transferred to and from the server will follow the HTTP standard.

Simply defined, the client will send a GET or a POST request to the server in response to some user event such as a link or button click, and the server will send a response. The response is then displayed by the client, and the process starts again.

ASP.Net MVC Controller Action Methods

ASP.Net MVC provides the server side request interpretation and response creation. In the simplest form, the request response cycle for MVC happens in this order, and is managed by the controller.

For an HTTP GET request, these steps occur in this order.

1. Routing interprets the HTTP GET Request, matching it with a unique controller and action method
2. The action method is executed
3. The action method creates a model object using the CreateModel<T>() method
4. The model is bound and loaded with properties and values to be shown by the view
5. The controller action method returns an action result that passes the model to the view.
6. The view retrieves the properties from the model, and renders the view
7. The view rendering is sent via the HTTP Response back to the client.

The HTTP POST request handling is similar and these steps occur in this order.

1. Routing interprets the HTTP POST Request, matching it with a unique controller and action method
2. The model with properties that match the request parameters is created
3. The model is bound, validated and loaded with properties and values to be shown by the view
4. The action method is executed
5. The controller action method returns an action result that passes the model to the view.
6. The view retrieves the properties from the model, and renders the view
7. The view rendering is sent via the HTTP Response back to the client.

Alternatively, the controller action may return a redirect result (instead of a view result.) The redirect result will instruct to browser to redirect to a different URL.

Regardless of GET or POST, FatModel provides automatic notifications.

HttpGet and FatModel Notifications

Models derived from FatModelBase will have these virtual methods called in this order during an HTTP GET request.

- The constructor is called to create the model object
- OnInitialize() - called after the model has been created
- OnBinding() - called just before MVC binds the model to any query string parameters
- OnBound() - called after MVC has performed model binding of query string parameters
- OnLoad() – called when the model content can be loaded

HttpPost and FatModel Notifications

Models derived from FatModelBase will have these virtual methods called in this order during an HTTP POST request.

- The constructor is called to create the model object
- OnInitialize() - called after the model has been created
- OnBinding() - called just before MVC binds the model to any query string and form parameters
- OnBound() - called after MVC has performed model binding of query string and form parameters
- OnValidate() - called before MVC validates the model
- OnValidated() - called after MVC has completed model validation
- OnLoad() – called when the model content can be loaded

Other Request Verbs

FatModel is not limited to just GET and POST. FatModels and SkinnyControllers will work just the same with PUT, DELETE and more.

5. FatModel API Reference

This chapter covers all the public and protected FatModel objects, methods, properties, overrides and events. Although all the samples and descriptions that follow are written in C#, the API can be called from VB.Net just as well.

FatModelConfig

The FatModelConfig class contains the startup logic to add FatModel notifications to the ASP.Net MVC model binding process.

```
public static void RegisterBinder()
```

A call to the RegisterBinder method must be added to the startup code in the global.asax file.

SkinnyControllerBase

SkinnyControllerBase is the abstract base class for any controller that will make use of a FatModel. Simply derive your controller class from SkinnyControllerBase

```
protected T CreateFatModel<T>() where T : FatModelBase
```

T – The type of model to create. The model must be derived from FatModelBase.

This method creates a fat model object. It should be called from an action method to explicitly create a FatModel on an HttpGet action. This will create the model and call the notifications, while the MVC data binder assigns values to properties if they are found.

```
protected T CreateFatModel<T, V>(V value) where T : FatModelBase
```

T – The type of model to create. The model must be derived from FatModelBase.

V – The type of value to pass to the constructor of the model T

This method creates a model object. It should be called from an action method to explicitly create a FatModel on an HttpGet action. This will create the model and call the notifications, while the MVC data binder assigns values to properties if they are found.

V is the type of the value will be passed to the constructor of T. This allows a parameter or some other value or object to be passed from the controller action to the constructor of the FatModel.

```
public HttpVerbs RequestMethod { get; }
```

This property is set by SkinnyControllerBase and returns the current request method. Most often the request method will be GET or POST. Other possible request methods can be DELETE, HEAD, OPTIONS and PUT.

Internally, this property will be used by the FatModel binder to set the IsPostBack property to true when the request method is POST.

FatModelBase

FatModelBase is the abstract base class for all FatModels. This model contains all the properties, notifications and events for loading, validating and updating data in the model.

```
protected ModelStateDictionary ModelState { get; }
```

The ModelState object is created and owned by the ASP.Net MVC Controller but is exposed as a property on the FatModel so that it can be manipulated and updated within the model. Use this ModelState the same way that you would use the ModelState in the controller.

```
protected bool IsPostBack { get; }
```

The IsPostBack property is true whenever the Http Request method is HttpPost. The value is false for any other request method, including HttpGet.

This property is most useful to determine how to load or update content in the model when using the best practice of Get-Post-Redirect-Get.

```
protected virtual void OnInitialize()
```

The OnInitalize method is called immediately after the Model has been fully created by the constructor. Use this method to include logic that needs to happen after the model has been fully created, but before any data binding and validation.

```
protected virtual void OnBinding()
```

The OnBinding method is called after the OnInitalize, but just before the ASP.Net MVC property binder is run to set properties from the query string and/or form.

```
protected virtual void OnBound()
```

The OnBound method is called after the ASP.Net MVC property binder sets all the model properties from the query string and/or form.

```
protected virtual void OnValidate()
```

The OnValidate method is called only on a Post Back, after the properties have been bound, but before the ASP.Net MVC validation logic is run.

```
protected virtual void OnValidated()
```

The OnValidated method is called only on a Post Back, after the ASP.Net MVC validation logic is run. Use this method to execute any custom server side validation code and logic.

```
protected virtual void OnLoad()
```

The OnLoad method is the last method called during the FatModel data binding process, and just before the flow of control is handed back to the controller's action method. Use this method to set default values when IsPostBack is false; and use this method to persist changes when the ModelState is valid and IsPostBack is true.

```
public event EventHandler Initialize;
public event EventHandler Binding;
public event EventHandler Bound;
public event EventHandler Validate;
public event EventHandler Validated;
public event EventHandler Load;
```

These six events will be fired during the base implementation of the above similarly named virtual methods, and can used in the same situations. The events exist only for cases where an event would be more useful than overriding a virtual method. Which you use is up to you and your personal style.

```
public EnumClearModelStateBehavior ClearModelStateBehavior { get; protected set; }
```

The ClearModelStateBehavior property defines how the ModelState behaves after the OnLoad method completes. The default is None, which mimics the standard ASP.Net MVC model state behavior.

By default ASP.Net MVC keeps a hidden duplicate copy of all the bound values in the ModelState. The view helpers for .TextFor and .EditorFor will bind to the values in the ModelState first, and if the values are not in ModelState the view helpers will bind to the model properties. Most times this goes unnoticed as the hidden values in the ModelState will be the same as those in the model itself.

However, there are times where you will want the view to use the values in the model, not the model state. To do this, in your OnLoad override, set the ClearModelStateBehavior to OnPostAfterLoad or OnPostAfterLoadIfModelIsValid

Follow this link for a full technical description about how and why the view helpers use both model state and model when setting the values in the view. https://weblog.west-wind.com/posts/2012/Apr/20/ASPNET-MVC-Postbacks-and-HtmlHelper-Controls-ignoring-Model-Changes

6. ASP.Net MVC Best Practices

Microsoft provides lots of material, videos and other documentation on how to get started with MVC, describing what MVC is, and a few Hello World type of examples. One thing that Microsoft has never been very good at is the next steps.

Often these next steps are left to the consumer and developer to figure out on their own, and MVC itself does not guarantee that the application will be well written. This results in junior developers who write unstructured spaghetti code, who grow up to be developers who continue to write spaghetti code, because they've always done it that way. Some developers grow up to become senior developers and system architects who break the bad code mold, and are stuck fixing and maintaining someone's spaghetti code. The lucky ones, learn some best practices and ultimately get the opportunity to build something great, leaning on the trial and tribulations of the knowledge gained from past failures. If only Microsoft would provide some best practices, then maybe spaghetti code would not be as common, and we'd all be building great applications from the get-go.

The remainder of this chapter focuses on some MVC best practices, as well as why these are best practices. Keeping in mind that best practices are partly a matter of opinion, you may or may not agree with what follows. Please take away what you want, and ignore the parts that you disagree with. However, should you follow all of these best practices and utilize FatModel, you can be assured that you are building solid, maintainable web applications that can grow and change over time.

Disclaimer aside, here are LomaCons's ASP.Net MVC Best Practices. For additional details on these best practices, as well as supporting arguments and code samples from third parties, please search the internet for more information.

Get-Post-Redirect-Get

A common pattern is to use Get to initially load a page, Post to update the data store, then a Redirect to Get the next page. On the server side on the connection, the HttpGet controller action must never modify data in the data store; only the HttpPost action is allowed to modify the data store.

The big advantage for this best practice is that it ensures that when the user refreshes their browser window, that data is not modified because a client-side-refresh will only re-issue the Get request. On the server side, the code will be consistent such that HttpGet actions are "read-only", while HttpPost actions are "read-write" in regards to the data store.

HttpGet Actions Should Never Modify Data

This is a corollary to the Get-Post-Redirect-Get best practice above. Get actions should be used to retrieve the content to display in the browser, and should never, ever, modify data in the data store.

Controllers should be Skinny

The controller should do one or more of the following

- Create a model
- Handle the action requested
- Determine the ActionResult to return as the response.

Anything more, should be handled within the model and/or business layers. There should be little to no business logic in a controller.

One of the peculiarities of ASP.Net MVC is that the controller holds onto the ModelState. The ModelState in the MVC pattern should be part of the Model, not the Controller, and the Controller should only read the model state, to determine if it is valid, and to return an action result accordingly . Oddly enough,

because MVC does not provide a true base class for model, the ModelState cannot live in the model. FatModel solves this by adding a reference to the ModelState into the FatModelBase class.

How skinny should the controller be? That is up to you, but with FatModel, you can move most all calls to the business layer into the Model, thus making the controllers very skinny. Or you can call business layer methods from your controller, reading and writing values to/from the model. Or both. Either way is valid and FatModel does not enforce one or the other. Where and how you call your business layer methods is up to you. (But as a best practice, the business layer should never be called from the view.)

Models should be Fat

Models should be fat, in that they should do the majority of the work during the request and response cycle. Models should do one or more of the following

- Receive the values from the form and/or query string.
- Be the container for properties and lists to be displayed by the view.
- Be the container for the state of the model.
- Validate properties that were sent to the server and set the ModelState accordingly.
- Read data from the business layer.
- Update or create content via the business layer.
- The role of the model is to be the container for the data that the View will display

Views must be Dumb – Don't Modify the Model

The view is responsible for rendering the output and returning it to the caller. The model is the container for the data points that the View will render. Once the controller passes the model to the view, the model content must remain static. The view can read the values from any property or object exposed by the view, and should never modify the view.

A corollary to this is that the View should never access the business layer, not the database. Any interaction with the business layer and/or database should have been done before the view code is executed - when the Model was created or during the execution of the controller action.

Do NOT use ViewData and ViewBag

Never, ever, use ViewData or ViewBag for a real web application. At first, this best practice might seem counterintuitive, as many samples and starter projects from Microsoft and others make use of it, thereby indirectly suggesting that it is OK to use. Yes, these are easy to use, and because they are easy to use these objects often end up in samples and code demos as a substitute for a real model object.

The problem with ViewData and ViewBag is that you need to use magic strings and object casting which the controller and view must interpret at runtime. Often this leads to confusing code that is difficult to maintain over time and syntax bugs that cannot be found until runtime. ViewBag attempts to resolve some of this by making use of dynamic variables which provide pseudo-strong typing of variables, but it is still a bad idea to use.

Another problem with ViewData and ViewBag is the confusing nature of it, and how it is used in many samples. Is it a replacement for a model? Do you set ViewData properties in the controller, or in the model? The default web project from Visual Studio goes so far as to set properties in the view itself, even though views should never modify the model! (Can you start to taste the fresh spaghetti?)

So when is ViewData and ViewBag ok to use? The answer is probably never. The one exception might be for a throwaway prototype or a small sample application where no models are needed. In this case there is no model and instead the ViewBag is the model for passing content from the controller to the view.

However, considering that many good intentioned throwaway prototypes will ultimately become the basis for the real product, this is not a good idea.

Therefore, resist the temptation to use ViewData and ViewBag. Always create a real model class.

Do NOT use TempData

Never, ever, use TempData for any application. Similar to ViewData, this will initially seem counterintuitive as many samples and starter projects from Microsoft and others make use of it, thereby indirectly suggesting that it is OK to use. However, any use of TempData is a red flag for a poorly designed application. (More fresh spaghetti.)

A properly designed application should be able to perform any controller action completely and independently of other controllers and actions. Controller actions should perform their work using the data provided within the request and/or data stored in the data store.

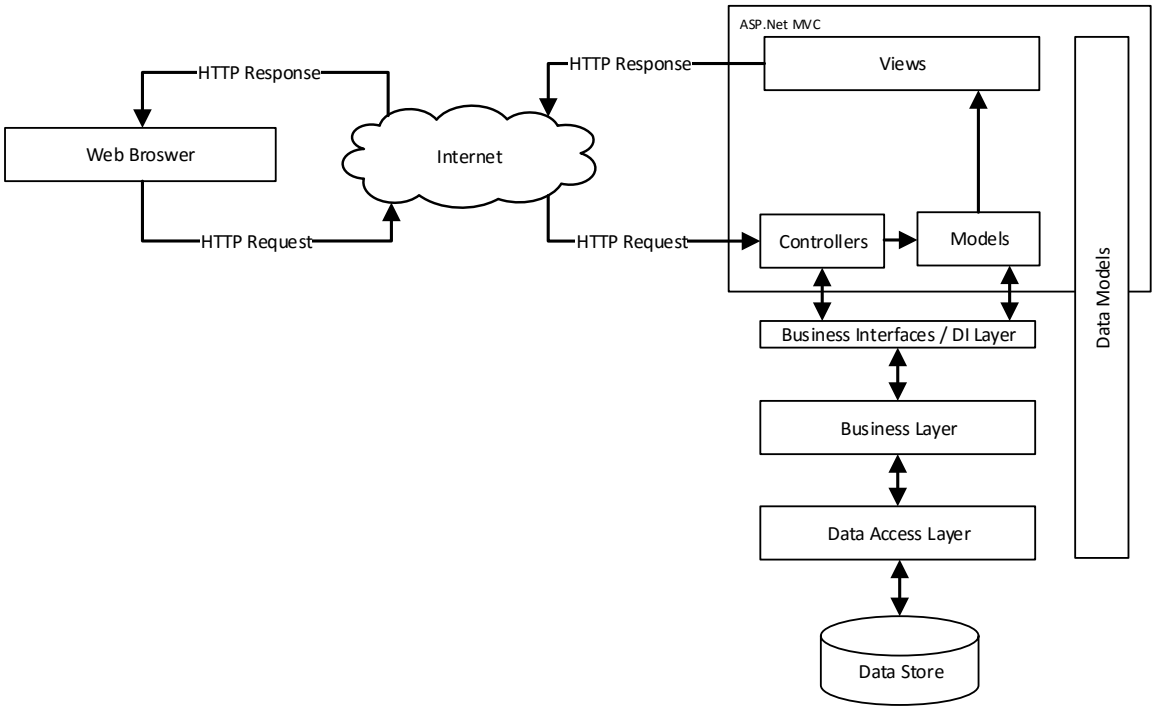
Even if you should decide to use TempData, there are other inherent problems with it. First the data that is stored in TempData is temporary and is lost after it is first accessed. Therefore if a user refreshes their browser, the controller will not be able to locate the data in TempData because it was removed by the prior request. Additionally, internally TempData is just a thin wrapper around the ASP.Net Session object, which if overused or used in a web farm can cause problems.

The bottom line is that any use of TempData should be a cause for alarm, and the code surrounding it should be refactored to eliminate it.

Implement Layers

Some developers have convinced themselves that ASP.Net MVC will solve all their coding nightmares and problems of the past. However, ASP.Net MVC by itself will never solve the problem of poorly written applications. Only carefully planned and designed applications can be well written and maintainable over time.

Independent, purposeful, loosely coupled layers are critical to building any application with ASP.Net MVC. Consider the diagram below, and notice how ASP.Net MVC is only part of the solution and is really just the User Interface layer – it is not the entire application. This UI layer is aware of only the data models and the business layer. It is loosely coupled to the business layer by an IOC or DI layer. The UI is also aware of data models and data object that are passed up and down through the layers.



ASP.Net Application with N-Tier Application Layers

Although it is helpful, it is not necessary to break up the layers into separate assemblies. But even if you are building a single project web application, it is often advisable to put the data models, business logic and data access logic in separate namespaces.

UI Models are for the UI Layer

The purpose of an MVC Model is to hold the data that the view will render. The content it contains is either self-loaded when the model is created, or it is loaded by the controller. Once loaded with content, the model should not change.

Take note of the difference between a UI model and a data model. These are often confused and one should never be substituted or confused with the other. An MVC Model is strictly for use within the UI Layer – it is a model of the data to be rendered by the view. Data objects and entity models, such as those created by the Entity framework are not MVC Models. Models are not a data objects – although models may contain data objects.

Data Models are for all Layers

Data Models are a representation of the data that is held in the data store and are the only objects that can be visible in one way or another through all layers of the application. Data models can be individual class objects or full relational entities created by ORM tools such as the Entity Framework.

Common data stores are a database or the file system. Data models and data objects are persisted and read by the data access layer. They are then passed up and down through the layers.

Avoid Custom Routing

For browser based, web applications, stick with the default routing, which will be limited to these two routes.

- Controller/Action

- Controller/Action/Id.

Additional, more complex routing can be created, but this usually results in an unnecessary over-complicated code that needs to be maintained and debugged. Deeper routing should be used for RESTful services and Web base APIs, but not websites.

If you really need more values on your URLs, often it will be easier to use a query string with named parameters. The additional query string parameters will be bound to your named parameters in your action methods automatically by ASP.Net MVC.

7. Rebuilding the Source Code

If you purchased the source code for FatModel, you will be able to rebuild the FatModel assembly using Visual Studio 2012 or newer.

Keep in mind that you are still bound by the LomaCons FatModel license agreement when you rebuild the source code. If you rebuild the LomaCons.FatModel.Mvc.dll you must obfuscate this dll should you redistribute it in compiled form with your application. In addition, you cannot redistribute or resell any of the source code.

To rebuild the assembly, unzip the FatModelSource.zip file. Open the FatModelSource.sln file with Visual Studio, and rebuild the solution.

Visual Studio should automatically pull in the required NuGet packages when the solution is built for the first time. However, depending upon your Visual Studio settings, this may not happen. If you cannot compile the source and are getting missing assembly reference errors, you may need to explicitly tell Visual Studio to Restore the NuGet packages. To do so, right click on the solution in the Solution Explorer, and choose the option to restore the packages.

8. Future Enhancements

Many additional new features and customizations are planned for future releases. Any additional requests for new features or enhancements can be made by contacting LomaCons. See www.lomacons.com for information on how to contact LomaCons.

9. Development Methodology and Goals

Version 1 was the first major release of FatModel to benefit ASP.Net MVC web development, targeting both current developers and new developers. The goals of the first release of version 1 follow below. Many of these will continue to be guiding principles for current and future releases of FatModel.

Make FatModel easy to use
We want to ensure that those who begin to use FatModel can start working with it right away, with just a small learning curve. It should take no more than a few minutes for someone new to FatModel get it installed and start working with it.

Make FatModel Non-intrusive
We want to ensure that FatModel adds productivity and power where needed, without complexity and overhead. So adding FatModel to a project does not force the developer to use it for all controllers and models. Rather, it can be added only to those controllers and models where the developer wants to make use of it.

Similarities to ASP.Net WebForms
FatModel is 100% MVC and has nothing to do with ASP.Net WebForms – However, one thing that may be noticed if you have done some ASP.Net WebForms development is the naming conventions for the IsPostBack property and the notification methods. This was intentional, as the names of these properties and notifications have always been the common names for placeholders where your application logic should go. Simply put – these names just feel right.

For developers coming from WebForms, who have not done much work yet in ASP.Net MVC, FatModel will feel a little more at home, and the best practices should help when using a notification framework like FatModel.

However, conversely, many MVC developers will deliberately eschew anything related to WebForms, even to the point of denouncing and degrading it. Therefore, this is the only place in the documentation where WebForms will be mentioned. It is hoped that such MVC purists will be able to overlook the naming similarities and realize that FatModel is 100% MVC and follows modern best practices that they can be proud of.

Stay true to core values of less-is-more, easy and customizable
FatModel contains no complicated setup routines and is easy to use and add to any ASP.Net MVC application. FatModel is simple to use and will be the basis for more – Additional functionality and support for future ASP.Net MVC frameworks will be added over time.

10. Product Support and Contact Information

All product support is provided by web and email.

Frequently Asked Questions

Frequently asked questions and FAQs can be found at http://www.lomacons.com.

Email Support

Support for InstallKey can be sent to support@lomacons.com. In most cases, you should receive a reply by the next business day or sooner.

Sales and Information

Product sales can be reached by email at support@lomacons.com.