3

View Fundamentals

In this chapter, we will cover

* Something
* Another thign

The view's responsibility can be deceptively simple. Its entire goal in life is to take the model given to it, and use it to render content. Because the controller and related services already executed all business logic and packaged the results into a model object, the view only needs to know how to take that model and turn it into HTML. While this separation of concerns removes much of the responsibilities that can plague traditional ASP.NET applications, views still need to be carefully and deliberately designed. Views require knowledge and understanding of the building blocks of the web, including HTML, CSS and JavaScript.

In this chapter, we'll take examine how ASP.NET MVC renders views, the default WebFormsViewEngine, and structuring and organizing views. Next, we'll look at a couple different approaches for using the model to render content in a view. Finally, we will cover the templating features new to ASP.NET MVC 2.

3.1 View Fundamentals

A view's responsibility is to render content. But how does the MVC framework decide which view to use? How do we control what gets rendered, and how do we organize our content? How do we even tell MVC to render a View? In the ASP.NET MVC framework, the controller decides that a view should be rendered by returning a ViewResult object from a controller action, shown in Listing 3.1 below.

Listing 3.1: Using the ViewResult object to render a view

[Authorize]

public ActionResult ChangePassword()

{

return View();

}

Although the method name seems to indicate that a view is rendered as the result of calling the View method, it is merely a helper method in the Controller base class to create a ViewResult object. The ViewResult object contains all the information needed to render the view at a later time. This information includes the view name, the model, and other pertinent information an IViewEngine can use to render a view.

Internally, the ViewResult object delegates to a specific IViewEngine to render the actual content for a view. There a several more classes involved, but the IViewEngine implementation is the class responsible for examining the ViewResult information as well as other context information and locating the correct IView to render.

3.2 Examining the ViewDataDictionary

The main object used to pass model information to a view is the ViewDataDictionary class. Like other MVC frameworks, ASP.NET MVC exposes a dictionary to enable the controller action to pass any number of model objects and information to the view. With a dictionary object, we can pass as many items as need be for the view to render appropriately. For example, our application includes a profile page where users can view other user's profiles. However, only the current logged in user can edit their own profile. To display the profile information on the profile screen, we can pass in the Profile object directly to the view, shown in Listing 3.2 below.

Listing 3.2: The Profile class

public class Profile

{

public Profile(string username)

{

Username = username;

}

public string Username { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

}

While our Profile class has all the information needed to display our Profile, it does not include any information about the current logged in user, or whether the view should display the edit link. We need to give the view more information than solely the Profile object to make this decision. To do so, we can use the ViewData dictionary object for this extra piece of information, shown in Listing 3.3 below.

Listing 3.3: The Show controller action

public ViewResult Show(string username)

{

var profile = \_profileRepository.Find(username);

bool hasPermission = User.Identity.Name == username;

ViewData["hasPermission"] = hasPermission;

return View(profile);

}

In the Controller base class, we have access to the ViewDataDictionary object used to pass to the view in the ViewData property. We check the current user's name, compare it to the profile to be shown in the username parameter, and place the result of the comparison into ViewData with a "hasPermission" key. Next, we use the helper View method to create a ViewResult object and set the ViewData's Model property to our Profile object. On the view side, we will pull the "hasPermission" information out of ViewData, and use it to hide the Edit link, shown in Listing 3.4 below.

Listing 3.4: Using ViewData information to hide a link

<p>

<%

bool hasPermission = (bool)ViewData["hasPermission"];

if (hasPermission)

{ %>

<%=Html.ActionLink("Edit", "Edit", new { username = Model.Username }) %>

|

<%=Html.ActionLink("Back to List", "Index") %>

<% } %>

</p>

In our view, we extract the "hasPermission" information from ViewData. Next, we conditionally show the edit link based on the "hasPermission" variable. Finally, we display a link to take the user back to the profile list page. The final rendered page for showing the current user's profile is shown in Figure 3.1.

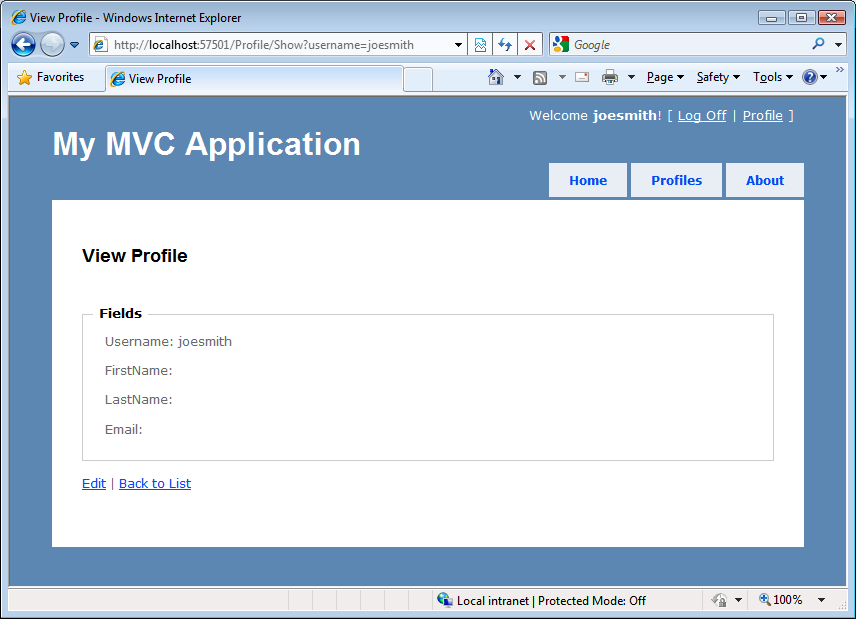


Figure 3.1: The current user's profile page

While using the technique of utilizing the dictionary aspects of the ViewDataDictionary gives us a lot of flexibility, it comes at a price. Because we create weak, compile-unsafe link in a dictionary, we open ourselves to problems in the future. For example, we might misspell "hasPermission" in the view, and only learn of our mistake at runtime. However, our use of the Profile object as our ViewModel gives us a strong link between controller action and view, compile-time safety, and IntelliSense in the view.

Using the loose-type semantics of a dictionary can also hinder us in more complex scenarios. Consider a login screen, where the username and password are required fields. With an actual object to represent the model for this view, we can then decorate our ViewModel object with validation attributes. In the next section, we'll look at taking advantage of ViewModel types with strongly-typed views.

3.2 Strongly-typed views with ViewModel

In the WebFormViewEngine, our views can inherit from two different types: System.Web.Mvc.ViewPage, or System.Web.Mvc.ViewPage<T>. The generic ViewPage<T> inherits from ViewPage, but offers some unique additions not available in the non-generic ViewPage class. The skeleton member definition of ViewPage<T> is shown in Listing 3.5 below.

Listing 3.5: Skeleton definition of ViewPage<T>

public class ViewPage<TModel> : ViewPage

{

public AjaxHelper<TModel> Ajax { get; set; }

public HtmlHelper<TModel> Html { get; set; }

public TModel Model { get; }

public ViewDataDictionary<TModel> ViewData { get; set; }

}

In addition to providing a strongly-typed wrapper over ViewData.Model through the Model property, we have access to strongly-typed versions of the associated view helper objects, AjaxHelper and HtmlHelper. To use a strongly-typed view, we first have to ensure that our controller action sets the ViewData.Model properly. In Listing 3.6, we retrieve all of the profiles for display in a list page, and pass the entire collection of profiles to the View method, which encapsulates setting the ViewData.Model property.

Listing 3.6: Passing a collection of profiles to our view

public ViewResult Index()

{

var profiles = \_profileRepository.GetAll();

return View(profiles);

}

In the Index view used with this action, even the loose-typed ViewPage class can use the ViewData.Model property. However, this property is only of type object, and we would need to cast the result to use it effectively. Instead, we can make our view page inherit from ViewPage<T>, shown in Listing 3.7 below.

Listing 3.7: Inheriting from ViewPage<T> for a strongly-typed view

<%@ Page Language="C#"

MasterPageFile="~/Views/Shared/Site.Master"

Inherits="System.Web.Mvc.ViewPage<AccountProfile.Models.Profile[]>" %>

By inheriting from ViewPage<T> instead of merely ViewPage, we now have a strongly-typed view. In the next section, we'll look at how we can use our ViewModel object to display information in a view.

3.3 Displaying ViewModel data in a View

Typically, to display information in a view, we will use the HtmlHelper object to help us use our ViewModel to generate HTML. However, we do have the option to output HTML directly from the Model object. In our profile list screen, we want to iterate over the profiles passed in our Model and display select information from each, shown in Listing 3.8.

Listing 3.8: Displaying a list of profiles in our view

<h2>Profiles</h2>

<table>

<tr>

<th>Username</th>

<th>First name</th>

<th>Last name</th>

<th>Email</th>

</tr>

<% foreach (var profile in Model) { %>

<tr>

<td>

<%= Html.Encode(profile.Username) %>

</td>

<td>

<%= Html.Encode(profile.FirstName) %>

</td>

<td>

<%= Html.Encode(profile.LastName) %>

</td>

</tr>

<% } %>

</table>

Because we would rather not open ourselves to the myriad of scripting attacks possible when displaying un-encoded user input to the screen, we encode all user-entered information displayed on a screen. In the example above, this is accomplished through the HtmlHelper object, provided through the Html property on our base ViewPage<T> (and ViewPage) class.

In our login page, we use a ViewModel object to represent the entire form, shown in Listing 3.9 below.

Listing 3.9: Our LogOnModel class

public class LogOnModel

{

[Required]

[DisplayName("User name")]

public string UserName { get; set; }

[Required]

[DataType(DataType.Password)]

[DisplayName("Password")]

public string Password { get; set; }

public bool RememberMe { get; set; }

}

The logon screen shows input elements for each of these properties, shown in Figure 3.2.

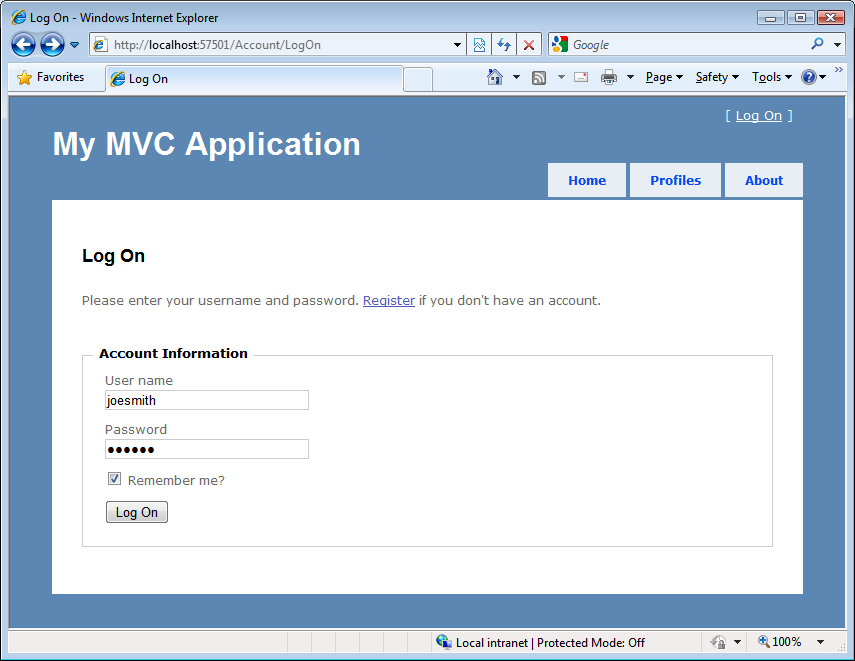


Figure 3.2: The log on screen

Because we opted for a strongly-typed view for our log on screen, we can use the built-in helpers to render the HTML for each input element. Instead of loosely-bound strings to represent the action parameters, we can take advantage of the expression-based HtmlHelper extensions for creating various types of input elements, shown in Listing 3.10.

Listing 3.10:

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

<div>

<fieldset>

<legend>Account Information</legend>

<p>

<%= Html.LabelFor(m => m.UserName) %>

<%= Html.TextBoxFor(m => m.UserName) %>

<%= Html.ValidationMessageFor(m => m.UserName) %>

</p>

<p>

<%= Html.LabelFor(m => m.Password) %>

<%= Html.PasswordFor(m => m.Password) %>

<%= Html.ValidationMessageFor(m => m.Password) %>

</p>

<p>

<%= Html.CheckBoxFor(m => m.RememberMe) %> <label class="inline" for="rememberMe">Remember me?</label>

</p>

<p>

<input type="submit" value="Log On" />

</p>

</fieldset>

</div>

<% } %>

In the code above, we take advantage of several of the HtmlHelper extension methods designed for strongly-typed view pages. Instead of a loose-typed string to represent properties, these helper methods utilize the C# 3.5 feature of expressions to generate HTML. Since these HTML elements need to be generated to match properties on objects, it is only fitting that the original types and objects are used with expression to generate the related HTML. The Html.LabelFor and Html.TextBoxFor methods used in the example above for the UserName property generate the HTML shown in Listing 3.11.

Listing 3.11: HTML generated from expression-based HtmlHelper methods

<label for="UserName">User name</label>

<input id="UserName" name="UserName" type="text" value="" />

For our page to pass accessibility validation, every input element needs to include a corresponding label element. Because our label and input elements are generated using expressions, we no longer need to worry about our hard-coding label and input names. Including the helper methods used above, the HtmlHelper extensions designed for strongly-typed views include:

* DisplayFor
* DisplayTextFor
* EditorFor
* CheckBoxFor
* HiddenFor
* PasswordFor
* RadioButtonFor
* TextBoxFor
* LabelFor
* DropDownListFor
* ListBoxFor
* TextAreaFor
* ValidateFor
* ValidationMessageFor

Since our form was generated using a strongly-typed view, we can take advantage of this in the design of our action that the form posts to. Instead of enumerating every input field as separate action method parameters, all of the parameters can be bound to the same ViewModel we used to render the view, shown in Listing 3.11.

Listing 3.11: The signature of the LogOn action using the ViewModel as a parameter

public ActionResult LogOn(LogOnModel model, string returnUrl)

{

if (ModelState.IsValid)

{

if (MembershipService.ValidateUser(model.UserName, model.Password))

{

FormsService.SignIn(model.UserName, model.RememberMe);

Our LogOn action method takes a single LogOnModel object, as well as the potential return URL, instead of a method parameter for each input element on our form. As powerful as the HtmlHelper extensions for strongly-typed views can be, we still introduce quite a bit of duplication in our views if we rely solely on these extensions for generating HTML. For example, if every input element requires a corresponding label, why not always include it? Every user interface is different, so the MVC team cannot predict the layout everyone wants to use for input and label elements. Instead, we can take advantage of a new feature in ASP.NET MVC2, templates, to enforce a standardized approach to generating HTML.

3.4 Using strongly-typed templates

As we move towards using strongly-typed views throughout our application, we will start to see more and more patterns emerge. If a ViewModel object has a boolean property on a form, we will almost certainly want to display a checkbox on a form. Email addresses should always render the same, as should password fields and so on. It is also very rare that an input element won't also include the corresponding validation message. We could tackle this duplication with HtmlHelper extensions, but it's still not quite as clean as we would like.

HtmlHelper extension methods work well for individual snippets of HTML elements, but tend not to scale as the HTML generated starts to become more complex and include more varieties of elements. If we also want to start basing our rendering decisions based on model metadata, such as the existence of validation attributes on our model, our HtmlHelper extensions start to become just unmaintainable without quite a bit of extra work.

With ASP.NET MVC 2, the MVC team designed a view feature that tends to sit between HtmlHelper extension methods and full-blown partials in size and scope. This feature is templates, and is designed to assist in generating HTML based on strongly-typed views. Templates can be used to generate HTML for the entire model or one member at a time. Because HTML for viewing and editing are radically different, generating templates for each is accomplished through two different sets of methods, with two different sets of templates.

3.4.1 EditorFor and DisplayFor templates

The Editor and Display templates are generated from the following methods:

* Html.Display("Message")
* Html.DisplayFor(m => m.Message)
* Html.DisplayForModel()
* Html.Editor("UserName")
* Html.EditorFor(m => m.UserName)
* Html.EditorForModel()

Although string-based versions exist for using templates against loosely-typed views, we will prefer to use the expression-based methods to gain the benefits of using strongly-typed views. If our model is very simple, we can use the "ForModel" methods, which enumerate over every member in our model to generate the complete HTML. Since our "Change Password" page is very simple, we can use the EditorForModel method, shown in Listing 3.12.

Listing 3.12: Using EditorForModel for a simple model

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

<div>

<fieldset>

<legend>Account Information</legend>

<%= Html.EditorForModel() %>

<p>

<input type="submit" value="Change Password" />

</p>

</fieldset>

</div>

<% } %>

This EditorForModel method loops through all of the members on our Model for this view, generating the editor templates for each member. Each template generated may be different, depending on the model metadata information on each member. This HTML might suit our needs, but there is only so much you can embed in your view model before you can no longer sanely emit HTML based solely on model metadata. The model for the Change Password screen, shown in Listing 3.13, already has validation and label information.

Listing 3.13: The Change Password model

[PropertiesMustMatch("NewPassword", "ConfirmPassword",

ErrorMessage = "The new password and confirmation password do not match.")]

public class ChangePasswordModel

{

[Required]

[DataType(DataType.Password)]

[DisplayName("Current password")]

public string OldPassword { get; set; }

[Required, ValidatePasswordLength]

[DataType(DataType.Password)]

[DisplayName("New password")]

public string NewPassword { get; set; }

[Required]

[DataType(DataType.Password)]

[DisplayName("Confirm new password")]

public string ConfirmPassword { get; set; }

}

We include validation information (the Required attribute) as well as display information (the DisplayName and DataType attributes), both of which can be used to influence final HTML generated in our templates. However, we may need more control over our HTML than what is allowed or even desired in our model class through metadata information. For example, we might want to surround some of our elements with paragraph tags. For this level of individual control where we want to lay out individual elements, we can use the EditorFor method, shown in Listing 3.14.

Listing 3.14: Using EditorFor for extra layout control

<p>

<%= Html.EditorFor(m => m.OldPassword) %>

</p>

<p>

<%= Html.EditorFor(m => m.NewPassword) %>

</p>

<p>

<%= Html.EditorFor(m => m.ConfirmPassword) %>

</p>

Since templates are shared across our site, we may not want to force every editor to include a paragraph tag. But for complex forms, we're likely to include organizational elements such as horizontal rules, fieldsets and legends to organize our elements. But for very simple display and edit models, the EditorForModel and DisplayForModel will likely meet our needs.

We've seen the new templates in action, but how exactly do they work?

3.4.2 Template locations and names

3.4.3 Customizing templates

3.5 Summary