21

Organization with Areas

This chapter covers:

* Organizing large applications with areas
* Creating links between areas
* Managing global, area-agnostic content
* Using T4MVC to help manage links and URLs

As ASP.NET MVC websites become larger and more complex, the number of controllers inevitably grows. With a large number of controllers, we start to notice many controllers might logically belong together as a group. We might have administration sections of our application, product catalog sections, customer care sections, shopping cart and ordering sections and so on. Each of these application areas will likely share nothing more than perhaps a common logon widget or a master page. However, each application area probably has quite a lot of common functionality with other controllers and views within that area. To help tame large applications, ASP.NET MVC 2 introduces the concept of areas. Areas allow us to segregate controllers, models and views into different physical locations. In this chapter, we will examine using areas to separate our application's different concerns. We will also use T4MVC templates to help us generate our URLs and links between areas.

21.1 Creating a basic Area

To create our first area, we can start by right-clicking the project in the Solution Explorer and selecting Add > Area..., as shown in figure 21.1.

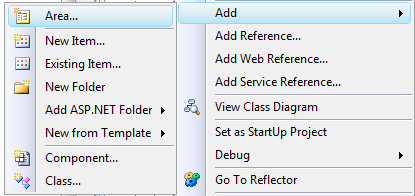


Figure 21.1 The Add Area menu option

Selecting Add Area brings up the Add Area dialog box, where we need to enter an Area name, shown in figure 21.2.

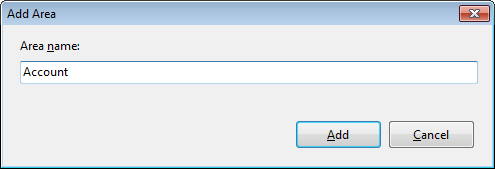


Figure 21.2 The Add Area dialog box

When the first area is created, a new top-level Areas folder is added to the MVC project. Inside this Areas folder, each individual area resides in its own folder. In each Area folder, you will find folders for controllers, models and views specific to that area. Finally, the Add Area wizard adds an area registration class. The project shown in figure 21.3 includes three areas for administration, product catalog and account information.

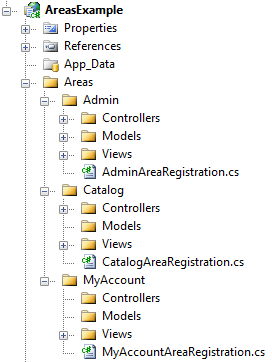


Figure 21.3 A project with three separate areas

The Add Area wizard is included with the ASP.NET MVC 2 installer, but we are not forced to use the wizard. The wizard creates the correct folder structure and area registration class. If the tooling were not available to us for some reason, we simply need to follow the same folder structure conventions. Besides the folder structure, the wizard creates an important area registration class. This class contains information describing the name and routing information pertinent to our area, and allows us to modify the default area registration information. If we used the wizard, our area registration class would similar to listing 21.1.

Listing 21.1 The default area registration class

public class AdminAreaRegistration : AreaRegistration

{

public override string AreaName

{

get

{

return "Admin";

}

}

public override void RegisterArea(AreaRegistrationContext context)

{

context.MapRoute(

"Admin\_default",

"Admin/{controller}/{action}/{id}",

new { controller = "Profile", action = "Index", id = "" }

);

}

}

Our AdminAreaRegistration class contains area registration information, and inherits from the MVC class AreaRegistration. AreaRegistration is an abstract class with one abstract property, AreaName, and one abstract method, RegisterArea. The AreaName property is used later for routing purposes. The RegisterArea method accepts a single AreaRegistrationContext object. The AreaRegistrationContext contains properties and methods which we can use to describe our area. In general, we can simply use the MapRoute method to describe the routes our area should use. In the above example, all route URLs starting with "Admin" will be directed to controllers in the Admin area.

Including route information, the AreaRegistrationContext also allows us to configure our area's type namespace. By default, the Namespaces property will contain the namespace our AdminAreaRegistration class resides. Each of the namespaces added will be used for global route registration, so that our controllers in our area-specific namespace will be chosen by the routing engine correctly. If we decide to break the convention and place our controllers in a namespace that does not reside in the same base namespace as our AdminAreaRegistration type, we would need to add these namespaces to the AreaRegistrationContext accordingly.

Once we have our AreaRegistration classes set up, we need to ensure that our areas are registered at application startup. Projects created with the default ASP.NET MVC 2 project template will have the registration code already present. If we are migrating an existing MVC 1.0 project, we will need to add the code in listing 21.2 to our Application\_Start method.

Listing 21.2 The application startup method with route and area registration

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

RegisterRoutes(RouteTable.Routes);

}

The AreaRegistration.RegisterAllAreas method scans the assemblies in the application bin folder for types derived from the AreaRegistration class, and that have a constructor with no arguments. Once we have our area registration in place, we can now add controllers, models and views to our area-specific folders. In our sample, we have administration screens related to the current user's profile. Because these might be related to other administration screens, we will place this controller and its views in the Admin area folder, shown in figure 21.4.

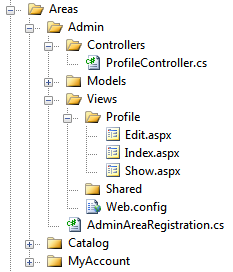


Figure 21.4 The Profile controller and views in the Admin area folder

Our ProfileController includes three actions: Edit, Index and Show. Each of its views resides in the controller-specific view folder, the Profile folder. View resolution now searches the area-specific folder first, then moves to area-specific Shared folder, and then on to the global Shared folder. Partials and master pages specific to this area can be placed in the area's Shared folder, so that they are only visible to this specific area. In this way, we can create a global master page that contains only a very general site-wide template. Each area could then include area-specific master pages used only by views in that area. If our administration screens share a common layout, we can use a master page only for our administration screens.

Each controller action does not need to specify the area name for selecting views. In listing 21.3, the Index action selects the Index view (by leaving the view name absent).

Listing 21.3 The Index action in the ProfileController

public virtual ActionResult Index()

{

var profiles = \_profileRepository.GetAll();

return View(profiles);

}

In our area registration, controllers in our area-specific namespace get a special route data token assigned, "area". This route data value is populated from our area name we specified in our area registration. When searching for views, the view engine uses this area token value to look for folders with that area name. Inside our views, we do not need to specify the area route data value for linking inside that area. In listing 21.4, we include a link in our Edit screen to link back to the list of profiles.

Listing 21.4 Linking to an action within the same controller and area

<div>

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

</div>

We only supply the action name, as the controller and area name will come from the existing route data for the current request. If we want to link to an outside area, we will need to supply that route data explicitly. In figure 21.5, the edit Profile page contains menu items, as well as a logon widget.

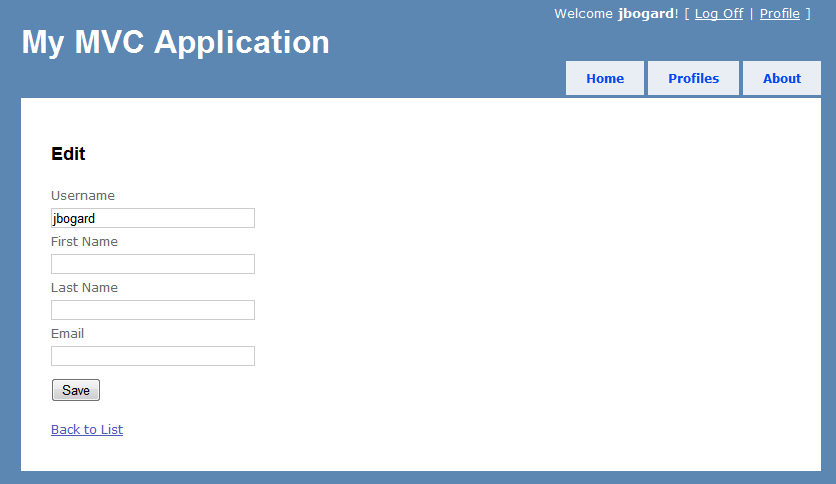


Figure 21.5 The edit profile screen with links to outside areas

The Edit action resides in the ProfileController, which itself resides in the Admin area. In the figure above, the Home and About menu items link back to the root (or default) area. Additionally, the LogOff and Profile links navigate to the root and Admin area respectively. But these items are both rendered in content residing in the root area. The Edit view inherits the Site.Master, shown in listing 21.5.

Listing 21.5 The Edit view inheriting from the global master page

<%@ Page Title="" Language="C#"

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

Inherits="System.Web.Mvc.ViewPage<EditProfileInput>" %>

In our master page, we include links to the Profile controller, as well as a logon widget that links to multiple areas. In the Edit view, we did not need to specify the area when linking back to the ProfileController's Index action, as this action was still logically in the same controller and area as the Edit view. However, we need to make the global links and widgets resilient and area-agnostic. If we did not specify the area name for the LogOff link, it will not render correctly when rendering a request in the Admin area. The generated URL would contain incorrect area information, as shown in figure 21.6.



Figure 21.6 The incorrectly generated URL containing extra area parameters

Our Account controller resides in the root Controller folder, yet was generated as if it were in the Admin area. When generating URLs in global content shared by areas, and linking outside to different areas, we need to include the area route information. In listing 21.6, our menu HTML contains area route data to ensure that the menu links correctly no matter what area the master page might be used from.

Listing 21.6 The menu HTML with area route information

<ul id="menu">

<li><%= Html.ActionLink("Home", "Index", "Home", new { area = "" }, null)%></li>

<li><%= Html.ActionLink("Profiles", "Index", "Profile", new { area = "Admin" }, null)%></li>

<li><%= Html.ActionLink("About", "About", "Home", new { area = "" }, null)%></li>

</ul>

In each ActionLink method, we specify the additional area route data for each link. The Home and About link are in the root Controllers folder, so we specify a blank area name. The Profiles link directs to the Admin area, so we need to specify the "area" route value with the AreaName of our Admin area, "Admin". The "area" route value needs to match with the AreaName used in the AdminAreaRegistration class for the URL to generate correctly. We also need to change our shared logon partial, as this partial is used across all areas. The links will now specify the areas explicitly, shown in listing 21.7 below.

Listing 21.7 Our modified logon partial including area information

<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>

<%

if (Request.IsAuthenticated) {

%>

Welcome <b><%= Html.Encode(Page.User.Identity.Name) %></b>!

[ <%= Html.ActionLink("Log Off", "LogOff", "Account", new { area = "" }, null) %> |

<%= Html.ActionLink("Profile", "Show", "Profile",

new { area = "Admin", username = Html.Encode(Page.User.Identity.Name) }, null)%> ]

<%

}

else {

%>

[ <%= Html.ActionLink("Log On", "LogOn", "Account", new { area = "" }, null)%> ]

<%

}

%>

Unfortunately, there is not an ActionLink overload that allows us to specify the area name without a RouteValueDictionary. In the next section, we will examine taking advantage of the T4MVC project to help generate route-based URLs in our application.

21.2 Managing links and URLs with T4MVC

Out of the box, ASP.NET MVC contains many opportunities to get tripped up with magic strings. Magic strings are string constants that are used to represent other constructs, but with an added disconnect that can lead to subtle errors that only show up at runtime. For example, in listing 21.8, our Edit action contains a BeginForm method call that references the Save action on the Profile controller.

Listing 21.8 The brittle Edit view with magic strings

<% using (Html.BeginForm("Save", "Profile")) {%>

<%= Html.EditorForModel() %>

<p>

<input type="submit" value="Save" name="SaveButton" />

</p>

<% } %>

The magic strings in the example above lie in the Html.BeginForm method. The strings "Save" and "Profile" are route data that refer to a ProfileController class and Save method. If we were to change the name of our controller and action via built-in refactoring tools, our Edit view would now break. Ideally, all the places where we reference controllers, actions, views and route values by magic strings could be replaced by something more resilient to the inevitable change we see in most projects. In the previous section, we saw hard-coded route data values reference "area". If we were to accidentally mistype or misspell the area route entry or value, our application would break at runtime.

To eliminate these potential issues, we have two options. We can use constants and strongly-typed, expression-based URL generation. The other option is to use a form of code generation to allow us to easily reference views, controllers and actions. The T4MVC project (<http://aspnet.codeplex.com/wikipage?title=T4MVC>) uses T4 templates to generate extension methods, view name constants, and action link helpers to easily eliminate the pesky magic strings that would otherwise litter our application.

To use T4MVC, we first need to download the latest T4MVC release and place the following two files in the root of our application:

* T4MVC.tt
* T4MVC.settings.t4

In figure 21.7, we see these two files added to the root of our MVC application.

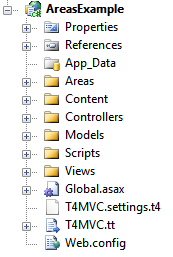


Figure 21.7 Our application including the two T4MVC template files

The T4MVC templates use the T4 (Text Template Transformation Toolkit) templating technology introduced with Visual Studio 2008. When the T4MVC templates are added to the project, or when the project is build or run, the templates are regenerated. In some environments, a security dialog may pop up, shown in figure 21.8.

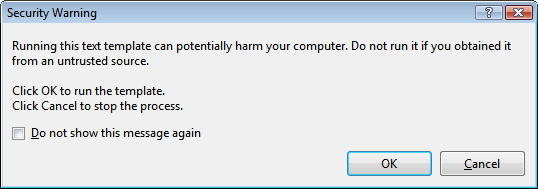


Figure 21.8 The T4 template security dialog

We can check the "Do not show this message again" if we do not want this dialog showing up again, and click the OK button to run the template generation. The T4MVC template both modifies our existing controllers, making them partial classes, and generates a set of helper files. These helper files, shown in figure 21.9, include a set of code-generated controller partial classes and extension methods.

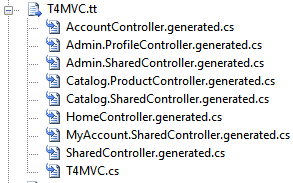


Figure 21.9 Generated helper files from the T4MVC templates

With partial classes, the T4MVC templates generate a set of helper methods and properties that allow us to easily refer to controllers, actions and views from anywhere in our application. For example, the original LogOff action in the AccountController was rife with magic strings, shown in listing 21.9.

Listing 21.9 The original LogOff action

public virtual ActionResult LogOff()

{

FormsService.SignOut();

return RedirectToAction("Index", "Home");

}

Instead of referring to the Index action on the Home controller by strings, we can instead navigate the hierarchy created in the generated MVC class, shown in listing 21.10.

Listing 21.10 Using the generated MVC class to refer to controllers and actions

public virtual ActionResult LogOff()

{

FormsService.SignOut();

return RedirectToAction(MVC.Home.Index());

}

Internally, the new RedirectToAction method lives on the generated code-beside controller class. The Index method above records the controller and action name, allowing the generated RedirectToAction method to build the correct ActionResult. All of this is behind the scenes, and our existing controllers can start using the new, generated overloads for generating ActionResult objects.

In our views, we will use some new, generated HtmlHelper extension methods for generation action links and URLs. Listing 21.11 below shows our modified logon partial.

Listing 21.22 Using the generated HtmlHelper extension methods

<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>

<%

if (Request.IsAuthenticated) {

%>

Welcome <b><%= Html.Encode(Page.User.Identity.Name) %></b>!

[ <%= Html.ActionLink("Log Off", MVC.Account.LogOff()) %> |

<%= Html.ActionLink("Profile", MVC.Admin.Profile.Show(Html.Encode(Page.User.Identity.Name)))%> ]

<%

}

else {

%>

[ <%= Html.ActionLink("Log On", MVC.Account.LogOn())%> ]

<%

}

%>

Instead of supplying the area route information manually, we navigate a logical controller hierarchy structure. The Profile controller resides in the Admin area, and the generated helper class is located in an Admin property. The MVC class hierarchy matches the area and controller layout of our project. If we were to rename an action method, we simply need to re-generate the templates and our code will be updated accordingly. The methods referring to actions also include overloads that accept the original action parameters, allowing us to easily supply route information for action parameters. The Show action accepts a "username" parameter, which we simply pass in directly.

Code generation can be quite powerful, but it does come with all the normal caveats of code generation. We need to remember to run the templates when our application changes and running the code generation takes longer as our project grows. While code generation helps prevent runtime errors, it moves them to compile-time instead of eliminating them entirely. Code generation is still not resilient to refactoring. However, T4MVC is another powerful that can eliminate much of the magic string proliferation in ASP.NET MVC applications.

21.3 Summary

Large MVC applications can become quite unwieldy to manage. To tame the natural organization that sites with many different sections and areas have, we can use the new feature of areas in ASP.NET MVC 2.0. These MVC areas allow us to segregate content into logical and physical folders, each with their own shared content hidden from other areas. For global content, we can still take advantage of global shared content. With the added flexibility of areas comes some added work with generating URLs from routes to ensure that URL generation works across areas. To help with this URL generation, we can use the T4MVC project. T4MVC uses the T4 templating technology to generate code-beside partial classes for our controllers, providing easy access to a hierarchical structure describing the controllers, actions and views in our site. In the next chapter, we will take the componentization of areas to another level with portable areas.