22

Building and distributing components using Portable Areas

This chapter covers:

* Building a Portable Area
* Embedding Views
* Distributing a Portable Area
* Creating a RSS Widget Portable Area
* Integrating with a host using the Bus

ASP.NET MVC 2's areas allow us to structure the controllers and view within our application, organizing hierchically our projects into folders and namespaces. Portable areas, a feature in MvcContrib, allow us to take that concept even further. Portable areas are like regular areas in that they are a collection of controllers and views - segmented from other areas. But they are also portable: the entire area is a separate assembly - typically deployed as a DLL file - and can be shared among several ASP.NET MVC 2 projects. In other words, areas allow us to segment our application, but portable areas allow us to compose several applications together in one project.

Imagine a common set of pages and logic that a company would want to share among all its projects. Take, for instance, the common AccountController that's generated in the default ASP.NET MVC 2 project template. AccountController provides basic authentication support - registering users and logging in and the traditional things you'd need to start accepting users. That template could be used as a starter kit for many projects, and they'd all work the same. But as it stands, the AccountController and its supporting players would be duplicated in all of them. We can instead move that stuff into a portable area which all our projects could use. We can eliminate that boilerplate code from our projects and share the new assembly instead of code files. We'll use this example to demonstrate how to use MvcContrib to create a simple portable area, gaining all the benefits of non-duplicated code.

22.1 A simple portable area

A portable area is a class library project with controllers and views. It has all the trappings of an ASP.NET MVC 2 project: controllers, folders for views and the views themselves. To extract the AccountController we'll simply move those related files from the default template to a new class library project. The overall structure of the project is the same, but it's not a web project, as shown in figure 22.1.

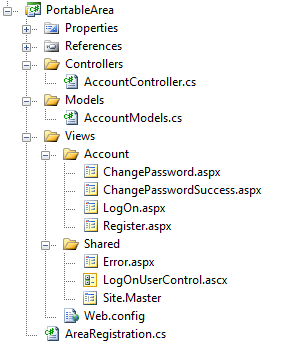


Figure 22.1 A portable area class library project

Developers familiar with the ASP.NET MVC 2 default template will recognize most of the files in the portable area shown in figure 22.1. For the most part, it's exactly the same and in the same structure. The views, however, are not content files like in ASP.NET MVC 2 projects; they are embedded resources. To make a view an embedded resource, highlight it in Solution Explorer and press the F4 key, or right-click it and select Properties from the context menu. The properties window (shown in figure 22.2) will appear.

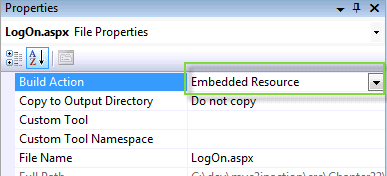


Figure 22.2 Visual Studio's file properties window

Select "Embedded Resource" to instruct Visual Studio to include the file as an embedded resource of the project.

Embedded resources

Embedded resources are project artifacts that are compiled into the assembly and they can be programmatically retrieved. Normally, views are set with a build action of "Content" which means they will be stored and accessed like regular files in the file system. Class files have a build action of "Compile", which compiles them into the assembly regularly. For more information on embedded resources, visit the MSDN reference page: http://msdn.microsoft.com/en-us/library/ht9h2dk8.aspx

Like a regular area, portable areas must be registered. Here we use a base class provided by MvcContib, PortableAreaRegistration.

Listing 22.1 Registering our portable area by deriving from PortableAreaRegistration

public class AreaRegistration : PortableAreaRegistration #1

{

public override string AreaName #2

{

get { return "login"; }

}

public override void RegisterArea

(AreaRegistrationContext context, IApplicationBus bus) #3

{

context.MapRoute(

"login",

"login/{controller}/{action}",

new { controller = "Account", action = "index" });

base.RegisterTheViewsInTheEmbeddedViewEngine(GetType()); #4

}

}

#1 Deriving from PortableAreaRegistration

#2 We still provdide AreaName

#3 RegisterArea is familiar...

#4 but we call a special method

In listing 22.1 we register our portable area. It's very similar to the regular AreaRegistration classes we wrote in chapter 21, with one additional, required step: we must call base.RegisterTheViewsInTheEmbeddedViewEngine(GetType()); (4).

That call allows us to use a special view engine (also included in MvcContrib) that makes our embedded views available to the consuming project. The embedded views are the trick behind portable areas. When our consuming project needs a view, the special embedded view engine can find them. If we didn't use this view engine, we'd have to automate our deployments so that each portable area's views were in the correct spot in our projects file system. Even though this can be automated, using embedded views allows us to skip this tedious and error prone step. In the next section we'll actually use the portable area in our consuming application.

22.2 Consuming portable areas

Once we have our portable area class library project with its controllers and embedded views, we must configure our consuming application so that it can use them. MvcContrib makes this easy. We only need one additional call in the bootstrapping code in Global.asax.cs. This is shown in listing 22.2.

Queueballs in text

Listing 22.2 Consuming a portable area in a regular ASP.NET MVC 2 project

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas(); #1

RegisterRoutes(RouteTable.Routes);

MvcContrib.UI.InputBuilder.InputBuilder.BootStrap(); #2

}

#1 Register areas normally

#2 Required for portable areas

The routing to register all areas (1) will look for any assemblies in the bin folder - if our portable area project is referenced by the consuming application it goes there automatically. If our consuming application does not reference the portable area assembly, we need to put it in the bin folder. That can be done automatically using a post-build step, configued in the build tab of the project properties.

Our consuming application must also tell MvcContrib to prepare the portable area (2). This is all that's needed to begin using the shared functionality of our portable area. In our consuming project we can link to an otherwise use portable area controllers as if they were included in our project.

22.3 Creating a RSS Widget Portable Area

Building on the previous sample of a simple Portable Area, it is important to know that the Portable Area can and should include additional helpers to make the use of consuming a Portable Area frictionless for developers. Take a Portable Area that would provide a web page widget for rendering an RSS feed as an unordered list. We will walk you through an example and show how we can add a helper to make the Portable Area easier to use.

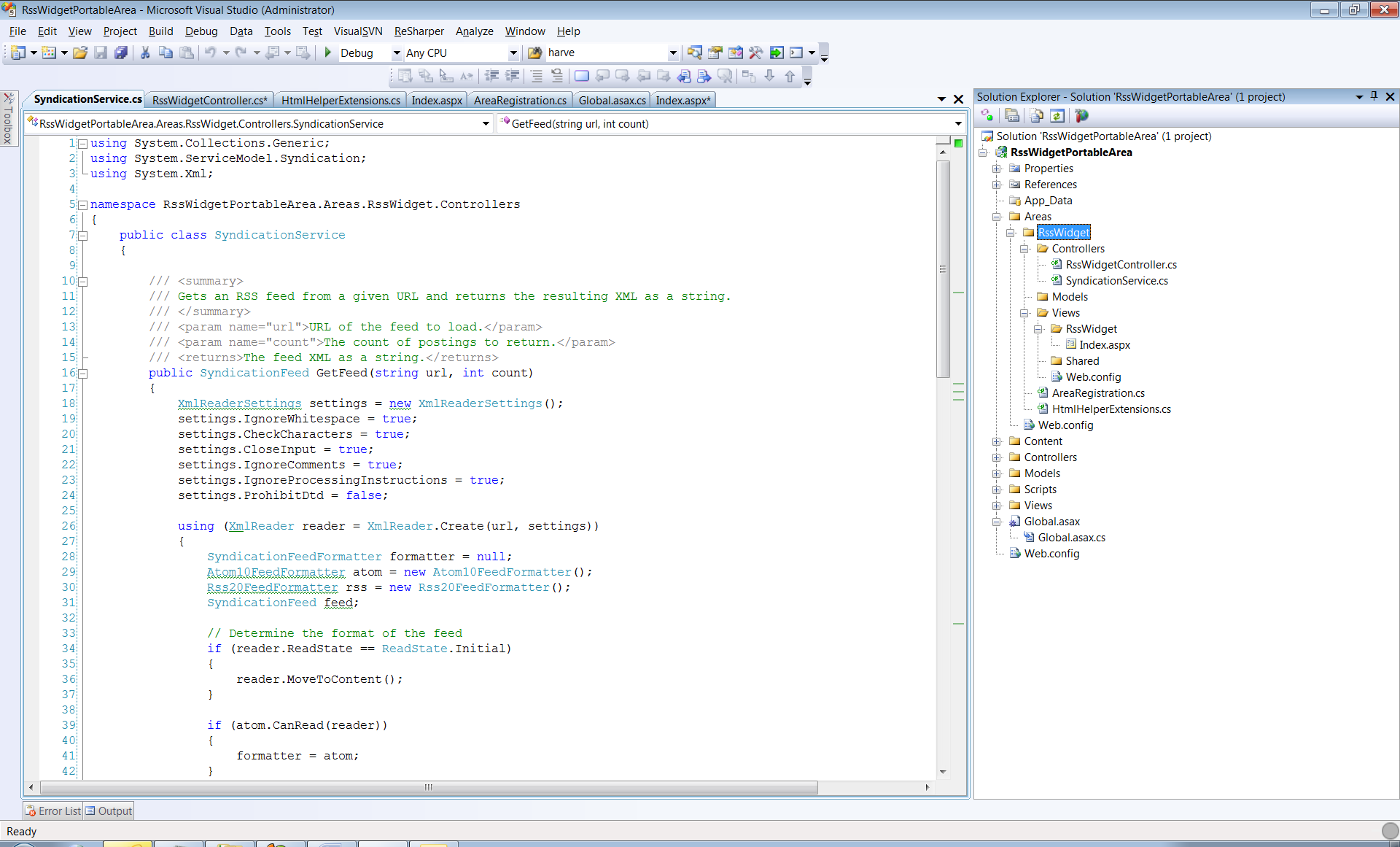


Figure 22.3 Layout of the RssWidget Portable Area.

Listing 22.3 of the RssWidget project shows all the files that are part of this Portable Area. The interesting differences between this example and the previous are the additions of the SyndicationService and the HtmlHelperExtensions classes. This examples demonstrates you can included a full vertical slice of functionality in a Portable Area. We have found that by including HtmlHelper functionality into the projects, the ease of use for the area increases significantly. Lets walk through the code.

Listing 22.3 RssWidget Registration

using System.Web.Mvc;

using MvcContrib.PortableAreas;

namespace RssWidgetPortableArea.Areas.RssWidget

{

public class RssWidgetAreaRegistration : PortableAreaRegistration

{

public override string AreaName #1

{

get { return "RssWidget"; }

}

public override void RegisterArea(AreaRegistrationContext context, IApplicationBus bus)

{

context.MapRoute( # 2

"RssWidget\_default",

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

new {action = "Index", id = ""});

base.RegisterTheViewsInTheEmbeddedViewEngine(GetType()); #3

}

}

}

#1 - The AreaName sets the name of the Area

#2 - The routes for the Area are mapped

#3 - The embedded views are registered

Queueballs in text

The registration code for the area, in listing 22.3, is boiler plate code. The standard calls to (2) MapRoute and (3) RegisterTheViewsInTheEmbeddedViewEngine are included. There is no special registration code needed for this sample. There is only one Action included in this Portable Area. It is the RssWidgetController.Index method. This method is really basic. Its only purpose is to tie together the RssUrl and the SyndicationService dependency. See Listing 22.4 to see the details of the Index method. The SyndicationService provides the logic to retrieve a Rss Feed from a url and return the model of the feed. The controller then sends that model to the view for formatting.

Listing 22.4 RssWidgetController

using System.Web.Mvc;

namespace RssWidgetPortableArea.Areas.RssWidget.Controllers

{

public class RssWidgetController : Controller

{

public ActionResult Index(string RssUrl)

{

return View(new SyndicationService().GetFeed(RssUrl, 10)); #1

}

}

}

#1 The GetFeed method of SyndicationService is called.

Once the feed model is sent to the view there is a really simple view, see listing 22.5, that will create an unordered list of the rss feed. The code is pretty simplistic in this view. It loops over a collection of System.ServiceModel.SyndicationSyndicationFeed objects and displays the Title and Author for each item. If a developer needed to control the Html for this widget, the great part about a Portable area is that they can just overide this view and still take advantage of the Controller and SyndicationService that is provided by the component. So using the Portable Area is not an all or nothing decision, because the Portable Area is built on top of the MVC 2 Areas implementation it is easy to start taking control back from the component and providing your own implementation code. This can be considered incremental customization.

Listing 22.5 View for the RssWidget.Index action

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

Inherits="System.Web.Mvc.ViewPage< System.ServiceModel.Syndication.SyndicationFeed>" %> #1

<ul> #2

<%foreach(var item in Model.Items) {%>

<li><%=item.Title.Text %> - <%=item.Authors[0].Name %></li> #3

<%} %>

</ul>

#1 - The Model of the View is set

#2 - Loop over the SyndicationFeed items

#3 - Render the list items with Title and Author

The experience for a developer using this RssWidget Portable Area is where this type of component model really shines. Using this widget in an application consists of referencing the html helper extensions from your view and then calling the RssWidget method.

Listing 22.6 Calling a RssWidget HtmlHelper extension

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

<%@ Import Namespace="RssWidgetPortableArea.Areas.RssWidget"%> #1

<asp:Content ID="indexTitle" #2

ContentPlaceHolderID="TitleContent" runat="server">

Home Page

</asp:Content>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">

<% #3

Html.RssWidget("http://search.twitter.com/search.atom?q=%23mvc2inaction");

%>

</asp:Content>

#1 The html helper namespace is referenced

#2 Existing view content

#3 RssWidget is called into the Portable Area

The only line of code in the application that calls the Portable Area is the call #3 to the RssWidget method. After calling that method and running a simple view that reference a Rss feed from Twitter to search for the MVC2InAction from the online service Twitter.com, the resulting webpage will be displayed. See figure 22.4. The title and tweeter will show up on the screen.

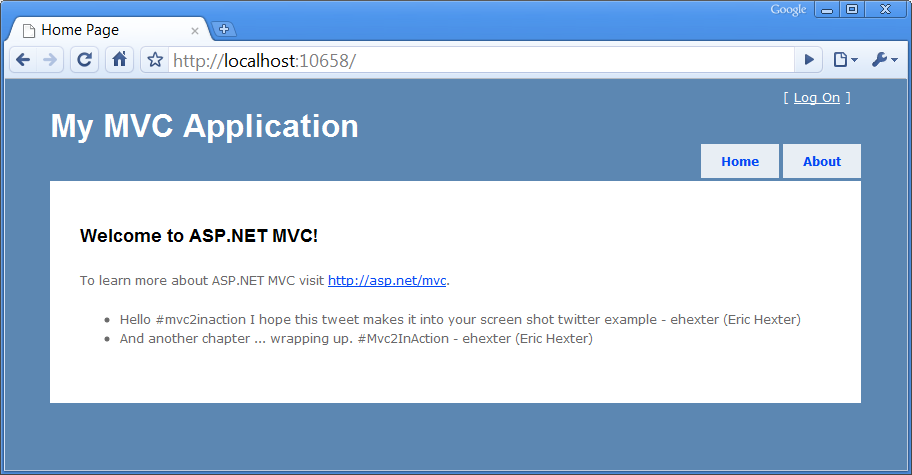


Figure 22.4 The view that uses the RssWidget Portable Area

The RssWidget html helper method that is used in the applications view, is the syntactic sugar that really makes consuming this Portable Area really simple. If this method was not made available, then developers using the Portable Area would need to know about some of the internals of how the Area was constructed. For example, the way the Area is intended to be used, in this case the RssWidget was intended to be used with the RenderAction method calling the RssWidgetController's Index method. In order to make that call the Area name registered in the Area's registration is required, in this case the Area name is RssWidget.

Listing 22.7 Hiding complexity in a HtmlHelper extension method.

using System.Web.Mvc;

using System.Web.Mvc.Html;

namespace RssWidgetPortableArea.Areas.RssWidget

{

public static class HtmlHelperExtensions

{

public static void RssWidget(this HtmlHelper helper, string RssUrl)

{

helper.RenderAction("Index", "RssWidget", #1

new {RssUrl, Area = "RssWidget"});

}

}

}

#1 - Call to RenderAction requires knowledge of the Areas internals

The HtmlHelper extension method, displayed in Listing 22.7, shows a call to RenderAction which could easily be put into an applications view, to call into the Portable Area, but this call does required knowledge about the internals of the Area. By moving this code into a Html Helper extension method, all of that Portable Area specific code can be pushed into the Portable Area. By doing this the developer using the Area just needs to worry about where the widget should be displayed in the application and what rss url needs to be displayed. Making this separation of concerns allows the Portable Area developer the flexibility to make internal changes to the implementation while leaving the public facing interface nice and simple.

22.4 Distributing the RSS Widget

So far we covered how to create the widget as well as how to use it from a MVC application. The one missing piece is distributing the RSS Widget Portable Area. This entire component was written in a way that allows it to be compiled down to one file. The in order to use this Portable Area from an MVC Application, the application just needs the Portable Area in its bin directory. So distributing the Portable Area just consists of distributing the dll. In order to really do the right thing, we recommend distributing Portable Areas through a zip file and that package should include the assembly, a readme file that explains what the Portable Are is intended to do, a sample application that shows how to use the Portable Area, and last but not least, you should include a license file. The license file is normally overlooked by developers but by selecting a license, either open source or closed source. It makes it very clear to anyone downstream of how the Portable Area creator intended to share the Area. We do not see Portable Areas being something that is tied to just open source or component vendors exclusively. The concept really demonstrates the technical solution to making sharing functionality easy. We see this as being very interesting to both open source and closed source, developers and companies.

22.5 Interacting with the Portable Area Buss

The samples that we have covered so far have solved some pretty specific problems. These examples have been able to take little input from the hosting application and provide some useful benefits. In most cases, a Portable Area will need to programmatically interact with the hosting application. Rather than leaving the method for doing this up to each Portable Area developer, the MvcContrib project laid out a very simple but effective manor for enabling this communication. The mechanism is a message bus. Specifically, the bus was created to allow synchronous communication to send and receive messages that the Portable Area defines. If there was a login Portable Area only provided the user interface and did not provide its own data store for looking up username and passwords, than it could send a message on the bus and the hosting application could than look up a username in its custom user data store as well as compare the password and then return the message letting the Portable Area know if the user credentials are valid. Let's look at how a message is sent from a Portable Area. A call to send a message down the buss is displayed in Listing 22.8.

Listing 22.8 Sending a message to the host through the Buss

MvcContrib.Bus.Send(new RssWidgetRenderedMessage{Url = RssUrl}); #1

#1 - Send a message to the buss.

This example shows one way message being sent to an application, say for logging purposes. In order for a message to be received the host application needs to register a handler.

Listing 22.9 Registering a message handler

MvcContrib.Bus.AddMessageHandler(typeof(RssMessageHandler)); #1

#1 This would be called during startup

Registering a message handler is a one line call that should only happen once in an application. This code should be called at the application startup. The Buss will keep track of the handlers and messages and make sure the handlers are called when needed. The code that is more interesting is that RssMessageHandler class. Each message handler needs to be implemented in the host application. Handlers should be considered integration code, to stitch together a Portable Area with the host application. This means that the handler code should be minimized and rely on application service classes rather than implementing lots of logic inside of a handler class.

Listing 22.10 A Message Handler class

using MvcContrib.PortableAreas;

using RssWidgetPortableArea.Areas.RssWidget.Controllers;

namespace RssWidgetPortableArea

{

public class RssMessageHandler : MessageHandler<RssWidgetRenderedMessage> #1

{

public override void Handle(RssWidgetRenderedMessage message) #2

{

//log the message to the applications log.

}

}

}

#1- Inherit from MvcContrib MessageHandler

#2 - Implement the Handle method

Listing 22.10 demonstrates the boilerplate code required to implement a message handler for a message using the bus. Inside the Handle method you can implement calls to your application services and data storage

Summary

This chapter walked you through the creation of Portable Areas. The biggest benefit that a Portable Area can provide over a standard area is the ability to distribute the Portable Area as a single assembly. Using this mechanism can allow you to build some reusable components, in a easy way. The are some other mechanisms that allow your application to provide functionality faster, we will show how using NHibernate can increase your teams productivity. The next chapter covers using NHibernate for your applications data access.