6

Hosting ASP.NET MVC Applications

This chapter covers:

* Understanding server environment requirements
* Revealing hosting options in IIS
* Configuring different environments
* Creating push-button deployments

Running an ASP.NET MVC application in Visual Studio is as easy as hitting F5, but what about deploying the application? In a Windows-hosted environment, web applications are typically deployed to Internet Information Services (IIS). But several different versions of IIS are on the market, each with different configurations and options for hosting an ASP.NET MVC application. With new features like routing, in some versions of IIS, hosting presents new challenges that did not exist with Web Forms applications.

Beyond server environment and hosting scenarios, deploying an application presents an entirely different set of challenges. Manual deployments are wrought with problems, as human errors become more prevalent. Automation eliminates these eleventh-hour problems by removing the human errors from deployments by letting the computer perform the scripted, repetitive tasks and allowing the human to focus on monitoring and testing the deployment. Each deployment environment is slightly different, because connection strings, configuration settings, and server environments can vary. By introducing change management into our automated deployment process, we can ensure we install the correct application with the correct environment settings.

In this chapter you will learn options for hosting in the different IIS versions supported today. The reader will learn how to simplify deployment through an xcopy deployment strategy, and automate deployment through build automation tools. With these build automation tools, the reader will see how to take advantage of configuration management to automate configuration changes to the various deployment environments.

6.1 Deployment scenarios

In most scenarios, deploying an ASP.NET MVC application involves deployment to a Windows Server OS environment. Occasionally, it is necessary to deploy to older environments such as Windows Server 2000 or Windows XP, with older versions of IIS. Table 6.1 shows Windows OSs and the version of IIS available.

Table 6.1 Windows and IIS versions

|  |  |
| --- | --- |
| Windows Operating System | IIS Version |
| Windows 2000 | IIS 5.0 |
| Windows XP Professional | IIS 5.1 |
| Windows XP Professional x64 Edition | IIS 6.0 |
| Windows Server 2003 | IIS 6.0 |
| Windows Vista | IIS 7.0 |
| Windows Server 2008 | IIS 7.0 |
| Windows 7 | IIS 7.5 |
| Windows Server 2008 R2 | IIS 7.5 |

For all practical purposes, there are only two types of hosting environments we need to worry about:

* IIS 7.0+
* Not IIS 7.0+

Deploying to an IIS 7.0 environment to support the routing features of ASP.NET MVC requires far less configuration than the older versions of IIS. Most of the configuration decisions for IIS 6 and older versions revolve around routing, where your deployment decision could affect how you configure your routes.

Before we look at IIS deployment options, let’s look at the hosting requirements for an ASP.NET MVC application. In addition to having IIS installed, the target machine will need to have the following software installed:

* .NET Framework 3.5
* .NET Framework 3.0
* .NET Framework 2.0

When installing on .NET 3.5 without service pack 1 (SP1), be sure to deploy System.Web.Abstractions.dll and System.Web.Routing.dll. .NET 3.5 SP1 includes these two new assemblies in the GAC. Next, we’ll see how to deploy to an IIS environment using XCOPY deployment.

6.2 XCOPY deployment

Regardless of the version of IIS used, not every file in your solution needs to exist in the final destination on the server. Those familiar with Web Forms deployments know not to deploy code-behind files. The same holds true for MVC deployments. For an MVC-only website, the files needed are:

* Global.asax
* Web.config
* Content files (JavaScript, images, static HTML, etc.)
* Views
* Compiled assemblies
* MVC assemblies
* System.Web.Abstractions.dll (not needed with .NET 3.5 SP1)
* System.Web.Mvc.dll
* System.Web.Routing.dll (not needed with .NET 3.5 SP1)

Deployments themselves can be difficult. Add complexities like installers, and deployments can become even more difficult to execute and maintain. Installers usually need a person logged in to the target machine to run them, and automation of installers is possible but still difficult. Log files from a botched installation usually consist of output from the MSI logger, which can be extremely verbose and indecipherable. While there is still no deployment solution built into the .NET Framework, you will mitigate many of these difficulties by scripting your deployments.

For many application deployment scenarios, an installer is unnecessary. Assuming the target machine is already configured correctly, simply copying over files is sufficient to deploy the application. This type of deployment is called “XCOPY deployment.” The term originated from the XCOPY DOS command, which allowed copying of multiple files in one command, along with many other options.

XCOPY deployment can significantly reduce the complexity of a deployment, as no one needs to perform a manual installation on the target server. Although the term XCOPY refers to a specific DOS command, any technology that copies files also applies.

Choosing an installation strategy

Although an XCOPY deployment is the simplest choice, it’s not always the right choice. XCOPY deployments are designed to copy files to the destination machine, and nothing more. Some IT environments require a specific deployment technology for a variety of reasons, such as traceability, logging, and reversibility.

XCOPY deployments work well for most web scenarios, but provide no out-of-the-box “uninstall” capabilities. Although other mechanisms exist to roll back an installation, some IT governance teams prefer the reliability of an installer for rolling back changes.

In practice however, an installer is only as good as the developer who created it. It is still important to have test environments to ensure the installer works before trying it in production.

Modern installer products allow endless customization, such as IIS configuration, SQL configuration, and custom actions. The learning curve for these types of products is not trivial, leaving many teams to assign one member to be the installer developer. If this person leaves the team for any reason, often both the installer tool and the actions it performs need to be entirely rediscovered and relearned.

As mentioned earlier, XCOPY deployments do not have to use a specific technology. Batch files, NAnt scripts, MSBuild scripts and third-party products such as FinalBuilder are all popular choices for creating XCOPY deployments. Particularly appealing are the latter choices, which include features that assist in automated deployments. Later in this chapter, we’ll look at taking advantage of NAnt to perform deployment tasks, in addition to copying files. But first, let’s look at deploying an ASP.NET MVC application to an IIS 7 environment.

6.3 Deploying to IIS 7

Before we look at automating our deployments, we need to configure our server to host an ASP.NET MVC website. An MVC website needs a location on the target machine’s hard drive. For this book, the location is unimportant, so we’ll choose something simple, “C:\websites\MVCSample.” Our sample application will have no dependencies on a database, but later we will look at how to incorporate a database into our deployment strategy.

Our controller for this sample application will be simple but incorporate some common routes, as shown in Listing 6.1.

Listing 6.1 Our simple controller

public class ProductController : Controller

{

private static readonly Product[] Products = #A

new[]

{

new Product {Id = 1, Name = "Basketball",

Description = "You bounce it."},

new Product {Id = 2, Name = "Baseball",

Description = "You throw it."},

new Product {Id = 3, Name = "Football",

Description = "You punt it."},

new Product {Id = 4, Name = "Golf ball",

Description = "You hook or slice it."}

};

public ActionResult List() #B

{

ViewData["Products"] = Products;

return View();

}

public ActionResult Show(int id) #C

{

var product = Products.FirstOrDefault(p => p.Id == id);

ViewData["Product"] = product;

return View();

}

}

#A Dummy list of products

#B Parameterless action

#C One parameter, from RouteData

Navigating to the List action renders the screen shown in Figure 6.1.

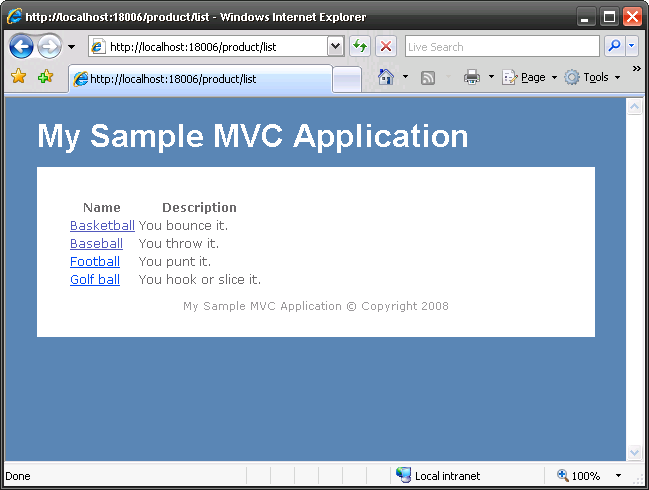


Figure 6.1 Running the MVC application locally allows us to use “pretty” URLs, with no extensions.

To deploy this ASP.NET MVC application to an IIS7 box, we’ll first create a local folder and move all our deployment files over. For this sample application, the folder structure is:

The following will be converted to a screenshot before typesetting

* C:\Websites\MVCSample
* \bin
* Iis7DeploymentSample.dll (our compiled application assembly)
* System.Web.Abstractions.dll \*only needed when .NET 3.5 SP1 is NOT installed
* System.Web.Mvc.dll
* System.Web.Routing.dll \*only needed when .NET 3.5 SP1 is NOT installed
* \Content
* Site.css
* \Views
* \Product
* List.aspx
* Show.aspx
* \Shared
* Error.aspx
* Site.master
* Web.config
* Default.aspx
* Global.asax
* Web.config

When content is in place, we can configure a new website in the IIS Manager by clicking Add Web Site…, as shown in figure 6.2.

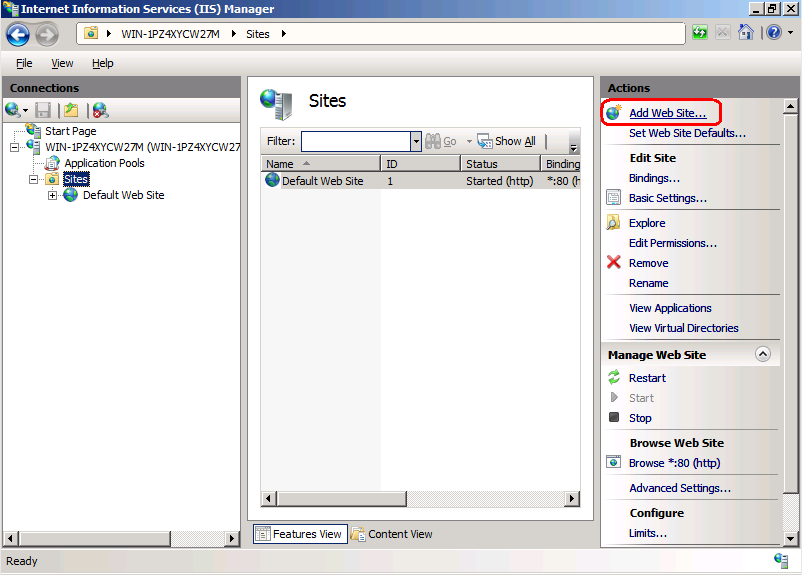


Figure 6.2 Add Web Site in the IIS 7 Manager console

In the Add Web Site dialog that comes up, we’ll need to configure the:

* Site name
* Application pool
* Physical path
* Binding

For the Site name, I chose an arbitrary name that did not exist, “MVCSample.” In the Application pool dialog, any application pool will suffice as long as it is configured as a .NET 2.0 application pool. In IIS 7/7.5 it is preferred to use Integrated mode, although with a wildcard mapping, Classic mode can be made to work as well. ASP.NET MVC is not supported to run on lower versions of ASP.NET. We won’t look at application pool strategies, but with IIS6 onward, IIS supports multiple websites, each with a shared or individual application pool. The Physical Path will point to our C:\Websites\MVCSample directory. Finally, I chose simply to bind to port 81 for this website. You can choose any unused port. Typically in production scenarios, the Host name would be configured. The final configuration values are shown in figure 6.3.

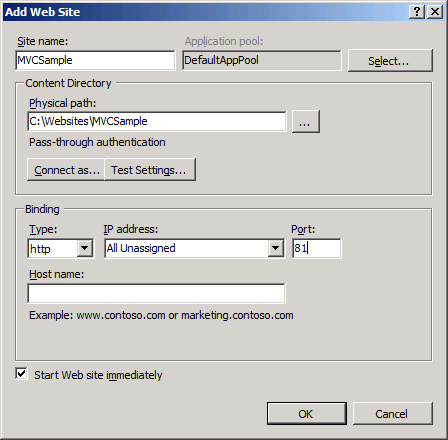


Figure 6.3 Final configuration values for the IIS 7 MVC deployment

Now that our website is configured and started, we can navigate to our MVC application, as seen in figure 6.4.

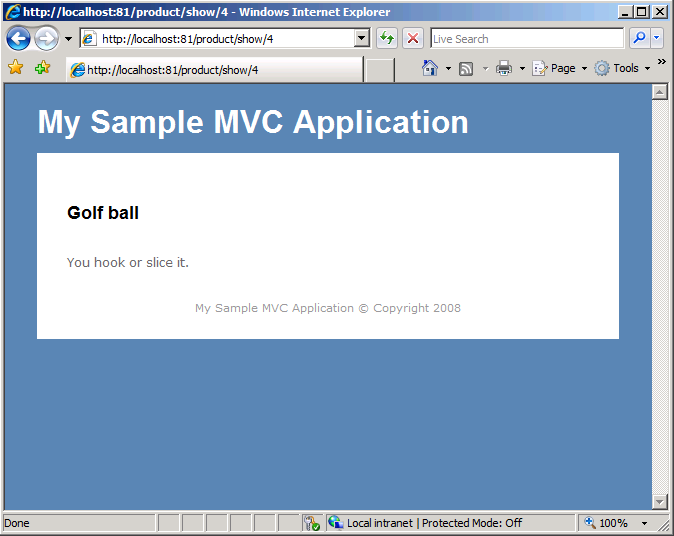


Figure 6.4 MVC application deployed in IIS 7

Besides extra configuration steps, such as security or binding, we did not have to perform additional steps to get our MVC application running under IIS 7. The new managed architecture of IIS 7 allows us to have simple deployments. Additionally, our URLs look exactly the same as they did when running locally out of Visual Studio, without .aspx or other extensions. IIS 7 supports “pretty” URLs out of the box, with no configuration necessary.

In the next section, we’ll examine configuration options available in IIS 6/5, and how we can achieve the same effect of pretty URLs.

6.4 Deploying to IIS 6 and earlier

When we deploy our MVC application to IIS 6 and earlier, we can consider a few options concerning routes. IIS 6 and earlier use ISAPI filters, which map file extension requests to ISAPI handlers. Extensions, such as .aspx and .ascx, map to the ASP.NET ISAPI handler, but extensions in the pretty, extension-less MVC URLs do not. By the time ASP.NET handles the request, IIS has already chosen an ISAPI handler for the request, and the selection may not be ASP.NET. Unfortunately, developing custom ISAPI filters requires C/C++ knowledge. Although some open source projects exist for writing managed ISAPI filters, it is not as easy as creating a custom IHttpHandler or IHttpModule implementation.

Out of the box, ASP.NET MVC applications will not work in IIS 6. Getting an MVC application to run successfully in an IIS 6 environment requires either changes to our routes or extra configuration steps in IIS. Our four choices for deploying to IIS 6 are:

* Configure routes to use the .aspx extension
* Configure routes to use a custom extension (such as .mvc)
* Use a wildcard mapping with selective disabling
* Use URL rewriting

The last choice offers the most flexibility, but does require the use of third-party software. Each option requires more configuration in IIS, which may not be available in your deployment environment. First, let’s look at the easiest deployment option and configure our routes to use the .aspx extension.

6.4.1 Configuring routes to use the .aspx extension

When we install ASP.NET in IIS, by default the aspnet\_isapi.dll ISAPI filter is set up to handle requests to .aspx extensions. By configuring our routes to use the .aspx extension, we’ll avoid needing to configure extra mapping settings in IIS for our MVC application. To configure our routes to use the .aspx extension, we need to change the default route configuration to look like listing 6.2.

Listing 6.2 Route configuration with the .aspx extension

routes.MapRoute(

"Default",

"{controller}.aspx/{action}/{id}", #A

new { controller = "Product", action = "List", id = "" }

);

#A IIS7 deployments don’t need extension

After the {controller} element, we insert the .aspx extension into the route configuration. Note that the extension is outside the brackets, and before the first backslash. Deploying the application with the route configuration changes produces the result shown in figure 6.5.

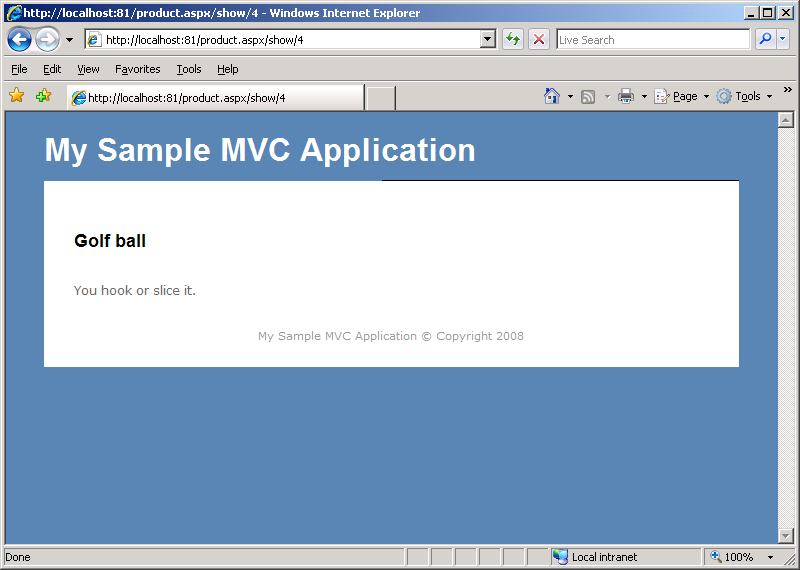


Figure 6.5 Using the .aspx configuration produces modified URLs

Unfortunately, using this deployment option produces ugly, unintuitive URLs. Note the URL, http://localhost:81/product.aspx/show/4, now has the extension immediately after the controller name. For those accustomed to extensions at the end of the URL, this URL can be confusing. Although we did not have to perform any additional configuration in IIS, the outcome is an ugly URL. The strategy introduced in chapter 6 for actions serving multiple formats (XML and JSON) becomes more challenging, as IIS may or may not have these extensions routing to ASP.NET. One of the benefits of using MVC over Web Forms is pretty URLs, which have now been lost with this deployment strategy. Our next option is to use a custom extension, which introduces a slight cosmetic change to the resulting URLs.

6.4.2 Configuring routes to use a custom extension

Instead of mapping our routes to the .aspx extension, a custom extension could reduce the confusion of users accustomed to Web Forms URLs. We’ll configure our routes to use the .mvc extension instead of .aspx, as seen in listing 6.3.

Listing 6.3 Route configuration using the custom .mvc extension

routes.MapRoute(

"Default",

"{controller}.mvc/{action}/{id}",

new { controller = "Product", action = "List", id = "" }

);

This configuration differs from the previous .aspx route configuration in the extension only. When it comes to deploying this route configuration, we need to perform additional steps in IIS. Since IIS is not configured to handle requests from the .mvc extension, we’ll need to add a mapping that will enable the ASP.NET ISAPI filter to handle the .mvc extension. To map the new extension, follow these steps and as shown in figures 6.6 and 6.7:

1. Create the website with the default configuration.
2. In the Home Directory tab in the Properties dialog for the website, click Configuration….

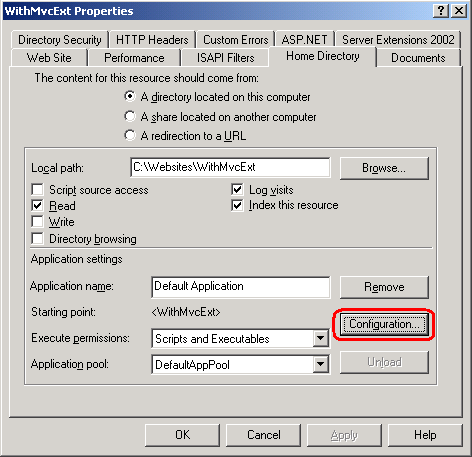


Figure 6.6 Website properties dialog

1. In the Mappings tab in the Application Configuration dialog, click Add….
2. In the Add/Edit Application Extension Mapping dialog:
3. Set the Executable value to the path to the aspnet\_isapi.dll. This is typically at C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\aspnet\_isapi.dll. Use the .NET 2.0 version of the dll.
4. Set the Extension value to .mvc. Make sure the extension has the leading dot.
5. Select All verbs in the Verbs section. If you know the HTTP verbs you wish to support, provide a comma-separated list of the verbs in the Limit to section.
6. Uncheck the Verify that file exists option. The requested URLs will not map to a location on disk, and IIS responds with a 404 if you don’t uncheck this value.

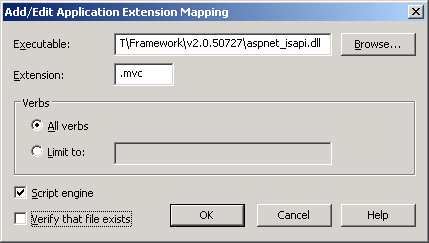


Figure 6.7 Configuration values for the new .MVC IIS extension mapping

1. Click OK on all of the configuration dialogs.

Now that we have configured IIS to allow ASP.NET to handle requests for the .mvc extension, we can use the MVC application. Our new URL is http://localhost:82/product.mvc/show/4, which is only a slight cosmetic change from the previous option. Although using the .mvc extension might prevent some users from getting confused between Web Forms .aspx URLs and .mvc URLs, these new URLs still go against normal URL conventions. In normal URL conventions, only querystring parameters follow an extension. Instead of using a custom extension, our next option uses a wildcard mapping.

6.4.3 Using wildcard mapping with selective disabling

With the next two options, we won’t have to perform any special route configuration. In fact, we can deploy the same MVC application to both IIS 7 and IIS 6 and previous versions with the wildcard mapping option. We no longer need an extension in our route configuration, and the URLs used for development will be identical to the URLs used for production on IIS 6.

With wildcard mapping, all requests are routed to a single ISAPI filter. We’ll configure the aspnet\_isapi.dll filter to be this single filter. To create the wildcard mapping:

1. Create the website with the default configuration.

2. In the Home Directory tab in the Properties dialog for the website, click Configuration….

3. In the Mappings tab in the Application Configuration dialog, click Insert….

4. In the Add/Edit Application Extension Mapping dialog:

a. Set the Executable value to the aspnet\_isapi.dll path. The path is typically C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\aspnet\_isapi.dll. Use the .NET 2.0 version of the dll.

b. Uncheck the Verify that file exists option and ensure the configuration matches that shown in figure 6.8.

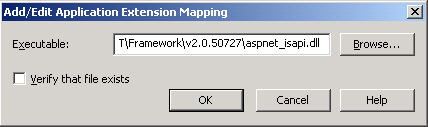


Figure 6.8 Configuring wildcard mapping to map to ASP.NET

5. Click OK on all configuration dialogs

After this configuration change, we can navigate to our MVC application, without special extensions. Our URL is now http://localhost:81/product/show/4, matching the URL that we see in IIS 7 deployments. This wildcard mapping has one unfortunate side effect: all requests are now handled by ASP.NET, which does not perform as well as IIS for many file types. For example, static files such as images, CSS, and JavaScript files now pass through ASP.NET.

We can configure subdirectories to remove the wildcard mapping. Because all static content for deployed websites usually exists in subdirectories like “Content,” “Scripts,” and others, we can perform extra configuration steps to allow IIS to handle these static files, instead of IIS. Figures 6.9, 6.10, and 6.11 illustrate some of the steps. For each subdirectory, we’ll need to:

1. Right-click the subfolder and click Properties in the IIS Management Console.

2. In the Directory tab in the Properties dialog, click the Create button. This will create an application for this folder, and will enable the Configuration… button.

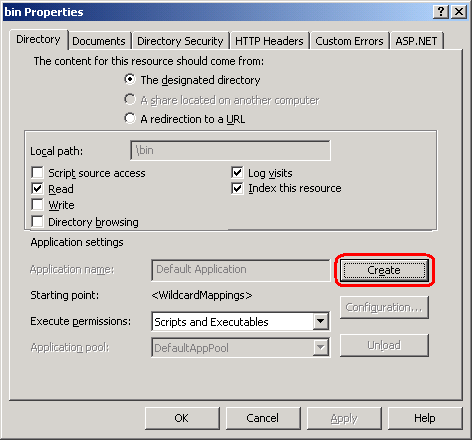


Figure 6.9 Creating an application for a subfolder temporarily

3. In the Directory tab in the Properties dialog, click the Configuration… button.

4. In the Mappings tab of the Application Configuration dialog, click the Remove button in the Wildcard application maps section. This will remove the wildcard mapping we configured at the root earlier.

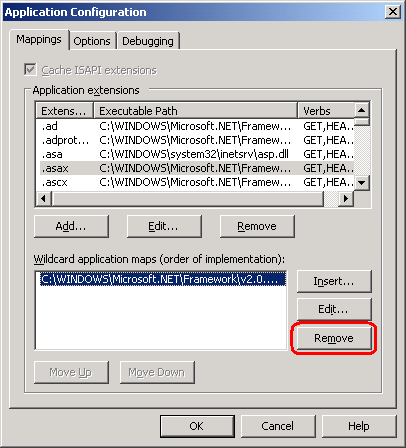


Figure 6.10 Removing the wildcard mapping from a subfolder

5. Click OK to return to the Application Configuration dialog.

6. In the Directory tab in the Application Configuration dialog, click Remove. This will remove the Application from the subfolder.

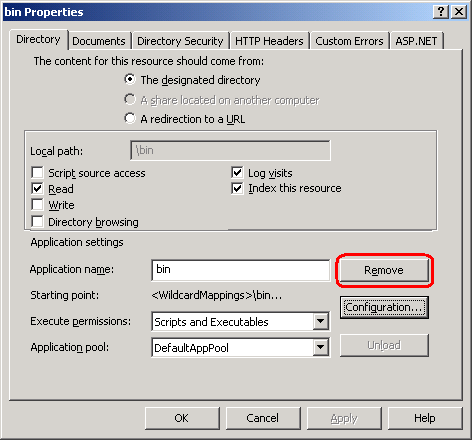


Figure 6.11 Removing the application from the subfolder

7. Click OK on all configuration dialogs.

When you repeat these steps for each subfolder, you prevent IIS from using the wildcard mapping in these subfolders. Because the only way to enable the Configuration… button is to create an application, we have to temporarily configure the subfolder as an application. Removing the application after configuration does not remove our custom configuration, however. Our changes are safe, although we had to perform extra temporary configuration to get there.

Although this option requires a bit of configuration in IIS, it does not require any additional software. Our route mappings do not need to change, and we get to keep our pretty, extension-less URLs. Whenever we add another subfolder, we’ll need to repeat the extra configuration steps to ensure ASP.NET does not handle requests it does not need to. Sometimes, we need more control over our URLs than IIS 6 and earlier versions allow right out of the box. In the next section, we’ll look at URL rewriting to handle both MVC requests and additional URL rewriting scenarios.

6.4.4 Using URL rewriting

URL rewriting is a sizable topic, which covers resource management support, search engine optimization, and canonicalized URLs. In many other web application servers, URL rewriting is a first-class, built-in feature or easily configured and customizable add-on. In IIS 6 and earlier, there was no built-in URL rewriting ability. For IIS 7, Microsoft released an HttpModule that allowed configuration directly from the IIS Manager. Regardless of the version of IIS used, URL rewriting is a vital function for many websites.

Why should I care about URL rewriting?

URL rewriting is a general term for the ability to intercept URL requests and transform them. For resource management, such as RSS links, URL rewriting can permanently redirect requests to the new RSS URL, while remaining transparent to the subscribers. In many ASP.NET websites, many URLs point to the same page. For example, all of the following URLs resolve to the same page:

* http://codeplex.com
* http://codeplex.com/
* http://codeplex.com/default.aspx
* http://www.codeplex.com
* http://www.codeplex.com/
* http://www.codeplex.com/default.aspx

Yet they all resolve to different URLs, with a couple exceptions. Differing URL resolution has the potential to lower search engine results, as many pages point to the same content. With URL rewriting, all of the example URLs can be redirected to one canonical URL. With URL rewriting, we can not only allow extension-less routes in our MVC application, but set ourselves up for further vital URL rewriting scenarios.

Since URL rewriting is not available for IIS 6 and earlier out of the box, we’ll need to use a third-party extension for rewrites. Two popular URL rewrite ISAPI extensions are:

* Helicon Tech’s ISAPI Rewrite - <http://www.isapirewrite.com/>
* Ionic’s ISAPI Rewrite - <http://www.codeplex.com/IIRF/>

Helicon Tech has one free and one fully supported edition of its product. The Ionic extension is free and open source, so we’ll configure our application using that.

First, we’ll need to download the latest version of the filter from CodePlex. Once we have the latest binaries, we are ready to configure our MVC application to use the ISAPI Rewrite module. The general idea behind our URL rewriting strategy is to:

* Configure ISAPI rewrite to add a .mvc extension to our URLs.
* IIS will see a request for .mvc, and hand it off to ASP.NET.
* Configure our web application to remove .mvc extensions.

Because our web application removes the .mvc extension before the MVC route handler processes the request, we won’t need to change our routing configuration. To configure ISAPI rewrite, we’ll need to:

1. Modify our web application to remove the .mvc extension at the beginning of the request. We can place the code in listing 6.4 in a custom HTTP Module.

Listing 6.4 Removing the .aspx extension on each request

Public class IIS6ExtensionRewriteModule : IHttpModule

{

public void Dispose()

{

}

public void Init(HttpApplication context)

{

context.BeginRequest += context\_BeginRequest;

}

void context\_BeginRequest(object sender, EventArgs e)

{

string url = "~" +

HttpContext.Current.Request.Url.PathAndQuery;

if (url.Contains(".mvc")) #A

{

string newUrl = url.Replace(".mvc", "");

HttpContext.Current.RewritePath(newUrl);

}

}

}

#A Only requests for .mvc resources

2. Wire up the HTTP Module to our application by adding the following line to our web.config under the system.web/httpModules section:

<add name="IIS6ExtensionRewriteModule"

type="SampleIIS6WithISAPIFilter.IIS6ExtensionRewriteModule,

SampleIIS6WithISAPIFilter"/>

3. Create the website with the default configuration, and deploy our application as normal.

4. Create a folder to hold the ISAPI extension. We’ll use “C:\inetpub\isapirewrite”.

6. Copy the IsapiRewrite4.dll to the newly created folder.

6. In the newly created folder, create an IsapiRewrite4.ini file and add the contents in listing 6.5. Save this file when you have finished editing it.

Listing 6.5 The URL rewriting rules

RewriteRule ^(?!/Content)(/[A-Za-z0-9\_-]+)(/.\*)?$ $1.mvc$2 [I]

7. Open the Properties dialog for the website containing the MVC application in IIS Manager

8. In the ISAPI Filters tab in the Properties dialog, click Add….

9. Enter a name for the Filter name value, and the path to the IsapiRewrite4.dll for the Executable value as shown in figure 6.12.

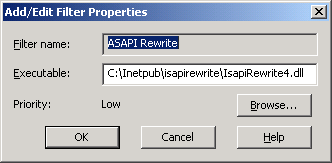


Figure 6.12 Configuring the ISAPI Rewrite filter

10. Click OK on all of the IIS configuration dialogs.

11. Restart IIS

We can now navigate to our website with pretty URLs in the form http://localhost:84/product/show/4. For more detailed configuration options, consult the readme included with the download from CodePlex. The download includes configuration examples, as well as instructions for enabling logging and other advanced features. Although we had to add an HTTP Module, the routes remained the same, without any extensions. In addition, all URL-generating action helpers still generate pretty URLs, ensuring that no end user ever sees a URL with the .mvc extension. With the URL rewriting extension in place, we can now employ its features to address canonical URLs, forwarding, and other rewriting concerns.

With our application deployed and configured, we’ll take a look at automating deployments.

6.5 Automating deployments

On launch night, tensions are high as the smallest mistake could bring your website down. To eliminate the human mistakes that inevitably occur, we would like to automate as much as possible. Ideally, we could simply push a button, and our website would be updated in moments. How this happens depends largely on the deployment environment. Regardless of the deployment environment, any good deployment strategy requires the use of continuous integration.

6.5.1 Employing continuous integration

Working in an environment without an automated integration process can be hectic, and nerve-racking. Because “it works on my machine” does not suffice in a deployment scenario, we need a set of practices to ensure our code always works, and is always ready to deploy. To achieve continuous integration, Martin Fowler laid out a set of practices to adhere to (from <http://www.martinfowler.com/articles/continuousIntegration.html>):

* Maintain a single source repository (use source control)
* Automate the build
* Make your build self-testing
* Make sure everyone commits every day
* Every commit should build the main line of your code on an integration machine
* Keep the build fast
* Test in a clone of a production environment
* Make it easy for anyone to get the latest executable
* Ensure everyone can see what’s happening
* Automate deployment

We won’t cover all of the continuous integration practices in this book, as entire books have been written on this topic. In addition to adhering to these practices, the “check-in dance” ensures that no one inadvertently breaks the build. The check-in dance steps are:

1. Run the local build

2. Announce to the team you are integrating (for large changes)

3. Pull down the latest version of the mainline. Merge any conflicts

4. Run the local build

5. If successful, commit the changes, providing a descriptive comment

6. Wait for the server build to be successful

7. If the build fails, drop everything and fix it

Depending on the development environment, there are several continuous integration server tools and technologies to employ. One popular continuous integration stack includes:

* Subversion (SVN) for source control
* NAnt for build automation
* NUnit for testing
* CruiseControl.NET for the continuous integration server

Which tool we use does not matter as much as the practices the tools enforce, although we would like our tools to introduce as little friction as possible into the development environment. If we have to wait for a slow or unreliable source control server, our practices are less likely to be followed. Whichever build technology we decide to use, the result of each build should be a single deployment file, checked in to source control at the end of a successful server build. To enable push-button XCOPY deployments, we’ll next look at some key NAnt features.

6.5.2 Enabling push-button XCOPY deployments

In an intranet environment, XCOPY deployments can be as simple as setting up a network share on the deployed machine. In other situations, the deployment file, whether it is an installer or self-contained .zip file, must be copied over manually or pulled down from source control. Regardless, if the files can be pushed from a network share, or pulled manually on the server, our deployment package will include:

* The complete application
* The build tool, if used (NAnt)
* A deployment script
* A batch file to kick the process off

Our automated continuous integration build creates and checks in this deployment package. When we have a deployment package in source control, it enables us to deploy any version of our application as needed. With a tool like CruiseControl.net, it is possible to automate the deployment of the latest version of the application as needed.

NAnt, along with the sister project NAntContrib, provides dozens and dozens of tasks out of the box, which can be compiled together to create a single deployment script. These tasks are:

* Source control tasks
* IIS tasks
* File and directory tasks such as creation, deletion, and copying
* Zip tasks
* XML manipulation tasks

With a manual process in place, we can start automating one step at a time with NAnt tasks, until the entire deployment process is automated. Many teams already employ a build process in the form of a Word document or wiki entry, detailing the manual steps. It is only a matter of finding the corresponding NAnt task for each manual task, and the deployment is automated. If no NAnt task exists for a particular operation, NAnt provides the Exec task, which can execute anything that can execute in the command-line. The key NAnt tasks for deployments include:

* unzip
* copy
* exec
* xmlpoke

We’ll need the unzip task to unzip the deployment package originally checked in to source control. If this is a manual pull of the deployment package, we can unzip the package manually. The copy task is used to copy the complete application to the correct deployed directory, performing an XCOPY deployment in one automated task. The exec task is used for a variety of scenarios, such as restarting IIS, stopping and starting services, registering assemblies, and so on. The xmlpoke task is used to manage deployment configurations by manipulating key configuration files, such as the Web.config file. In the next section, we’ll examine how to manage multiple deployment configurations with NAnt and xmlpoke.

6.5.3 Managing environment configurations

Development teams often deploy their applications in multiple environments. For any given project, there are at least two environments: production and development. Many teams integrate to one or more test environments before releasing to production. Among these different environments, the deployment must change. Some environments require merely a connection string change, and others require debug flags, configuration values, email addresses, and more. In an automated deployment, the deployment script must take into account the different environment settings. Notably, it must know what environment it is deploying to, and what changes to make to the application to match that environment.

With NAnt, managing all of these environment configurations is straightforward. Deployments are kicked off with a batch file, which merely starts NAnt. The deployment package zip file contains:

* Dev.bat
* CommonDeploy.bat
* deployment.build
* NAnt\
* website\
* database\

The NAnt folder contains the entire runtime distribution of NAnt. We include the distribution to avoid an environmental setup step on every server to which we deploy. The website folder contains the complete application that we XCOPY deploy to the correct folder on the server. The deployment.build is the NAnt build script that contains the complete deployment script. The Dev.bat file is a bootstrapper file that calls CommonDeploy.bat. In listing 6.6, the bootstrapper file Dev.bat call overrides the deploy directory and connection string properties by setting environment variables, and then calls the CommonDeploy.bat script.

Listing 6.6 Setting the environment configuration in Dev.bat

SET driverClass=NHibernate.Driver.SqlClientDriver

SET connectionString=Data Source=.\sqlexpress;Initial Catalog=TODO;uid=sa;pwd=TODO

SET localConnectionString=Data Source=.\sqlexpress;Initial Catalog=TODO;uid=sa;pwd=TODO

SET dialect=NHibernate.Dialect.MsSql2005Dialect

SET websiteTargetDir=\\TODO

SET databaseServer=TODO\sqlexpress

SET databaseName=TODO

SET databaseIntegrated=false

SET databaseUsername=sa

SET databasePassword=TODO

SET shouldReloadDatabase=true #A

CommonDeploy.bat

#A SET command declares variables.

In the Dev.bat file, we set up the environment variables for the environment configuration values (some of which still need to be filled in). With one CommonDeploy.bat batch file that runs off environment variables, we can create additional bootstrapper batch files for each target environment. The end of the Dev.bat batch script calls into the CommonDeploy.bat script, which provides a common bootstrapper file on top of NAnt, shown in Listing 6.7 below.

Listing 6.7 Bootstrapper CommonDeploy.bat file overriding NAnt properties

nant\nant.exe

-buildfile:deployment.build

-D:should.reload.database="%shouldReloadDatabase%" #A

-D:driver.class="%driverClass%"

-D:connection.string="%connectionString%"

-D:local.connection.string="%localConnectionString%"

-D:dialect="%dialect%"

-D:website.target.dir="%websiteTargetDir%"

-D:database.server="%databaseServer%"

-D:database.name="%databaseName%"

-D:database.integrated="%databaseIntegrated%"

-D:database.username="%databaseUsername%"

-D:database.password="%databasePassword%"

-D:test.database.name="%testDatabaseName%"

-D:excel.server.path="%excelServerPath%"

#A Use previously set environment variables

This entire command is in a single CommonDeploy.bat file, calling NAnt, using environment variables set up by a previous environment-specific batch file (Dev.bat in our case). The “-D” command-line switches for NAnt allow us to override properties with the correct deployed values. Because our deployment database will most likely require a different connection string than our local configuration, we need to use NAnt to override this value during deployment. A portion of the deploy.build file is in Listing 6.8 below.

Listing 6.8 Deployment.build NAnt script with the deploy target

<target name="deploy">

<call target="rebuildDatabase" if="${should.reload.database}" /> #A

<xmlpoke #B

file="website/bin/hibernate.cfg.xml"

xpath="${connection.string.path}"

value="${local.connection.string}">

<namespaces>

<namespace prefix="hbm"

uri="urn:nhibernate-configuration-2.2"></namespace>

</namespaces>

</xmlpoke>

<copy todir="${website.target.dir}" overwrite="true"

includeemptydirs="true" > #C

<fileset basedir="website">

<include name="\*\*" />

</fileset>

</copy>

</target>

#A Call another target

#B Change the connection string

#C Copy all website files

The first items to notice in this NAnt script are the XML attribute values in the format ${some.value.here}. These are NAnt properties, whose values were defined earlier through our bootstrapper file. When the CommonDeploy.bat file executes, the command-line switches set these property values with the appropriate environmental settings. Finally, the “deploy” target performs the actual deployment. An NAnt target is a named group of tasks, similar to a method in C#.

The actual CodeCampServer NAnt deployment script is considerably larger, but performs these common deployment steps:

1. Applies environmental configuration to various configuration files

2. Rebuilds the local database

3. Populates the local database with test data

4. Removes the existing application

5. Copies files to target location

Each of step in the deployment.build script first echoes a message to the console, for informational and debugging purposes. Although CodeCampServer’s current build script is large, it was built up over time to support the various configuration and deployment needs. Deployment scripts can be as simple as copy and delete tasks; it depends on each deployment scenario.

Summary

With the new routing abilities of ASP.NET MVC came new deployment challenges. Although IIS 7 supports extension-less, pretty URLs out of the box, earlier versions of IIS do not. However, we have a variety of deployment options with earlier versions of IIS, some of which enable pretty URLs. URL rewriting is the most powerful of these deployment options, as it opens up new scenarios in URL canonicalization and seamless resource management.

When we configure our environment, we must devise a reliable deployment strategy to ensure the right application is deployed with the correct configuration. At the heart of a solid deployment strategy is continuous integration, which includes practices such as automated deployments and self-testing builds. With free, widely used open source tools such as CruiseControl.NET, NAnt, NUnit, and others, we can build an automated build and deployment server. By packaging NAnt, a build script and a bootstrap batch file, we can harness the flexibility and power of NAnt to deploy and configure our application to multiple environments, up to and including production.

Next up in Chapter 7, you will learn how to leverage the many existing ASP.NET runtime features in you applications so that you can get up to speed quickly.