Microsoft

# Deploying .NET Applications Lifecycle Guide

patterns & practices

# Contents

# 1

# Introduction

Welcome to Deploying .NET Framework-based Applications. With the introduction of the .NET Framework, Microsoft created a new environment for developing applications. If your organization is developing .NET-Framework based applications, you face the challenge of deploying those applications efficiently and reliably throughout your environment. If you are experienced in the area of deployment, some of the challenges will be familiar. However, there are a number of new technologies in .NET Framework–based applications, and therefore several considerations that are unique to deploying them. This guide will give you the information necessary to plan and implement the effective deployment of your Framework-based applications. Should you have any questions, comments, or suggestions on this material you can send feedback to: devfdbck@microsoft.com.

**Note:** In this guide we will generally refer to .NET Framework-based applications as Framework Applications.

## Who Should Read This Guide

This guide is targeted at people within an organization responsible for deploying Framework applications. In some organizations those individuals will be specialists, but in many others, application deployment will form part of a larger role in either infrastructure or application development. As application deployment spans these two areas, we have made no significant assumptions for the prerequisite knowledge required by the reader. A developer with some infrastructure knowledge should gain benefit from this guide, as should an infrastructure professional with some knowledge of developer issues. The main focus of this guide is, of course, the deployment of applications, but where other issues (such as application design) have an impact on deployment, we discuss those issues. For example, although this guide does not discuss testing methodology, it does include recommendations for

deploying applications from development computers to test environments. Similarly, this guide also provides recommendations for deploying to staging servers as part of the release process.

This guide assumes that you are using a Windows Server 2003 server environment, and that you are deploying applications for the Framework v1.1. However, much of the information contained in this guide is also relevant to applications built on version 1.0 of the Framework.

Application deployment can occur on any scale, from single machine deployment to hundreds of thousands of computers. For this reason we have made no assumptions as to the size of organization involved, or the nature of the specific applications deployed. You should therefore read this guide if you are responsible for deploying .NET Framework-based applications of any size, with any performance or scalability requirements.

# Chapter Outlines

This guide consists of the following chapters, each of which takes you through a part of the application deployment process. Each chapter is designed to be read in whole or in part, according to your needs.

## Chapter 1: Introduction

This remainder of this chapter introduces important concepts for Framework applications, defining key terms and discusses deployment issues for the .NET Framework itself.

## Chapter 2: Planning the Deployment of .NET Framework – based Applications

Before you can effectively deploy your Framework applications, you need to understand what is involved in the deployment and how to plan effectively for that deployment. This chapter introduces the elements you are likely to deploy with a Framework application and discusses the different deployment strategies you can use, along with information on when each methodology is appropriate.

## Chapter 3: Implementing the Deployment of .NET Framework – based Applications

Now that you understand the issues surrounding application deployment, you should be able to carry out the deployment itself. This chapter focuses on demonstrating the methods used to deploy applications to target computers, for each of the deployment strategies discussed in the previous chapter.

### Chapter 4: Maintaining .NET Framework – based Applications

Deployment does not stop once an application has been installed. At some point after the application is successfully deployed, you may need to deploy updates. Also applications do not necessarily exist in isolation. In many cases, your applications will need to exist alongside other applications and may share resources with those applications. They may even need to exist alongside other applications build on a different version of the Framework. This chapter discusses each of these issues, showing you how you can continue to run and update your applications in more complex, real-world environments.

## What Is Application Deployment?

Application deployment is the process of taking an application that has been developed and tested, and installing that application on target computers that require it. If you are going to be consistently successful in your application deployment, you need to ensure that you follow established best practices throughout the lifecycle of your applications.

### Microsoft Solutions Framework (MSF)

MSF provides a series of best practices, principles and models that can be used to plan, build, and deploy solutions in an efficient manner.

The MSF process model is split into five integrated phases:

- Envisioning
- Planning
- Developing
- Stabilizing
- Deploying

Together, the phases form a spiral life cycle (see Figure 1.1 on the next page) that can apply to anything from a specific application to a complete environment.

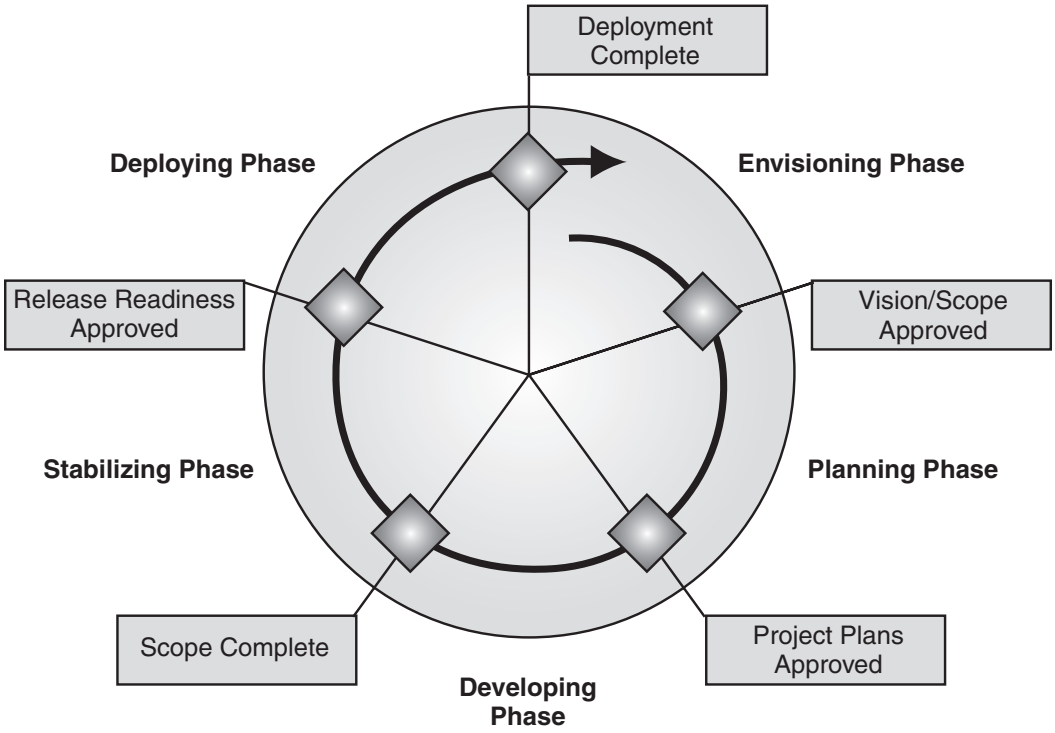


**Figure 1.1**
*The MSF process model v3.1*

The process defined here shows the development of solutions from inception to complete deployment. The application development lifecycle includes many different processes, such as the actual coding of the application, the management of source control in a team development environment, the build process, the test process, the final packaging and staging, the deployment process, and often the upgrade of applications after they are installed in the production environment. All of these processes are to some extent interrelated, but in this guide we are concerned with the final part of that process, the deploying phase. As the diagram shows, this covers each of the processes from when the application is judged ready for release up until the deployment is judged complete.

**Note:** For more information on MSF, see *http://www.microsoft.com/msf*

## The Deployment Process

Successful deployment relies on more than development of the application and the final packaging process. For example, applications need to be tested in a controlled way, and you should generally use staging servers between successful testing and final deployment.

The deployment process generally consists of managing your applications as they move through a series of different environments. These are:

- Development Environment
- Test Environment
- Staging Environment
- Production Environment

We will look at each of the environments in turn:

### Development Environment

This environment is where most of the coding and developing takes place. Because your developers require sophisticated development tools, such as Visual Studio® .NET 2003 development system and various software development kits (SDKs), this environment does not usually resemble the one where your applications will eventually be installed. However, the nature of your development environment does affect the deployment of your applications. Your applications are typically compiled and built in this environment, as well as packaged for distribution to the other environments. As such, you should be aware that applications which run without problems on development computers do not necessarily run as smoothly in other environments. The most common causes of this problem include:

- Dependencies required by your application (such as assemblies, Component Object Model (COM) objects, the .NET Framework classes, and the common language runtime) are present on the developer computers, but they may not be included or installed successfully in other environments. Some of these dependencies are present on your development computers simply because they are provided by the development tools and SDKs, so you need to be aware of exactly what needs to be deployed to ensure successful installation into other environments.
- Application resources (such as message queues, event logs, and performance counters) are created on developer workstations and servers, but they might not be successfully recreated or configured as part of the deployment to other environments. You need to thoroughly understand your application's architecture and any external resources that it uses if you are to deploy your application successfully.
- Applications are configured to use resources such as development database servers and Web services while they are being developed. When they are deployed to another environment, these configuration settings need to be updated so that they use the resources appropriate for that particular environment. Because many of these settings are stored in configuration files within the new .NET Framework, you need to manage the process of updating or replacing these configuration files as you deploy your application from one environment to another.

### Test Environment

The test environment should be used to test the actual deployment process, such as running Windows Installer files, copying and configuring application files and resources, or other setup routines. You (or your test team) need to verify that application files and resources are installed to the correct locations, and that all configuration requirements are met, before testing the application's functionality.

After you are satisfied that the application installs correctly, you should then perform further tests that verify the application functions as expected. As well as simple functionality, the tests should include performance, stress and load testing. This will help to ensure that your application is deployed on the correct hardware, or over the appropriate number of computers.

To be certain that your tests have meaningful implications for how your application will install and function in the live production environment, you should ensure that test computers resemble your production computers as closely as possible. They should not have the development tools installed that you used to produce the application, because that could mask potential problems that will then only become apparent when you roll out your application to the end users.

### Staging Environment

The staging environment is used to house your application after it has been fully tested in the test environment. It provides a convenient location from which to deploy to the final production environment. Because the staging environment is often used to perform final tests and checks on application functionality, it should resemble the production environment as closely as possible. For example, the staging environment should not only have the same operating systems and applications installed as the production computers, it should also have a similar network topology (which your testing environment might not have). Usually, the staging network mimics the production environment in all respects, except that it is a scaled-down version (for example, it may have fewer cluster members or fewer processors than your server computers).

### Production Environment

The production environment is the "live" environment where your applications are put to work. This environment is undoubtedly the most complex, usually with many installed productivity, line-of-business and decision-support systems used by different users in the organization. Consequently, it is the most difficult to manage and control; therefore, administrators and information technology professionals are employed to ensure that users can take advantage of your applications along with applications from other vendors. The main goal of this guide is to provide guidance for allowing you to ensure that your solutions are installed successfully in this environment. It also describes various approaches for upgrading applications after they are installed in this live environment.

# What are .NET Framework – based Applications?

.NET Framework–based applications (Framework applications) are programs that require the.NET Framework to run. For the purposes of this book we will refer to these applications as Framework applications.

The elements of a Framework application are generally split into multiple logical layers which handle the different pieces of functionality. Those logical layers can generally communicate with each other on the same computer, or over the network. Exactly how you map logical layers to the physical tiers of your solution will depend on a combination of security, scalability, and performance issues.

---

**Note:** Some Framework applications make use of Web services to connect to other applications or services. These are often given the name .NET-connected applications, but as they are a subset of Framework applications, you will not see that term used in this guide.

---

You can create Framework applications using a variety of development environments, but for the purposes of this book, we assume that you are using Visual Studio .NET 2003.

# What is the .NET Framework?

The .NET Framework is an object-oriented programming platform that simplifies application development in highly distributed environments, including the Internet. The .NET Framework fulfills the following objectives:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts, guarantees the safe execution of code, and eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with other code.

## Common Language Runtime

The Common Language Runtime (CLR) is the foundation of the .NET Framework. Think of the runtime as an agent that manages code at execution time, providing a set of core services such as:

- Memory management
- Thread management
- Remoting

The CLR also enforces strict type safety and other forms of code accuracy that ensure security. Thus, it is important to know how well the CLR is operating as this has a direct effect on your applications.

The following diagram shows how the Common Language Runtime underpins the architecture of .NET Framework-based applications.

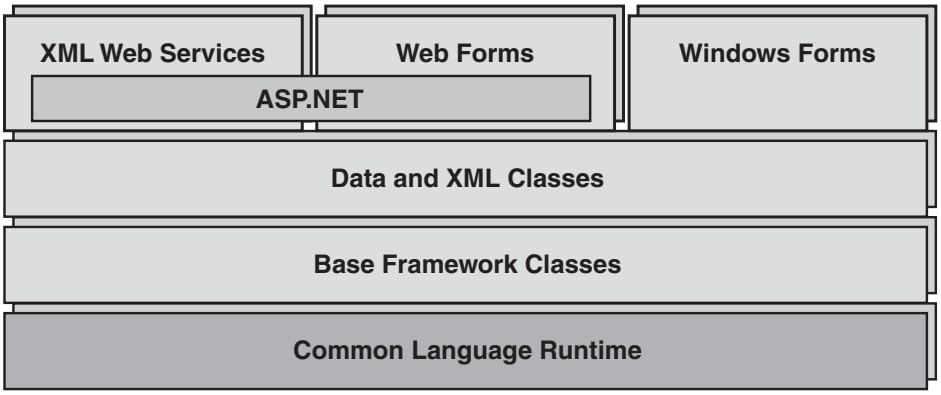


**Figure 1.2**
*Architecture of .NET Framework-based applications*

Any code which uses the Framework is known as *managed* code and can use the core services by executing under the runtime. Code that does not use the Framework is known as *unmanaged* code.

---

**Note:** To run managed code on a computer, the .NET Framework must be installed on that computer.

---

Using the managed environment of the runtime eliminates many common software issues. For example, the runtime automatically handles object layout and manages references to objects, releasing them when no longer in use. This automatic memory management helps to resolve the two most common application errors of memory leaks and invalid memory references.

The runtime also enforces code robustness by implementing a strict type- and code-verification infrastructure called the common type system (CTS). CTS ensures that all managed code is self-describing. A number of Microsoft and third party language compilers generate managed code that conforms to the CTS. Hence, managed code can consume other managed types and instances, while strictly enforcing type fidelity and type safety.

The CLR design supports a variety of different types of applications, from Web server applications to programs with a traditional rich Microsoft Windows® operating system user interface. Each type requires a runtime host to start the application.

The runtime host loads the runtime into a process, creates the application domains within the process, and loads user code into the application domains.

### .NET Framework Class Library

The .NET Framework class library is a comprehensive, object-oriented collection of reusable classes, interfaces, and types that you can use to develop applications. The classes provide access to system functionality such as:

- Security features
- Graphical user interface (GUI) elements
- Enterprise services (such as message queuing, load balancing, and transacted components
- File input and output
- Data access

The .NET Framework types are the foundation on which Framework applications, components, and controls are built. The .NET Framework includes types that perform the following functions:

- Represent base data types and exceptions.
- Encapsulate data structures.
- Perform I/O.
- Access information about loaded types.
- Invoke .NET Framework security checks.
- Provide data access, rich client-side GUI, and server-controlled, client-side GUI.

The .NET Framework provides a rich set of interfaces, as well as abstract and concrete (non-abstract) classes. You can use the concrete classes as is or, in many cases, derive your own classes from them. To use the functionality of an interface, you can either create a class that implements the interface or derive a class from one of the .NET Framework classes that implements the interface.

---

**Note:** Code that uses any of the classes from the .NET Framework class library requires the .NET Framework be installed on the computer on which it runs.

---

## Types of Framework Applications

You can use Visual Studio .NET 2003 and the .NET Framework to develop the following types of applications and services:

- Windows Forms Smart Client Applications
- ASP.NET-based applications (Web-based applications)

- Extensible Markup Language (XML) Web services
- Windows Services
- Console applications
- Scripted or hosted applications

We will discuss each of these in turn:

## Windows Forms Smart Client Applications

In Windows Forms Smart Client applications, the logic and code used to build the GUI runs on the client computer. Windows Forms controls and menus are actually .NET Framework classes, as are the Windows Forms themselves. In addition, display effects, such as colors, borders, sizes, and positions are all controlled by accessing classes and enumerations from the .NET Framework class library.

Windows Forms Smart Client applications provide the following benefits:

- A rich user interface
- Full access to the local disk and local application programming interfaces (APIs)
- Elimination of network latency (since the application runs on the client machine)
- Multiple deployment options
- Off-line application support
- Offloading of processing to the client machine
- Ability to interact with other locally installed applications

## ASP.NET Applications (Web-based Applications)

Much of the logic and code used to build the user interface for a Microsoft ASP.NET application resides (and runs) on a Web server, rather than on the computer at which the user is sitting. The browser-based GUI is dynamically built by your ASP.NET application, which transmits HTML to the browser. The GUI may be supplemented with some client-side script to provide ease of usability, increased responsiveness, data-entry validation, and dynamic display effects, but the bulk of the interface is controlled by server-based code to which the client Web browser does not have direct access. In effect, the client uses HTML, supplemented by dynamic HTML (DHTML), to present the user interface. Although the server-side code depends on the .NET Framework to build the GUI, it produces browser-agnostic HTML.

ASP.NET applications may also include Windows Forms user controls. These are assemblies referenced from Web pages that are downloaded to the user's computer and executed upon demand. These controls can provide extra functionality to your application, much like Microsoft ActiveX® controls hosted in a Web browser did in the past.

ASP.NET applications provide the following benefits:

- Processing handled by the server machine.
- Some local processing through Windows Forms user controls.
- Use of Dynamic HTML to create user interface features.

## XML Web Services

XML Web services provide a standards-based platform for application integration. .NET-based applications may be constructed to use multiple XML Web services from various sources that work together regardless of where they reside or how they were implemented. As well as working with each other, XML Web services can also be consumed by both Windows Forms and ASP.NET applications. There are three major differences between Web services and traditional Web applications:

- Web services use SOAP messages instead of MIME messages.
- Web services are not HTTP specific.
- Web services provide metadata describing the messages they produce and consume.

## Windows Services

Windows Services are applications which generally run in the background and are managed by the Service Control Manager. They can be started automatically at system boot, by a user through the Services MMC snap-in, or by another application.

Windows Services have their own processes, and start in a predefined security context that is independent of the logged on user. Windows Services do not usually have a user interface.

---

**Note:** Windows Services will not run on Windows 9x or Windows ME.

---

## Console Applications

A console application provides character-mode support in a text-only window. Unlike a Windows Forms application, it is not forms driven, although Windows Forms can be called from a console application.

The Framework provides classes that support output and input from console windows with standard I/O behaviors.

### Scripted or Hosted Applications

Scripted or hosted applications use existing hosting features of an operating system or application to run, meaning that the code can be interpreted at run time and does not have to be precompiled. The most common examples are Visual Basic Scripting Host (*.vbs) and Windows Scripting Host (*.wsf) applications. While these types of applications performs specific functions, the commonly do not have much if any user interface. These types of applications can often be incorporated into other applications to perform specific functions.

## Where to Deploy the .NET Framework

The .NET Framework needs to be deployed on all computers that will run managed code. If you have a distributed application, however, you will not necessarily need to deploy the Framework on all participating computers. Which computers require the Framework will typically depend on the role which they fill. This section will help you determine where it is appropriate to deploy the Framework.

### Clients

Whether you need to deploy the Framework on your client base or not will depend in part on the type of .NET-based applications you are deploying. If you are deploying (or plan to deploy) Windows Forms applications, you need to ensure that the Framework is installed, or they will not operate properly. This is because .NET Framework classes are responsible for the GUI of the application.

For ASP.NET applications, in many cases the client will not need the Framework installed. This is because the .NET Framework is being used at the server to generate the user interface and the server supplies the client with standard HTML. However, if your ASP.NET application uses Windows Forms user controls, your clients will need the Framework installed. This is because Windows Forms user controls rely on the .NET Framework to manage their execution.

---

**Note:** Using Windows Forms user controls introduces a number of other issues, such as browser and code security, with which you need to be familiar. A full discussion is outside the scope of this guide. For more information, see “Writing Secure Managed Controls” on MSDN.

---

### Web Servers

If you use server-side code in your .NET-based applications, then you must ensure that the target Web servers for your solution have the .NET Framework installed, or your application will not function properly. The distributed nature of modern

Internet or intranet-based applications means that Web servers often perform a number of different roles in your .NET applications. You might use Web servers to provide any (or all) of the following features for your distributed Framework application:

- ASP.NET Web Forms for the GUI
- Web Services for reusable, Web-based functionality
- Transfer of XML or text-based data
- Intermediate XML or text-based processing
- Connectivity and communication across computer and network boundaries

### Business Logic Servers

In some distributed Framework applications, there will be separate computers used to encapsulate business logic. The advantages of doing so include ease of manageability and scalability, among others. In a pure .NET environment, this business logic will be provided by .NET components and so the Framework will be required. However, in some cases your .NET-based application may be replacing a legacy application where all the business logic was encapsulated in COM objects. .NET-based applications can interoperate with COM objects, so in this circumstance you could leave any servers that exclusively run the Business Logic tier unchanged and not install the .NET Framework on those computers. However, in the medium term you should consider updating the components to take advantage of the .NET Framework.

### .NET Framework to Database Servers

For scalability, manageability, stability, and fault-tolerance, data tiers, in distributed applications, are often physically separated from the other tiers in a typical solution.

Microsoft SQL Server™ 2000 does not require the .NET Framework to be installed. Therefore, if your application architecture does include a physically separate Data tier, .NET Framework does not need to be installed on your database servers and analysis servers. However, if you need .NET assemblies to execute on your database servers, the .NET Framework does need to be installed on those computers.

## Deploying the .NET Framework

After determining which computers participate in your distributed .NET-based application require the .NET Framework, you need to ensure that the Framework is installed correctly on the appropriate computers in your production environment.

## Requirements for Deploying the .NET Framework

There are certain minimum requirements that need to be met so that you can deploy the .NET Framework within your organization. The following table lists the minimum requirements

**Table 1.1: Minimum Requirements for deploying the .NET Framework**

| Client | Server |
|---|---|
| Pentium 90 MHz | Pentium 133MHz |
| 32 MB RAM (96 MB recommended) | 128 MB RAM (256 MB recommended) |
| Windows 98 or above | NT 4.0 or above (Windows 2000 or above for ASP.NET applications [see Note]) |
| Windows Installer 2.0 | Windows Installer 2.0 |

**Note:** ASP.NET applications require Windows 2000 as a minimum because they require IIS v5.0, which cannot run on the Microsoft Windows NT® 4.0 operating system.

## Further Requirements for .NET-based Applications

In addition to the Framework itself, some applications will have other requirements as shown in the following table.

**Table 1.2 Requirements for some .NET-based Applications**

| Client | Server |
|---|---|
| Internet Explorer 5.01 | IIS 5.0 (for ASP.NET applications) |
| MDAC 2.6 | ASP.NET |
| WMI | MDAC 2.6 |
| | WMI |

**Note:** The table lists minimum requirements. Wherever possible you should use the latest version of each to minimize security risks to your environment.

Rather than manually ensuring dependencies such as MDAC and WMI are correctly installed before you roll out your solution to the production environment, you might want to build checks into your setup routines to detect whether these dependencies are present. If they are not installed, you can then automatically install them along with your application.

We will discuss each dependency in more detail:

### Internet Explorer 5.01

It is possible to run some .NET-based applications using versions of Internet Explorer 5.01. However this version is quoted as a minimum requirement as it is the first to support all of the added functionality required by .NET-based applications, including code download, cryptography APIs, intranet / Internet zone detection within the CLR and multilanguage support for international encodings.

### Microsoft Data Access Components (MDAC)

Microsoft Data Access Components (MDAC)are a collection of core files provided to help applications by providing a means of accessing data. MDAC includes core files for Open Database Connectivity (ODBC), ActiveX Data Objects (ADO), OLEDB, Network libraries, and client configuration tool for SQL Server.

Depending on your data access strategy, you may need to ensure that MDAC is installed on the client computers, the business servers, the Web servers, or the database servers. MDAC 2.6 or later is required by the .NET Framework, and at least MDAC 2.7 SP1 is recommended.

You should ensure that you use the same version of MDAC in the development environment that exists in your production environment. This will help to minimize any problems with transferring the application from the development environment to the production environment.

The latest version of MDAC is available for download at Microsoft's Universal Data Access Web Site.

### Microsoft Internet Information Services (IIS)

If your solution includes ASP.NET applications or Web services, you need to ensure that IIS and the latest security patches are installed on your Web server(s). IIS is not installed with the Microsoft Windows Server™ 2003 operating system by default as it is not always required. It can be installed from the operating system installation CD or DVD. The latest security patches can be downloaded and installed from the Windows Update site.

### Microsoft Windows Management Instrumentation (WMI)

WMI defines a nonproprietary set of environment-independent specifications that allow management information to be shared between management applications running in both similar and dissimilar operating system environments. WMI provides enterprise management standards and related technologies that work with existing management standards, such as Desktop Management Interface (DMI) and Simple Network Management Protocol (SNMP). WMI complements these other standards by providing a uniform model. This model represents the managed

environment through which management data from any source can be accessed in a common way.

If your solution uses WMI features, you need to ensure that the core WMI is installed on your target computers. WMI is included as a standard feature in Windows Server 2003, Windows XP, Windows 2000, and Windows Me. Additionally, it is automatically installed by Visual Studio .NET on Windows 95, Windows 98, and Windows NT 4.0. However, if your target computers do not have the core WMI installed, a setup package can be downloaded and installed from MSDN Code Center .

For more information on WMI see "Microsoft Windows Management Instrumentation: Background and Overview" on MSDN.

## Deploying the .NET Framework

The Framework is deployed using the.NET Framework redistributable package, which can be obtained from MSDN or the Windows Update Web site . Alternately, you can also obtain the redistributable package on a product CD or DVD. Dotnetfx.exe is available on the .NET Framework SDK, and on the Microsoft Visual Studio .NET 2003 DVD.

The .NET Framework redistributable package is actually a Windows Installer package that is wrapped into a single, self-extracting executable file called Dotnetfx.exe. The Dotnetfx.exe executable file launches Install.exe, which performs platform checks, installs Windows Installer 2.0 if necessary, and then launches the Windows Installer package (.msi file).

The following table describes the command line options you can specify when installing Dotnetfx.exe. To specify options when installing Dotnetfx.exe, you must pass the options to the Install.exe wrapper using the **/c:** option.

### Syntax

dotnetfx [/q:a] [/c:"Install [/l][/q]"]

**Table 1.3: Command line options for installing Dotnetfx.exe**

| Option | Description |
|---|---|
| **/l** | Creates the setup log, *netfx.log*, in the %temp% directory. Error codes returned from Dotnetfx.exe are written to this log. |
| **/q** | Specifies quiet install mode. Suppresses the display of the setup user interface. For a quiet install, you must also specify the Dotnetfx.exe **/q:a** option to suppress the extraction user interface. |

You should use the **Windows Add/Remove Programs** option to uninstall a .NET Framework-based application that hosts Dotnetfx.exe. This ensures that

Dotnetfx.exe will not be uninstalled independently of an application that is dependent on it in order to run.

**Note:** For more information on using Dotnetfx.exe see the document the .NET Framework Redistributable Package 1.1 Technical Reference. For further details see "More Information" at the end of this chapter.

**Note:** If you choose to use **Dotnetfx.exe** for distributing your application, you must have a valid, licensed copy of the Microsoft .NET Framework SDK and agree to abide by the terms of the Microsoft .NET Framework SDK end user license agreement (EULA).

The Framework will generally be deployed in one of two ways:

- Preinstalling to any computers that will require it.
- Installing alongside .NET-based Applications.

We will look at each of these in more detail:

### Preinstalling the .NET Framework

If you plan on using the Framework widely throughout your organization it often makes sense to deploy it before you deploy your .NET-based applications. There are a number of methods you can use to ensure that the Framework is distributed to those computers that require it. These include:

- Microsoft Systems Management Server (SMS)
- Group Policy
- Network Share
- E-mail
- Internal Web site
- Microsoft Web site

The .NET Framework Redistributable Package installs significant run-time components onto a target computer. This means that the user initializing the .NET Framework setup must have administrator privileges for that computer (or be able to perform the installation with elevated privileges). This requirement has implications for how you choose to deploy the .NET Framework. For example, using Group Policy software deployment to install the package over the network allows you to ensure that the package installs with elevated privileges. Similarly, using SMS allows you to install the Framework with the required permissions. However, if you expect your users to initialize the setup of the .NET Framework, either locally or over the network, then you (or your systems administrator) must ensure that the user has administrator privileges on his or her local computer or can run the installation with elevated privileges.

**Note:** To install the Framework using Group Policy or SMS, you first need to extract the Windows Installer file from dotnetfx.exe. For more details on how to do this, see: "Redistributing the .NET Framework" on MSDN

### Installing Alongside Framework Applications

In some cases, where you cannot pre-determine which computers will require the .NET Framework in advance, you may choose to install the Framework only when it is required, in other words when a .NET-based application is installed. This approach is particularly useful when you are developing and packaging your Framework application for general sale, or perhaps for installation onto a number of different customer networks, and you do not have any knowledge of the infrastructure where your application is to be deployed. You can do this by using the setup.exe Bootstrapper sample. This sample checks to see if the Framework has already been installed, and if it hasn't, it will then install the Framework prior to installing the application.

**Note:** For more information on using the setup.exe Bootstrapper sample, see Chapter 3, "Implementing the Deployment of .NET-Framework based Applications."

## Localized Versions of the .NET Framework

There are 22 different language versions of the .NET Framework redistributable. Each version is the same programmatically, but the user interface differs during installation is localized for each language.

If you want localized dialogs to be displayed by the Framework post-installation, you will also need to install a corresponding language pack. There are 21 language packs for the .NET Framework (An English language pack is not provided as all the error codes and messages are in English by default).

**Table 1.4: Language Packs Supported by .NET Framework v1.1 (v1.1.4322)**

| Language | Locale Identifier (LCID) |
|---|---|
| Chinese (Simplified) | 2052 |
| Chinese (Traditional) | 3076 |
| Czech | 1029 |
| Danish | 1030 |
| German | 1031 |
| Greek | 1032 |
| Spanish | 3082 |
| Finnish | 1035 |

| Language | Locale Identifier (LCID) |
|---|---|
| French | 1036 |
| Hungarian | 1038 |
| Italian | 1040 |
| Japanese | 1041 |
| Korean | 1042 |
| Dutch | 1043 |
| Norwegian | 1044 |
| Polish | 1045 |
| Portuguese (Brazilian) | 1046 |
| Portuguese (Portugal) | 2070 |
| Russian | 1049 |
| Swedish | 1053 |
| Turkish | 1055 |

Localized versions of the redistributable Dotnetfx.exe file are available from the Download Center on MSDN.

**Note:** When installing Dotnetfx.exe on a computer running Windows 98, you must install the localized version of Dotnetfx.exe that corresponds to the localized version of Windows 98 running on the computer. For example, you must install the German version of Dotnetfx.exe on a computer running the German version of Windows 98. This limitation applies only to Windows 98.

**Note:** You cannot install two different language versions of the .NET Framework on the same machine. Attempting to install a second language version of the .NET Framework will cause the following error to appear: "Setup cannot install Microsoft .NET Framework because another version of the product is already installed." If you are targeting a non-English platform or if you wish to view .NET Framework resources in a different language, you must download the appropriate language version of the .NET Framework language pack.

# Summary

This chapter has introduced you to this guide and summarized the chapters in it. It has also examined the nature of Framework applications and the .NET Framework and looked at deploying the .NET Framework. With the information provided in this chapter, you can go on to plan the deployment of your .NET-based applications, which is the subject of the next chapter.

# More Information

For more information on Windows Forms user controls:
Writing Secure Managed Controls on MSDN
*http://msdn.microsoft.com/library/en-us/cpguide/html/cpconwritingsecuremanagedcontrols.asp*

To obtain the latest version of MDAC:
Microsoft's Universal Data Access Web Site
*http://www.microsoft.com/data*

To get the latest Windows security patches:
Windows Update
*http://www.windowsupdate.com*

A WMI setup package can be downloaded and installed from MSDN Code Center
*http://msdn.microsoft.com/downloads/?url=/downloads/sample.asp?url=/msdn-files/027/001/576/msdncompositedoc.xml*

For more information on WMI:
Microsoft Windows Management Instrumentation: Background and Overview
*http://msdn.microsoft.com/library/default.asp?url=/downloads/list/wmi.asp*

To obtain the .NET Framework redistributable package:
*http://msdn.microsoft.com/library/default.asp?url=/downloads/list/netdevframework.asp*

or

The Windows Update Web site
*http://www.windowsupdate.com*

For information on extracting the Windows Installer file from dotnetfx.exe:
Redistributing the .NET Framework
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetdep/html/redistdeploy.asp*

To obtain localized versions of the redistributable Dotnetfx.exe file:
.NET Framework Downloads
*http://msdn.microsoft.com/library/default.asp?url=/downloads/list/netdevframework.asp*

# 2

# Planning the Deployment of .NET Framework-based Applications

Once a Framework application is approved for deployment, you may be tempted to just start transferring files on to target computers. However, not all Framework applications can be deployed in the same way. Before you start on the deployment, you need to understand what exactly is going to be deployed, what your options are for deployment, and which of those options you will choose. This chapter will help you decide on the most appropriate deployment strategies for your organization, and explain the planning considerations for deploying Framework applications.

## What to Deploy with the Application

### Files and Folders

There are a wide variety of files that may be deployed with an application. The file types that are deployed will vary according to the nature and complexity of the application. The table on the next page lists some of the more common file types you will see deployed with a .NET-based application.

**Table 2.1: Files Types that May Be Deployed with an Application**

| File Type | Windows Application/Service | Web Application/Service |
|---|---|---|
| Executable | X | X |
| Dynamic Link Library | X | X |
| .NET Configuration Files | X | X |
| Databases | X | X |
| Web Pages | | X |
| Web Form | | X |
| Web Service Files | | X |
| Discovery Files for Web Services | | X |
| XML Schema Definition Files | | X |

**We will look at some of these file types in more detail:**

## Configuration Files

Configuration files are used to store information about everything from process models, through application settings, to security and authentication models. There are three main types of configuration files—Application, Machine and Security Configuration Files.

### Application Configuration Files

Application configuration files contain settings specific to an application. These files contain configuration settings that the CLR reads (such as assembly binding policy, remoting objects, and so on), and settings that the application can read. The name and location of the application configuration file depend on the application's host.

For an executable file, the configuration file resides in the same folder as the executable file, and will be called <*AppName*>*.exe.config*, where <*AppName*> is the name of your executable (for example, *MyApp.exe.config*).

For Web applications, the configuration file is named *web.config* and resides in the virtual directory for your application. Subdirectories of your Web application can also have a *web.config* file, and the pages in the subdirectory are governed by both the settings contained in that specific *web.config* as well as the settings contained in *web.config* for the parent application's virtual directory. The settings contained in the configuration file in the virtual directory in which the Web files exist take precedence over the configuration file in the parent application's virtual directory if the settings conflict unless the **allowOverride** attribute is specified in the parent configuration file.

**Note:** For more information see "Hierarchical Configuration Architecture" on MSDN, see the More Information section at the end of this chapter for details.

For Internet Explorer-hosted applications, a configuration file can be referenced by the **<link>** tag with the following syntax:

```
<link rel="ConfigurationFileName" href="<location>">.
```

**Note:** For more information about Internet Explorer-hosted applications, see "Configuring Internet Explorer Applications" on MSDN. For further details, see the More Information section at the end of this chapter.

#### Machine Configuration Files

The machine configuration file, *machine.config*, contains settings that apply to an entire computer. This file is located in the <runtime install path>\Config directory. *machine.config* contains configuration settings for computer-wide assembly binding, built-in remoting channels, and ASP.NET.

**Note:** Many settings, such as assembly redirection, can be specified at either the application level or at the computer level. In general, you should store application settings in the application configuration files (such as *web.config* or the configuration file for your executable file) rather than in the *machine.config* file. The *machine.config* file stores settings for the entire computer — this can lead to an environment where settings become difficult to manage. Although this is more of a design issue than a deployment one, it is nevertheless important to bear this in mind as you plan for the deployment of your .NET-based applications.

#### Security Configuration Files

Security configuration files contain information about the code group hierarchy and permission sets associated with a policy level. Security configuration files are set at the Enterprise, machine, or user level. Rather than modifying the security configuration files directly, you should generally use the .NET Framework Configuration tool (Mscorcfg.msc) or Code Access Security Policy tool (Caspol.exe).

#### Managing Configuration Files Across Different Environments

The most difficult issue concerning the deployment of configuration files is how to manage settings for your application as you deploy to different environments. For example, if you store Web service URLs for your Web references in your application configuration file, these settings are often different in the development, test, staging, and production environments. The challenge you face is how to deploy the appropriate settings to each environment.

To manage different configuration files for different environments, you should:

1. Add the configuration file that is used in your development environment to your Visual Studio .NET project. This file is named either *web.config* or *app.config*, depending on your application type. (Note that for ASP.NET applications, Visual Studio .NET automatically adds the *web.config* file to your project for you, when you create the application).
2. Create separate configuration files for the test, staging, and production environments, in addition to the application configuration file your developers use in the development environment. Name these files so that it is immediately obvious which environment they are to be used in. For example, if you are developing a Web project, you might name your files *test.web.config*, *stage.web.config*, and *production.web.config*.
3. Add each file to your project. They are then maintained in your source control system.
4. Include only the file you need for the specific environment to which you are deploying, and rename it to the required name as part of the deployment. The required name is either *web.config* for ASP.NET applications, or *<AppName>.exe.config* for Windows applications. You could deploy all of the other configuration files along with your application and it does not affect your application in any way. However, because these files are not actually needed, you should avoid deploying them along with the application for efficiency.

---

**Note:** Visual Studio .NET 2003 allows you to automate the process of deploying the correct configuration file with your application, if you use Microsoft Windows Installer packages to deploy the application. For more detail see Chapter 3, "Implementing the Deployment of .NET Framework-based Applications."

---

The deployment process used to install your application to the test environment should be as similar as possible to the process you use to deploy to the production environment. If you are deploying different configuration files to these environments, this raises the issue that the deployment of the production configuration file will not be fully tested before your roll out your application.

One approach to solving this issue is to actually deploy the *production* version of your configuration file to the test environment, rather than the test version. That way, you can ensure that it is distributed correctly. Your testers can then perform the supplementary task of replacing the production version of the configuration file with the test version *before* running the application in the test environment. This should be a separate step, in order to avoid introducing unanticipated issues into the deployment process.

### Specifying Remoting Information

One important piece of information that can be contained in configuration files is remoting information. Remoting in .NET-based applications allows different applications to communicate with one another, whether they reside on the same computer, on different computers in the same local area network (LAN), over a wide area network (WAN) link or the Internet. The applications can even be running on different Microsoft operating systems. The .NET Framework provides a number of services such as activation and lifetime control, as well as communication channels responsible for transporting messages to and from remote applications.

If you are deploying a distributed .NET-based application, remoting information will provide the application with the information necessary for it to function properly. To configure remoting, create a remoting configuration section and include it in your application's configuration file, *web.config* file, or *machine.config* file. You then need to ensure that the appropriate configuration file is deployed along with your application.

You must provide the following information to the remoting system to make your types remotable:

- The kind of activation required for your type.
- The complete metadata describing your type.
- The channel registered to handle requests for your type.
- The URL that uniquely identifies the object of that type. In the case of server activation, this means a URL that is unique to that type. In the case of client activation, a URL that is unique to that instance will be assigned.

---

**Note:** For more information about the remoting elements you can use in your configuration file, see "Remote Object Configuration" on MSDN.

---

Remoting information can also be specified directly in your client and server code. For more information on this, see "Programmatic Configuration" on MSDN.

## Web Services Files

Many of the issues affecting the deployment of Web applications also apply to Web service deployment—IIS settings may need to be deployed with the Web service, as could HTTP handlers and modules.

One slight difference between the deployment of Web services and the deployment of Web applications is that you are deploying the Web service file (.asmx) rather than Web forms (.aspx). A Web service may also have one or more dependencies in the form of dynamic link libraries (*.dll) which will also need to be deployed.

## Discovery Files

For developers to take advantage of existing Web services, they must query the Web Service Description Language (WSDL) contained in the Web service file. However, before they can do this they must be aware of the existence of Web services and be able to contact them. In reality, developers do not always know what Web services are available. Therefore, they are not able to access the Web service description unless you provide them with a discovery mechanism. That is where discovery files come in. You can provide a discovery mechanism for Web services by creating and deploying a static discovery file (.disco) with the Web service. Developers can then use this discovery file when they add a Web reference to their applications—they can view the contents of the file in the **Add Web Reference** dialog box provided by Visual Studio .NET and can link to the WSDL for your service from this file.

You can create static discovery files by browsing to your Web service and appending the "?DISCO" query string to the URL (for example, *http://www.somedomainname-123.com/somewebservice.asmx?DISCO*). The resulting XML can be saved as a .disco file. The following example illustrates the contents of a static discovery file for the *counter* Web service:

```
<?xml version="1.0" encoding="utf-8" ?>
<discovery xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="http://schemas.xmlsoap.org/disco/">
  <contractRef ref="http://www.contoso.com/Counter.asmx?wsdl"
               docRef="http://www.contoso.com/Counter.asmx"
               xmlns="http://schemas.xmlsoap.org/disco/scl/" />
    <soap address="http://www.contoso.com/Counter.asmx"
        xmlns:q1="http://tempuri.org/"
        binding="q1:CounterSoap"
        xmlns="http://schemas.xmlsoap.org/disco/soap/" />
</discovery>
```

The location of the actual discovery information is contained in the **<contractRef>** element—developers referencing your Web service in Visual Studio .NET can click a link in the **Add Web Reference** dialog box to view the contract for *somewebservice* (that is, the WSDL for your Web service).

---

**Note:** If you create your discovery files in this way as a predeployment step by using your development servers, you need to update the URLs in both the *ref* and the *docref* attributes of the **<contractRef>** section before you deploy them to your production servers, so that they point to the live Web services and not the development ones.

---

Once you have created the initial discovery file, you can update it to contain references to additional Web services. In this case you simply need to include the **<contractRef>** element with the discovery file for each Web service that you want to allow developers to discover. When they browse to your discovery file using the

Visual Studio .NET **Add Web Reference** dialog box, they will see contract links for each **<contractRef>** element. A simple way to add these elements to your DISCO file is to simply browse to each of your Web services with the "?DISCO" query string, and then copy and paste the **<contractRef>** element into the discovery file.

In many cases it may be appropriate to build a DISCO file that advertises all of the Web services you wish to make available on your server, and deploy that to the Web server's root directory. That way developers will always know where to go in order to see what Web services are available on a particular server. You may also wish to modify the default document for the Web server (usually Default.htm, Default.asp, or Default.aspx) so that it indicates the presence of a discovery file and its location.

After you create your DISCO file, you need to deploy it to the production environment. Because the discovery file is a simple XML-based text file, it does not have any special deployment requirements—it can deployed to the appropriate location on the production Web server with simple copy operations (such as XCOPY or FTP), or it can be included in a Web setup Windows Installer file.

---

**Note:** Discovery can also be achieved with a process known as dynamic discovery. If you create your Web services with Visual Studio .NET, a dynamic discovery file (.vsdisco) is created for you. ASP.NET can use this file to perform a search through all subfolders in your Web service virtual directory to locate and describe Web services. You should not deploy this file to the production environment, because you will lose control over which Web services can be discovered by other developers — you should use this file only on development computers.

---

## XML Schema Definition (XSD) Files

XML Schemas are documents that are used to define and validate the content and structure of XML data, just as a database schema defines and validates the tables, columns, and data types that make up a database. XML Schemas are stored in XML Schema Definition (XSD) files.

You only need to deploy XSD files in when your application or service needs to access the XML Schema. A common example of this is for XML Web Services. XML Web Services can pass data in the form of XML datasets, and in some cases, the dataset will include information about the structure of the data from the schema, and is known as a typed dataset.

If you are deploying an XML Web Service, and that Web service returns typed datasets, you also need to deploy the corresponding XSD file, as the WSDL (and hence the DISCO file) for your Web service will contain references to the XSD files that define the schemas for those datasets. When developers browse to your DISCO file from the **Add Web Reference** dialog box in Visual Studio .NET, they will be able to access the dataset schema definition by clicking on the *View Schema* link.

### J# Redistributable Package

If your application utilizes J#, then you will also need to deploy the Visual J# .NET Redistributable Package (vjredist.exe). Applications and controls written using the Microsoft Visual J#® development tool require the Visual J# .NET Redistributable Package to be installed on the computer where the application or control runs.

The Visual J# .NET Redistributable Package is available from the Visual J# media or by Web download from *http://msdn.microsoft.com/vjsharp.* The .NET Framework must already be installed if you are going to install Visual J# .NET Redistributable Package.

The Visual J# .NET Redistributable Package supports all platforms supported by the redistributable package for the .NET Framework. The Visual J# .NET Redistributable Package has no additional system requirements over and above the requirements of the redistributable package for the .NET Framework. The setup program of the Visual J# .NET Redistributable Package installs the same files on all supported platforms to support both server and client scenarios.

---

**Note:** The Visual J# .NET Redistributable Package is not intended to run applications written with other Java-language development tools. Applications and services built with Visual J# .NET will run only on the .NET Framework and will not run on any Java Virtual Machine.

---

## Assemblies

Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An assembly provides the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly. Assemblies can contain up to four different elements:

- **The assembly manifest** – every assembly, whether static or dynamic, contains a collection of data that describes how the elements in the assembly relate to each other. The assembly manifest contains this assembly metadata. An assembly manifest contains all the metadata needed to specify the assembly's version requirements and security identity, and all metadata needed to define the scope of the assembly and resolve references to resources and classes. The assembly manifest can be stored in either a Portable Executable (PE) file (an .exe or .dll) with Microsoft intermediate language (MSIL) code, or in a standalone PE file that contains only assembly manifest information.
- **Type metadata** – defines the types that the assembly contains (much as type libraries do for COM objects).

- **MSIL code** – implements the types you have defined. MSIL is a language used as the output of a number of compilers and as the input to a just-in-time (JIT) compiler. The common language runtime includes a JIT compiler for converting MSIL to native code.
- **Resources** – could include graphics or localized strings.

You can group all of these elements together in a single file, such as a .dll or an .exe. Alternatively, you can have different elements of your assembly reside in separate files. For example, you might want large graphics or seldom used types to reside in separate files—that way they will only be loaded when required. For this scenario, your PE file that resides within the assembly (.dll or .exe) will contain the assembly manifest and core MSIL code and type metadata, and the assembly manifest will link to the external resource file(s) for graphics and to compiled .NET module(s) for seldom-used types and their MSIL implementation.

---

**Note:** For information on how the CLR determines which assembly to load, see Chapter 4, "Maintaining .NET Framework-Based Applications."

---

## Strong Naming Assemblies

In cases where you want to be sure that an assembly is unique, you may choose to provide it with a strong name. Strong names consists of the assembly's identity—its simple text name, version number, and culture information (if provided)— plus a public key and a digital signature. The name is generated from an assembly file (the file that contains the assembly manifest, which in turn contains the names and hashes of all the files that make up the assembly), using the corresponding private key.

---

**Note:** For a strong name to be secure, you must make sure that you keep the private key corresponding to the name secure.

---

Strong names provide the following features:

- **Name uniqueness**. They rely on digital signatures to establish unique identities.
- **Protection of the version lineage of an assembly**. A strong name can ensure that no one else can produce a subsequent version of your assembly, unless you provide them with the source code and the private key. Users can be sure that the version of the assembly they load comes from the same publisher that created the version the application was built with.
- **Differentiation by the CLR between the versions of an assembly**. This allows the appropriate version to be used by the referencing application.
- **Integrity check by the CLR against the digital signature**. This ensures that the file has not been tampered with.

**Note:** You can strong name any assembly. However, any assembly that will be placed in the global assembly cache (GAC) must be strong named. For more information on the global assembly cache, see the Shared Assemblies section later in this chapter.

**Note:** Although a strong named assembly is signed, there is no requirement for the publisher to prove its identity to a third party authority. You can use the signcode.exe utility to make this a requirement for any assembly, regardless of whether it is strong named. For more information, see Assembly Security Considerations on MSDN.

By default, any application that calls a strong named assembly needs to be fully trusted, anything less than that and the assembly will not be run. The trust level of an application is defined in the application config file, or determined by the application running in a particular zone in Internet Explorer. If you wish an application to call a strong named assembly, you need to either ensure that the application is fully trusted, or modify the assembly to allow calls from partially trusted applications. This is achieved by adding the **AllowPartiallyTrustedCallersAttribute** Class to the assembly.

**Note:** There are security implications to using the **AllowPartiallyTrustedCallersAttribute** Class. For more information on the security implications of using this, see AllowPartiallyTrustedCallersAttribute Class

Another significant issue with using strong assemblies comes when they need to be updated. The version number of the assembly must be included when an application references a strong-named assembly, this means that if a new assembly is simply added using a file copy technique, the application will not just start using the new assembly. For more information on this and how to deal with it, see Chapter 4, "Maintaining .NET Framework-based Applications."

To give an assembly a strong name, you can use the Sn.exe tool to create a public/private key pair, and then strong name your assembly by setting the **Assembly** attributes of the **Application** project in Visual Studio .NET 2003. Alternatively, you can use the assembly linker utility (Al.exe) to strong name your assembly with the keys generated by Sn.exe after they are built. This latter approach is most useful if you need to delay the signing of your assemblies until after you have compiled the assembly for security reasons. When you delay the signing of your applications, you can build and test an assembly with only the public key. The private key can then be applied later when the assembly is due to be actually deployed.

**Note:** For more information on delaying the signing of assemblies see Delay Signing of an Assembly MSDN.

For more information about strong naming your assemblies with attributes, see "Signing an Assembly with a Strong Name" on MSDN.

## Private Assemblies

If an assembly is designed for the sole use of your application, you should deploy it as a private assembly. Private assemblies are usually deployed directly to the application base directory. The application base directory can be any of the following:

- The directory or subdirectory that contains the executable, if you have built a Windows Forms-based application.
- The bin folder located in the virtual directory, if you have built an ASP.NET Web application.
- The location specified in the **<codebase>** element of the application configuration file.

In most cases, you should deploy private assemblies to the application base directory as this is the first place that the CLR looks for these assemblies if they are not specified in the application configuration file.

Private assemblies are used only by the application for which they were developed, allowing for your application to be isolated from other applications. This helps to avoid problems with different versions of the same DLL being installed by different applications, and preventing one or more applications functioning properly—a situation commonly referred to as DLL hell.

## Shared Assemblies

In some cases the same assemblies are required by different applications. If this is the case for your applications, you have three deployment choices:

- Deploying the assembly with each application.
- Deploying the assembly in one directory and directing each application to use it.
- Deploying the assembly in the global assembly cache.

We will discuss each of these in turn:

### Deploying the Assembly with Each Application

If you deploy the assembly with each application, you would generally deploy the assembly in the same directory as the application. In this case each copy of the assembly is considered to be independent from the others.

Deploying the same assembly multiple times introduces some considerations you should be aware of. It is possible to run multiple versions of the assembly on the same computer, including those compiled under different versions of the Framework. This can be important if you are going to run multiple Framework applications side by side on a computer. However, running multiple versions of an assembly can introduce version control problems. If there is security issue with a particular assembly you may have to update the same assembly multiple times. This may give you significant management difficulties, depending on the deployment strategy you choose. It is therefore very important to ensure that you are fully aware of where the different copies of the assembly are installed.

**Note:** For more details on running applications side by side see Chapter 4, "Maintaining .NET Framework-based Applications."

#### Deploying the Assembly in One Directory

If you wish, you can place your assembly in one directory and direct each application to use it by modifying the **<codebase>** entry in each application config file, pointing to the location of the assembly in question.

If you use a single directory for your assemblies you lose the ability to run multiple versions of the assembly side by side. However with just a single copy of the assembly in place, you eliminate version control issues.

You should think carefully before deploying your assemblies to a single directory in this way, as there may be future situations where side-by-side operation becomes important, forcing you to change your deployment strategy at that time.

#### Deploying the Assembly in the Global Assembly Cache

The global assembly cache is a special location on computers running the Framework that allows multiple versions of the same assembly to reside side by side. Installing your assemblies into the global assembly cache provides the following advantages:

- A central point to distribute and manage code that is installed on the machine. Depending on your deployment strategy, it can be much easier to simply add a new version of a component to the global assembly cache than it is to update many copies that happen to be installed in numerous private application directories.
- Support for side-by-side installation of components and the Framework. For more details, see Chapter 4, "Maintaining .NET-Framework Based Applications."
- Better performance on loading strong-named assemblies. This is because signature verification is done on installation into the global assembly cache, whereas for private assemblies the check occurs on each assembly load.
- Better performance on loading multiple copies of assemblies. This is because multiple ASP.NET applications that load the same assembly will all map to the same in memory copy of the assembly if it was loaded from the global assembly cache. If that same assembly was loaded from the application directory, multiple copies of the assembly would be loaded in memory.

However, installing an assembly into the global assembly cache introduces the following issues that are not encountered with other assemblies:

- Installing an assembly into the global assembly cache requires administrative privileges by default, because the physical location of the global assembly cache is a subfolder of the Windows directory.

- Assemblies in the global assembly cache must be created with a strong name because the CLR performs integrity checks on all files that make up shared assemblies based on the strong name. The cache performs these integrity checks to ensure that an assembly has not been tampered with after it has been created (for example, to prevent a file from being changed without the manifest reflecting that change).
- You cannot simply copy an assembly into the global assembly cache and have your applications use it. The assembly must be installed in the global assembly cache. This can be done by using Windows Installer technology or by using the Gacutil.exe tool.

Do not assume that just because an assembly is required by multiple applications it should always be installed in the global assembly cache. The strategy you choose for deploying shared assemblies will depend on the specific requirements of your applications, but also the overall deployment strategy you use. Specifically, if you choose a no-touch deployment approach, using multiple private assemblies is usually more appropriate. For more details, see the No-Touch Deployment section later in this chapter.

In a production environment, you should always install assemblies into the global assembly cache with some mechanism that can maintain a count of the number of references to that assembly. This prevents premature removal of the assembly from the global assembly cache by the uninstall routine of another application, which would break any application that relies on that assembly. There are two ways to achieve this—using Windows Installer packages to install your assemblies into the global assembly cache, or by using the gacutil.exe tool with the **/ir** switch to add a traced reference. If you install an assembly using gacutil.exe, you should also use the same tool to remove it, this time with the **/ur** switch.

It is theoretically possible to add assemblies into the global assembly cache by dragging and dropping them into the cache Windows Explorer. However, this does not implement reference counting and should be avoided.

---

**Note:** Reference counting is only useful if it is always implemented. Any assembly that is installed without using reference counting will cause the reference count for that assembly to be inaccurate.

---

If you are developing components that may be used by other developers, you should always create strong names for them. That is because you do not know when you develop your assembly whether the other developers will wish to install the assembly in the global assembly cache. The other developer should not apply a strong name to your assembly after you create it, because then it cannot be verified that you (or your company) is the author of the component.

**Note:** The CLR attempts to locate assemblies in the global assembly cache before searching in the application base folder. Consequently, if you deploy a strong named private assembly with your application but it is already present in the global assembly cache, the CLR uses the shared version, rather than the private one, at run time. This should not lead to any problems, because assemblies with the same strong name should always be identical.

For more information about specifying assembly locations, see "Specifying an Assembly's Location" on MSDN.

For more information about how the CLR locates assemblies, see "How the Runtime Locates Assemblies" on MSDN.

For more information about redirecting assembly binding, see "Redirecting Assembly Versions" on MSDN.

### Windows Forms User Controls

As mentioned in Chapter 1, Windows Forms user controls are assemblies referenced from Web pages that are downloaded to the user's computer and executed upon demand. Windows Forms user controls can present particular deployment challenges because they run in the security context of the browser.

From a code access security perspective, there are two types of Windows Forms user controls: those that are run under the default security policy and those that require higher trust. To write Windows Forms user controls that are intended to run under the default security policy, you only need to know which operations are allowed by the default security policy for the intranet or the Internet zones, depending on where the CAB file is deployed. As long as a Windows Forms user control does not require more permission to execute than it receives as a result of its zone of origin, then it will run. To execute managed controls that demand higher trust, the administrator or user must adjust the security policy of any computer that runs the code.

Controls that require higher trust should always be strong named or signed with a publisher certificate. This allows policy administrators to grant higher trust to controls signed with a particular certificate without reducing their security with regard to other intranet/Internet code. After the assembly is signed, the user must create a new code group associated with sufficient permissions, and specify that only code signed by the user's company or organization be allowed membership to the code group. After the security policy is modified in this manner, the control has sufficient permission to execute.

Because security policy must be modified to allow high trust downloaded controls to function, deploying this type of control is much easier on a corporate intranet than on the Internet—there is usually an enterprise administrator who can deploy the described policy changes to multiple client computers on the intranet. In order for high-trust controls to be used over the Internet by general users, the user must be

comfortable following the publisher's instructions to modify the policy and to allow the high-trust control to execute. Otherwise, the control will not run.

For more information on using Windows Forms user controls in Web applications see "Host Secure, Lightweight Client-Side Controls in Microsoft Internet Explorer" on MSDN.

## Installation Components

Distributed .NET applications consist not only of traditional program files and assemblies, but also of associated resources, such as message queues, event logs, performance counters, and databases. These elements need to be created as part of your deployment process. The .NET Framework provides installation components that allow application resources to be created and configured when your application is installed, and removed when your application is uninstalled. These installation components integrate with widely available deployment tools, such as Installutil.exe, Windows Installer files (.msi), and the Windows Installer service. There are two types of installation components: predefined installation components and installer classes.

### Predefined Installation Components

Microsoft supplies five predefined installation components that can be used with applications. These are:

- **EventLogInstaller** – used to install and configure event logging.
- **MessageQueueInstaller** – used to install and configure message queues..
- **PerformanceCounterInstaller** – used to install and configure performance counters and monitoring.
- **ServiceInstaller and ServiceProcessInstaller** – used to install and configure Windows Services.

Each of these installation components are used by the application designer to ensure that the correct application resources are installed at deployment time. If application designers wish to use any of these application resources, they should include and configure the corresponding predefined Installation component in a Visual Studio .NET 2003 application project. When the component is added, this also adds an associated class, called **ProjectInstaller** to the project, which is used to install the component. The developer can then compile the project to a dll or an exe file, ready to hand off to deployment.

Then when the install occurs, the deployment project runs the **ProjectInstaller** class, which in turn launches the installation process for each of the components. By using a proper installation process for these resources, you ensure that all the appropriate settings, such as specific registry settings are installed for each component. The installation component also accesses the resource itself in order to retrieve the

properties of that resource (for example maximum queue size and journal recording settings in the case of a message queue). When you deploy your project to the production environment, it recreates the resource from your development environment and ensures that those settings are applied. All of these settings are stored in the code for the installation component—this happens automatically, so you do not need to manage the settings manually.

**Note:** You do not have to create and configure application resources in your development environment. If you wish, you can access the installation component directly in the **ProjectInstaller** class (or any class with the **RunInstallerAttribute** value set to **true**) and manually set the necessary values to create and install the resource in the state you desire.

For details on deploying pre-defined installation components, see Chapter 3, "Implementing the Deployment of .NET Framework-based Applications."

## Installer Classes

In some cases you need to add application resources for which there are no corresponding predefined installation components. You can add installer classes to your .NET-based application to perform specific actions during installation. For example, you can use installer classes to create databases required for your application, or you can use them to precompile a certain assembly to native code after the main installation is complete.

### Instrumented Assemblies

One example of an application resource that needs to be added using installer classes is an instrumented assembly. Instrumented assemblies integrate with Windows Management Instrumentation (WMI).

Instrumented assemblies need to undergo a registration stage, in which its schema can be discovered and registered in the WMI repository. Schema publishing is required on a per assembly basis. Any assembly that declares instrumentation types (events or instances) must have its schema published to WMI.

As with other application resources, it is the responsibility of the application developer to make sure that instrumented assemblies are installed successfully. If the project already contains a predefined installation component, you should use the **ManagementInstaller** class for your instrumented assembly. However, no predefined installation components have been added, the developer must add an installer class, the **DefaultMangementProjectInstaller** class, to the project. Then the developer can compile the assembly ready for deployment.

For more information about registering WMI schemas for your instrumented applications with the **ManagementInstaller** class, see "Registering the Schema for an Instrumented Application" on MSDN.

For information on deploying instrumented assemblies, see Chapter 4, "Maintaining .NET Framework-Based Applications."

## COM Components

Applications built completely on the .NET platform will eventually replace those developed with COM. However, until then, your developers may need to use or create COM components with Visual Studio .NET 2003.

COM objects that interoperate with your Framework applications must be registered on the target computer. Similarly, COM+ components that interact with your .NET-based applications must be properly installed into the COM+ catalog.

The main challenge in deploying COM components comes in ensuring that these components can communicate with .NET-based assemblies. This is because .NET-based assemblies are managed code and COM components are unmanaged code.

The interoperability features of .NET, known as COM Interop, allow you to work with existing unmanaged code in COM components along with Microsoft Win32® application programming interface DLLs. It also allows you to use managed components from your unmanaged, COM-based code.

When a .NET component is called from COM, the runtime generates a wrapper object to bridge the gap between the managed and unmanaged environments. In this case, the runtime reads the type information for the component from its assembly metadata and generates a compatible COM callable wrapper (CCW). The CCW acts as a proxy for the unmanaged object.

When a COM object is called from .NET, the runtime generates a runtime callable wrapper (RCW). Similar to the CCW, the RCW acts as a proxy between the managed .NET code and the unmanaged COM code. RCW and CCW are created at run time and so are not deployment issues. However, the RCW is created from type information that is stored in an interop assembly, which does need to be deployed with your application.

---

**Note:** For more information on COM wrappers, see the COM Wrappers section of the .NET Framework Developers Guide. For further details see the More Information Section at the end of this chapter.

---

An interop assembly, unlike other .NET assemblies, contains no implementation code. Interop assemblies contain only the type definitions of types that are already implemented in COM components. It is from these type definitions that the CLR generates the RCW to allow managed code to bind to the types at compile time and provides information to the CLR about how the types should be marshaled at run time.

There are two types of interop assemblies:

- **Primary interop assemblies (PIAs)** – These are interop assemblies that are provided by the same publisher as the type library they describe, and they provide the official definitions of the types defined with that type library. They are always signed by their publisher to ensure uniqueness.
- **Alternate interop assemblies** – These are any assembly that is not a PIA. They are not signed by the COM object publisher.

You should always use PIAs if they are available. This is because PIAs uniquely identify a type. Types defined within an interop assembly that is not provided by the component publisher are not compatible with types defined in the PIAs. For example, consider two developers in the same company who are writing managed code that will interoperate with an existing third-party supplied COM component. One developer acquires the PIA from the component publisher. The other developer generates his or her own interop assembly by running Tlbimp.exe against the COM object's type library. Each developer's code works properly until one of the developers tries to pass the object to the other developer's code. This results in a type mismatch exception; although they both represent the same COM object, the type checking functionality of the CLR recognizes the two assemblies as containing different types.

A PIA is created from a type library by running the TlbImp.exe utility with the **/primary** switch. If you do not have a PIA, alternative interop assemblies can be created by Visual Studio .NET 2003 (by setting a reference to a COM object) or by using Tlbimp.exe. If you use Visual Studio .NET to create a reference to a COM object, Visual Studio .NET looks in the registry to see if a PIA is registered on that computer for the COM object being referenced. If one is registered, it uses the PIA instead of generating one. If there is no PIA, Visual Studio .NET creates an alternative interop assembly.

---

**Note:** The Type Library Importer (Tlbimp.exe) converts most COM method signatures into managed signatures. However, several types require additional information that you can specify by editing the interop assembly. For more information about Tlbimp.exe, see "Type Library Importer (Tlbimp.exe)" on MSDN.

---

All PIAs are strong named. Alternate interop assemblies do not have to be, and if they are generated by creating a reference to a COM object in Visual Studio .NET 2003, they will not be strong named. If you want a strong named alternate interop assembly, you must manually create it, strong name it, and then set a reference to it. The interop assembly will vary according to the platform on which it is created, so you may have to create multiple interop assemblies depending on the platforms to which you will be deploying the assemblies.

**Note:** Several of the Windows system .dlls do not ship with Primary Interop Assemblies. If you wish your managed code to interact with these .dlls, you will need to create interop assemblies for this purpose.

For more information on deploying COM components and interop assemblies, see Chapter 3, "Implementing the Deployment of .NET Framework-based Applications."

For more information about proxies, see "Deploying Application Proxies" on MSDN.

## Serviced Components

.NET components rely on COM+ to provide them with component services such as transaction management, object pooling, and activity semantics. A .NET component that uses COM+ services is known as a serviced component.

Because your serviced components are hosted by a COM+ application, they must be accessible to that application. This introduces the following registration and configuration requirements for the serviced component:

- The assembly must be strong-named.
- The assembly must be registered in the Windows registry.
- Type library definitions for the assembly must be registered and installed into a specific COM+ application.

Serviced components can often register dynamically the first time they run. The first time a client attempts to create an instance of a serviced component, the CLR registers the assembly, the type library, and configures the COM+ catalog. Registration occurs only once for a particular version of an assembly. This is the easiest method for registering your serviced components, but it works only if the process running them has administrative privileges. Also, any assembly that is marked as a COM+ server application requires explicit registration, and dynamic registration does not work for unmanaged clients calling managed serviced components.

In many cases, the process that uses your assembly will not have the required privileges for dynamic registration. For example, if the assembly is used in a Web application, ASP.NET is not able to register your serviced component. ASP.NET does not run with administrative privileges by default (unless you set it to run as SYSTEM, which is not recommended for security reasons) so when your serviced component attempts to register, it will fail with an access denied exception. Under these circumstances you will need to ensure that the registration occurs at deployment time. For more information on how to achieve this, see Chapter 4, "Implementing the Deployment of .NET Framework-based Applications."

As serviced components take advantage of COM+ services, they have some of the same deployment considerations as COM+ components. These include:

- Any communication with the COM+ application occurs using DCOM by default, so you need to deploy interop assemblies to client computers just as you would for a traditional COM components.
- In addition to any COM+ settings that can be set using attributes on the assemblies themselves, you must make sure other configuration settings (such as adding users to roles and setting the process security identity) are created and assigned correctly. This can be achieved by adding a script to the custom action that registers the component.

**Note:** You can configure a COM+ application to use SOAP rather than DCOM. This circumvents the need to deploy interop assemblies. However, this approach does not allow you to flow transaction context from client to server. You would need to initiate your transaction at the remote serviced component.

## IIS Settings

ASP.NET applications require IIS in order to run. In some cases you will need to alter the settings of IIS to allow applications to run, or to control the environment in which they run. When you install ASP.NET on a server running Windows Server 2003, ASP.NET applications are executed by IIS 6.0.

IIS 6.0 runs in one of two distinct modes of operation, known as application isolation modes. These modes are:

- Worker process isolation mode (the default)
- IIS 5.0 isolation mode

When ASP.NET runs in the worker process isolation mode, it runs in the W3wp.exe worker process. In this case, the process model built into ASP.NET is disabled, and the worker process isolation architecture of IIS 6.0 is used instead. In worker process isolation mode, you can isolate anything—from an individual Web application to multiple sites—in its own self-contained World Wide Web Publishing Service (WWW service) worker process, preventing one application or site from stopping another. The processes are also completely separated from the core WWW service, Inetinfo.exe. Because these Internet Server (ISAPI) applications run separately from the WWW service, an application failure prevents all services hosted by the WWW service from failing. Only the worker process that hosts the ISAPI application is affected. Worker processes can be configured to run on specific CPUs, which allow you greater control of balancing system resources. Plus Web applications run with the Network Service identity, which provides a security advantage: the Network Service account has lower access privileges than LocalSystem.

If you configure IIS 6.0 to run in IIS 5.0 isolation mode, ASP.NET runs in its own process model, Aspnet_wp.exe, and uses its own configuration settings. When IIS 6.0 is in IIS 5.0 isolation mode, the worker process isolation architecture of IIS 6.0 is disabled and the process model build into ASP.NET is used for all ASP.NET applications on the computer.

You must use IIS 5.0 isolation mode for applications that conflict with worker process isolation mode until the applications are modified. The following application characteristics conflict with worker process isolation mode:

- Dependency on Inetinfo.exe. If the application must run in the Inetinfo.exe process, it must be run in IIS 5.0 isolation mode because applications do not run in Inetinfo.exe in worker process isolation mode.
- Requires Read Raw Data Filters. Read Raw Data Filters are available in IIS 5.0 isolation mode only.
- Requires Dllhost.exe. Applications that must be run in a Dllhost.exe environment can be run only in IIS 5.0 isolation mode because Dllhost.exe is not available in worker process isolation mode.

If the IIS 6.0 service is running in worker process isolation mode (the IIS 6.0 default mode), and you must run applications that do not meet the requirements for worker process isolation mode, you will need to switch to IIS 5.0 isolation mode. This means you will not be able to take advantage of worker process isolation and the other features of worker process isolation mode.

In IIS 5.0, settings for ASP.NET applications were stored in the **<processModel>** element of the *machine.config* file. This is still the case if you run the IIS 6.0 service in IIS 5.0 isolation mode. However, if you use IIS 6.0 in worker process isolation mode, only 3 settings in the **<processModel>** element are honored. These are:

- **maxWorkerThreads**
- **maxIoThreads**
- **responseDeadlockInterval**

All other settings are ignored. In some cases the other settings are irrelevant for IIS 6.0, but for others, there is an equivalent setting which should be specified in the IIS 6.0 metabase. For information on this, see "Mapping ASP.NET Process Model Settings to IIS 6.0 Application Pool Settings" on MSDN.

---

**Note:** For more information on the architectural differences between IIS 5.0 and IIS 6.0, see the "Technical Overview of Internet Information Services (IIS) 6.0" white paper.

---

### HTTP Handlers and HTTP Modules

ASP.NET provides **IHttpHandler** and **IHttpModule** interfaces that allow you to use application programming interfaces (APIs) that are as powerful as the ISAPI programming interfaces available with IIS, but with a simpler programming model. HTTP handler objects are functionally similar to IIS ISAPI extensions, and HTTP module objects are functionally similar to IIS ISAPI filters.

ASP.NET maps HTTP requests to HTTP handlers. Each HTTP handler enables processing of individual HTTP URLs or groups of URL extensions within an application. HTTP handlers have the same functionality as ISAPI extensions with a much simpler programming model.

An HTTP module is an assembly that handles events. ASP.NET includes a set of HTTP modules that can be used by your application. For example, the **SessionStateModule** is provided by ASP.NET to supply session state services to an application. You can also create custom HTTP modules to respond to either ASP.NET events or user events.

To deploy HTTP handlers and HTTP modules you need to first deploy the corresponding .NET class in the **\bin** directory under the application's virtual root and then registering the HTTP handler or module under in the *web.config* configuration file.

For more information about working with and registering HTTP handlers and HTTP modules, see "HTTP Runtime Support" on MSDN.

### SSL Settings

The SSL features in IIS cannot be used until a server certificate is obtained and bound to the Web site. For Internet (public) applications, you obtain a server certificate from a recognized third-party certification authority. For private (intranet) applications, you can issue a server certificate yourself. IIS associates the certificate with a particular IP address and port number combination.

You need to determine a way to bind your server certificate to your production Web site as you deploy your ASP.NET application. The most common way to do this is to manually bind the certificate using the IIS management console. If all of the virtual directories and Web sites on your Web server can use the same certificate, you can bind a server certificate at the Master Properties level for your Web server—this binds the certificate to every Web site that has not previously been bound to a certificate. If this is your first server certificate, it means that it is effectively bound to the entire Web server. Any new virtual directories or Web sites that you create as part of your deployment routines are bound with that server certificate.

For more background information about using SSL, see "Untangling Web Security: Getting the Most from IIS Security" on MSDN.

**Note:** ASP.NET is not available on Windows XP 64-Bit Edition and the 64-bit versions of the Windows Server 2003 family.

## Registry Settings

.NET-based applications should be less reliant on registry entries than previous applications. For example, assemblies do not require registry entries, unlike COM components in the past. However, your application might still rely on registry settings in some cases. These include:

- If your application includes pre-.NET based components such as COM, COM+, or Windows Services.
- If your assemblies must communicate with a COM component or pre-.NET services (The registry entries occur when you register the assembly during installation).
- If you wish to add registry entries to provide information such as licensing or versioning.

Registry entries may also be added by installation components.

For more information about deploying registry settings with Windows Installer files, see Chapter 3, "Implementing the Deployment of .NET Framework-based Applications."

## Merge Modules

Merge modules are reusable setup components. They cannot be installed directly—instead they are merged into a Windows Installer package for each application that uses the component. Much as dynamic-link libraries allow you to share application code and resources between applications, merge modules allow you to share setup code between Windows Installer files. This ensures that the component is installed consistently for all applications, eliminating problems such as version conflicts, missing registry entries, and improperly installed files. Merge modules may also be used to deploy third-party components as part of your application deployment.

Merge modules can only be used in conjunction with Windows Installer files and cannot be deployed in any other way.

For more details on creating and deploying merge modules see Chapter 3, "Implementing the Deployment of .NET Framework-based Applications."

For more information about merge modules, see "Introduction to Merge Modules" on MSDN.

## CAB Files

CAB files provide a means of packaging together the files needed by an application so that they can be distributed and installed more easily.

Creating CAB files provide the following advantages:

- They can be downloaded and any managed controls that they contain can be extracted and executed upon demand.
- They support a number of compression levels, allowing for a quick download time.
- You can use Microsoft Authenticode® technology to sign your CAB files so that users will trust your code.
- They have zero client footprint (with the exception of the download cache).
- They contain controls that can easily be updated simply by repackaging the new version into a CAB and replacing the existing copy on the Web server. The CAB projects support versioning, so you can ensure that end users are always using the most recent copy.

In some cases you may need to deploy CAB files that have been created for other applications. In others you may choose to create CAB files yourself to support the distribution option you have chosen for your application.

For more details on creating and deploying CAB files, see Chapter 3, "Implementing the Deployment of .NET Framework-based Applications."

For more information about managing the security requirements for downloaded CAB files, see "Writing Secure Managed Controls" on MSDN.

## Localization

The CLR provides support for retrieving culture specific resources that are packaged and deployed in satellite assemblies. Satellite assemblies contain only resource files, or loose resources such as .gif files. They do not contain any executable code.

In the satellite assembly deployment model, you create an application with a default assembly (the main assembly) and several satellite assemblies. You should package the resources for the default language neutral assembly with the main assembly and create a separate satellite assembly for each language that your application supports. Because the satellite assemblies are not part of the main assembly, you can easily replace or update resources corresponding to a specific culture without replacing the application's main assembly.

---

**Note:** If your main assembly uses strong naming, satellite assemblies must be signed with the same private key as the main assembly. If the public/private key pair does not match between the main and satellite assemblies, your resources will not be loaded.

---

If you have many different languages for which you need to localize your application, you can streamline the packaging and deployment process by:

- Packaging your core, language neutral files into a merge module (MSM) using a Visual Studio .NET setup and deployment project.
- Creating a project that contains the localized resources for *all* languages that you want to support.
- Creating a separate Windows Installer file for each language you are supporting, using Visual Studio .NET setup and deployment projects. This allows you to localize the installer as well as the application.
- Creating a base installer file for the primary language you wish to support and then creating transforms for each additional language that you wish to support.
- Adding the output for the project that contains the localized resources into each Windows Installer project. That essentially includes *all* of the localized resources. You can then use the **ExcludeFilter** to filter out all but the one localized resource that you need for your specific language.

As an alternative, instead of filtering out the localized resources that are not needed, you can simply distribute them all. That way, your clients can use the different localized resources on the same computer; for example, they can change their locale settings and have the appropriate resources loaded by your application.

Another approach is to create a core installer and a separate set of localized installers (typically known as language packs). Your global customers can then install your core application and then one (or more) of your language packs.

---

**Note:** For more information Satellite Assemblies, see "Creating Satellite Assemblies" on MSDN.

---

## Debug Symbols

When developers build an application in Visual Studio .NET 2003, they create two default configurations for the project: release and debug. One of the main differences between these two configuration modes is the way that debugging symbols are managed. In the debug configuration, a debugging symbols file program database is created, whereas for the release configuration, debugging information is not generated by default.

The program database file contains the information needed to map the compiled MSIL back to the source code; this allows debugging tools to fully report information such as variable names. In addition, the JIT compiler can generate tracking information at run time to map the native code back to the MSIL. Both tracking information and symbol files are needed to effectively debug managed code.

You should not distribute the symbol files with your application to the production environment, because this allows customers or other users to more easily reverse engineer your application (which is obviously a security concern). However, you should strongly consider changing the release configuration to ensure that it does generate debugging symbols. If you have not generated debugging symbols for the release version of your application, but you find that you need to debug it in the production environment, you need to rebuild the application, including debugging information, and redeploy the application to be able to debug it in the production environment. If you generate the debug symbols file with your release configuration, but keep them secured separately from the deployed application, you can install them into the production environment if (and only if) you need to debug the released application.

In Visual Studio .NET 2003, generation of the program database file and tracking information is controlled by the **Generate debugging information** option on the build page of the configuration section, in the project property pages of your projects. You can specify that the release configuration should generate the debugging symbols files by modifying the project property pages when the **Release** configuration is selected. If you compile your application with the command line, you can specify the **/debug** switch with various options to control the generation of debugging symbols and tracking information.

To help ease the management of symbols across your enterprise, you can use Microsoft Symbol Server to store your symbols in a centralized location and have the debuggers access them when they need them.

For more information about symbol management, see the following:

- Bugslayer: Symbols and Crash Dumps
- How to Get Symbols
- INFO: Use the Microsoft Symbol Server to Acquire Debug Symbol Files
- INFO: PDB and DBG Files–What They Are and How They Work
- Production Debugging for .NET Framework Applications

# Choosing a Deployment Strategy

Exactly which method of deployment you choose will depend to a large extent on the nature of your organization, which deployment technologies are available, and the type of the application you are deploying. However, you should ensure that you complete the planning of your deployment before you start any deployment steps, because a different distribution method may dramatically affect how you would package that application.

One of the main factors that determines how an application will be deployed is the nature of the application, in other words is the application Windows Forms-based, or Web-based?

There are three main methodologies for deploying applications:

- No-touch
- Windows Installer package
- File copy

We will look at each of these in turn:

## No-Touch Deployment

No-touch deployment is an increasingly popular feature of the .NET Framework used to package Windows Forms-based applications. In this scenario you store the application files (such as executables and DLLs) on a Web server. To install the application, the users connect to the application location using HTTP. The initial files and assemblies that are needed when the application is first run are downloaded to the .NET Framework assembly cache download folder (<windir>\assembly\download\) and the Temporary Internet Files folder. Each additional resource used by your application is automatically downloaded to the client computer, and stored in one of these two locations.

The advantage of this approach is that you can combine all of the richness of a traditional Windows graphical user interface with the manageability and maintainability of Web applications. As resources are downloaded on an "as needed" basis, download time for the initial run of the application is minimized. Then, when other assemblies are needed later, they are downloaded at that time. All of this happens automatically—when your executable file requests a class from another assembly, .NET Framework locates that assembly in the same location on the Web server and downloads it.

Although the application is effectively running from the .NET cache, you can still update your application files on the Web server, and those changes will take place immediately. Before the user's computer loads the assembly, it verifies that the downloaded application files have not changed since it last requested them—if they have, it downloads the later versions as needed.

There are some situations where no-touch deployment is inappropriate. These include:

- Where you need to predict and throttle bandwidth usage. With the application being downloaded on an as-needed basis by users, there is the potential unpredictable spikes in bandwidth usage.
- Where clients need to use the application offline. No-touch deployment applications store the application in the temporary Internet Files folder. Therefore, while

it is possible to run the application offline, it will only run those features that have already been downloaded and will not look for updates in offline mode. A good example of this is, you use a word processor with this method, but don't use the spell checker. If you then take this offline, you will still have the word processor cached, but the spell checker function will not be included (because it was never downloaded on demand) and will not work when you ask the word processor to check your spelling. This is true even if you are connected to the network as the browser is running in offline mode.

- Where advanced install operations are needed, such as a driver install or a COM object to be registered.
- Where the application cannot run within the bounds of the default security policy and deploying a different security policy is not practical.
- Where assemblies need to be deployed in the global assembly cache.

If you wish to deploy applications using no-touch deployment, it is a good idea to start planning for the deployment at the application design phase. This allows you to resolve any issues (for example, with security policy) at the design phase rather than attempting to do so as the application is deployed.

---

**Note:** You can use a network share rather than a Web server to host your .NET applications. In this case, the users connect to the share and start the executable file from there. This approach is similar to the URL-launched executable file, except that the download cache is not used to cache the downloaded application and assemblies — all assemblies are loaded directly into memory as needed. The absence of caching makes the network share approach less efficient than the Web server approach.

---

### Deploying Shared Assemblies

No-touch deployment does not directly support installing shared assemblies in the global assembly cache. This means that if you wish to install assemblies in the global assembly cache you will need to do so using another mechanism. However, before adopting this strategy you should consider carefully if you need to deploy your assemblies in the global assembly cache at all. A good alternative to using the global assembly cache is to install the assemblies separately with each application. In some cases this can produce manageability problems, but in the case of no-touch deployment these management issues are compensated by the fact that if an assembly is modified, it only has to be updated on the Web server, and will then be downloaded on each client machine the next time the application is run.

### Multi-Assembly Applications

Applications consisting of single assemblies are the easiest to deploy using no-touch deployment. However, it is also possible to deploy applications consisting of multiple assemblies, although this can require some work by the application developer.

The **Assembly** class as specified in the .NET Framework Class Library has a **LoadFrom** method to initialize a reference to a particular assembly. The parameter for **LoadFrom** is either a URL or a file pathname. When a URL is specified, the .NET Framework first checks to see if the named assembly exists on the client by checking the assembly download cache. If the assembly is not in the assembly download cache, it is fetched from the Web server and a copy is placed in the download cache. Then the assembly object is ready for use in the code.

The advantage of this method is that the assembly is only downloaded if it is invoked by the user, although of course this can lead to a delay when that functionality it required by the user.

For more information about no-touch deployment of .NET-based applications, see the MSDN articles, "Death of the Browser?" and "Security for Downloaded Code." For details, see the More Information section at the end of this chapter.

## Windows Installer Package Deployment

In cases where a no-touch deployment will not be appropriate, you would normally look to deploy your applications using Windows Installer packages. Windows Installer packages can be used to deploy all kinds of .NET-based applications, along with merge modules and CAB Files. The Windows Installer packages can be distributed in a number of ways, including placing the installer packages on a file or Web server, using the Group Policy functionality of Active Directory, or using the software distribution capabilities of SMS. Advantages of deploying an application using Windows Installer include:

- A simple-to-use GUI installation program that can be customized
- Integration with Control Panel's Add/Remove Programs utility for:
  - Installing
  - Uninstalling
  - Adding or removing application features
  - Repairing a broken installation
- A setup routine that:
  - Can run in silent mode, with no user interaction
  - Rolls back the system to the state it was in prior to the start of installation if any part of the setup routine fails
  - Rolls back the system to the state it was in prior to the start of installation if the user cancels the installation part way through the setup routine

Using Windows Installer packages, you can control almost every aspect of application installation. A Windows Installer package can:

- Run prerequisite hardware and software checks prior to installation
- Create desktop and **Start** menu shortcuts for running the application
- Manage file and folder locations
- Manipulate the Registry
- Create File Associations
- Install COM and COM+ Objects
- Install assemblies into the Global Assembly Cache
- Run custom tasks after installation is complete
- Maintain version information, ensuring that patches and upgrades are installed in the correct order

Using Windows Installer packages can resolve many of the issues surrounding deployment of more complex Windows-based and Web-based applications. The following table describes some of these issues and explains the advantages of using Windows Installer files for deploying solutions that include these features.

**Table 2.2: Installation Issues for More Complex Windows and Web Applications**

| Feature | Installation Issues |
|---|---|
| Shared Assemblies | Windows Installer files provide an easy and reliable mechanism for installing assemblies into the global assembly cache. |
| Legacy COM Components | If your Web application includes legacy COM components, they need to be installed and registered properly before they can be used. Windows Installer files provide an easy and reliable mechanism for installing and registering COM components. However, you can also use RegSvr32 to manually register your COM libraries. |
| IIS Settings | Simply copying build outputs for your Web applications does not copy IIS settings from your development computer to the production Web server. You need to manually change any settings that need modifying or develop a script for ensuring the correct settings are in place. An easier approach is to use a Web setup project to package your solution in Windows Installer files — you can specify IIS settings for your Web setup project to have them applied when you run the installer package. |
| Application Resources | Windows Installer files provide an easy way to deploy application resources, such as message queues, event logs, and performance counters, along with your project outputs. You can use predefined installation components provided by Visual Studio .NET in your project to deploy this type of application resource. After you add the required predefined installation components to your application's project, you can easily add them to the Windows Installer project as custom actions. Similarly, if you have created installer classes to handle the deployment of other application resources for which there are no predefined installation components (such as serviced components), you can also add them to your setup project as custom actions. |

### Using Windows Installer Packages for Multi-Tiered Applications

For a multi-tiered application, you should build separate installers for each different physical tier. Building separate installers is appropriate for most scenarios because it is extremely complex (and in some cases impossible) to run an installer on one specific physical tier and have other physical tiers deployed to other computers as part of the same installation process. For example, if you are running the installer on your Web server, it is difficult to deploy your business logic components to a separate computer. In some cases different physical tiers may even be suited to different packaging strategies. For example, the Web tier might best be deployed as a collection of build outputs, while the business logic tier and database server might best be packaged in one Windows Installer file, with perhaps a Windows Forms-based application packaged in another installer.

### Security Considerations

One issue that affects deployment of Windows Forms-based applications to users with Windows Installer files is whether the user running the setup will have the required privileges to complete the installation. The privileges required depend on the actions that the Windows Installer file performs and the platform that users are installing your application onto. For example, no special privileges are required to install applications on Windows 95, Windows 98, or Windows Me, whereas even creating an application folder beneath the Program Files system folder on Windows 2000 or Windows XP by default requires that the user to be a member of a local group with elevated privileges, such as Power Users or Administrators.

When a user who is not a member of the local Administrators group attempts to run a Windows Installer file on Windows 2000, the user is first prompted that he or she may not hold sufficient permissions to complete the installation. They are also offered the choice of running the installer file as a different user who does have all required permissions (Administrator by default). To be able to run the installer file as a different user, the person running the installation needs to know the password for that account.

One way to ensure that your Windows Installer file will not fail due to insufficient privileges is to distribute it with an electronic software deployment tool, such as SMS or Active Directory Group Policy for software distribution. Both of these tools allow you to run your Windows Installer file with administrator privileges, regardless of the privileges held by the user logged on to the computer.

### Distributing Windows Installer Packages

There are a number of ways that Windows Installer packages can be distributed to users or computers. These are:

- Group Policy functionality of Active Directory
- Systems Management Server (SMS)
- Other Methods (including placing files on a Network Server, Web Server, or Distributing media)

We will look at each of these in turn:

#### Group Policy Functionality of Active Directory

Active Directory allows you to distribute applications to users or computers automatically using Group Policy. Group Policy Objects can be defined at a domain level, site level, OU level, or local computer level. Using Group Policy you can ensure that applications are installed automatically when a specific user starts their computer or logs on, or you can ensure that the applications appear automatically in Add/Remove Programs.

Group Policy allows you to distribute applications in two ways—assigning and publishing. The table compares the two methods.

**Table 2.3: Assigning and Publishing Applications with Group Policy**

| Distribution Strategy | Description |
|---|---|
| Assigning Software | You can assign a program distribution to users or computers. If you assign the program to a user, it is installed when the user logs on to the computer. When the user first runs the program, the installation is finalized. If you assign the program to a computer, it is installed when the computer starts, and is available to all users that log on to the computer. When a user first runs the program, the installation is finalized. |
| Publishing Software | You can publish a program distribution to users. When the user logs on to the computer, the published program is displayed in the **Add/Remove Programs** dialog box, and it can be installed from there. Alternatively you can specify that the program is installed when a user attempts to open a file type associated with the application. |

To allow you to install application using Group Policy, your destination computers must be running Windows 2000 or greater and be part of an Active Directory environment.

You do not have to use Windows Installer packages in order to use Group Policy as a distribution mechanism, however it does provide you with the most flexibility when choosing your distribution options. The Windows Installer packages can be assigned to users or computers, or published.

► **To assign a Windows Installer (MSI) package to computers using Group Policy**

1. Create a folder to hold the MSI package on a network server. Share the folder with appropriate permissions to allow the users and computers to read and run these files, and then copy the MSI package file into this location.
2. Start the **Active Directory Users and Computers** snap-in.
3. From **Active Directory Users and Computers**, click the object that contains the computers you wish to assign the application to. Right-click that container, click **Properties**, and then click the **Group Policy** tab.

4. If necessary, create a new GPO for installing your MSI package, and give the new GPO a descriptive name (you could also use an existing GPO).
5. While the GPO is selected, click **Edit**. This starts the Group Policy snap-in and lets you edit this GPO.
6. Expand Computer Configuration, Expanding the **Software Settings** folder and selecting the **Software Installation** icon, Right-click **Software Installation**, choose **New**, and **Package…**
7. You are prompted for the path to the Windows Installer file (.msi) for this package. View the network location that contains the Windows Installer file, click the file, and then click **Open**.

   **Warning:** If the Windows Installer file resides on the local hard disk, do not use a local path, instead, use a UNC path (such as *servername**sharename**path**filename.msi*) back to the local computer to indicate the location of the installation files. Otherwise, client computers that try to install the package will look on their local hard disks in the location that was indicated, and will not find the installation files at that location, so the installation does not work.

8. When prompted to choose between **Assigned, Published**, and **Advanced Published or Assigned**, click **Assigned.** Click **OK**.

   **Note:** Choosing **Advanced** will bring up another window that will allow you to make modifications to the delivery of the package. This can include upgrades, applying transforms and security settings. Of course these advanced settings can also be modified after you choose **OK** by selecting the package, right-click and choose **Properties**.

9. Close the Group Policy snap-In.
10. Close Active Directory Users and Computers.

**Note:** For more information on how to create Windows Installer packages, see Chapter 3, "Implementing the Deployment of .NET Framework-based applications.

#### Systems Management Server (SMS)

Systems Management Server 2.0 provides some significant functionality aimed at making software distribution easier and more accountable. If you already have SMS in your environment, you should strongly consider using it for distributing your .NET-based applications. If you do not currently have SMS deployed, you should consider the benefits it will bring to your organization. These include:

- **Software distribution to multiple client platforms**. All Windows-based platforms are supported by SMS.
- **WAN-aware automated distribution of software**. SMS can monitor its bandwidth usage and throttle it if it is using more than it should at a particular time of day. This allows you to ensure that software is not installed over the WAN at inappropriate times.

- **Targeting to specific users and computers**. Software can be distributed based on user names, group names, computer names, domain names, network addresses, or inventory collection values.
- **Scheduled application deployment**. Software can be distributed at a specific time. This may be useful to reduce the load on the network at certain times of day.
- **Status of application deployments**. This allows you to gain an enterprise view of the application deployment.
- **Reporting of successes/failures**. Just because an application should have been deployed to a particular client does not necessarily mean that the deployment was successful. SMS will report if a deployment succeeded or failed, allowing you to take action if there were problems in distributing the software to particular clients.

SMS also has a number of other features that are closely related to the successful distribution of applications. These include:

- **Inventory management**. Provides you with a record of the existing hardware and software deployed throughout your enterprise. Inventory information is a critical part of effective change and configuration management, allowing you to properly plan and deploy applications.
- **Application usage tracking (metering)**. Allows administrators to make smart purchasing and deployment decisions because they understand what applications their customers are using.
- **Status and reporting**. Allows administrators to monitor activity to ensure that the correct software to the correct locations when needed.

SMS does not require that you use Windows Installer packages, as it has its own packaging mechanism. However, you can still use SMS to deploy Windows Installer packages to clients.

For more instructions on using SMS to deploy applications, see the product documentation (*http://www.microsoft.com/smserver/techinfo/productdoc/default.asp*).

#### Other Methods

There are a number of other methods that are used to distribute Windows Installer packages. The following table outlines their uses:

**Table 2.4: Methods of Distributing Windows Installer Packages**

| Method | Description | Strengths/Weaknesses |
|---|---|---|
| Web/FTP Server | Installer package is placed on a Web server and the link sent out to users so they can download it. | Good way of making packages available over the Internet and intranet.<br>Package can be zipped to prevent over the network install.<br>No control over when downloads occur.<br>Local Admin rights required to install package. |

| Method | Description | Strengths/Weaknesses |
|---|---|---|
| Network Server | Installer package is placed on a network server and the link sent out to users so they can download it. | Good way of making packages available internally.<br>Package can be zipped to prevent over the network install.<br>No control over when downloads occur.<br>Local Admin rights required to install package. |
| E-mail | Installer package sent through e-mail system. | Easy to find package.<br>E-mail system may block installer package.<br>Can place considerable load on e-mail system.<br>Local Admin rights required to install package. |
| CD/DVD | Installer package burned to CD/DVD | Portable media.<br>Suitable where bandwidth is low.<br>Local Admin rights required to install package. |

## Choosing Between Group Policy and SMS for Distribution

Although there are a large number of methods for distributing Windows Installer packages, often the decision comes down to a straight choice between Group Policy and SMS. You may have Active Directory deployed within your environment, but are considering using SMS in the future for application distribution. The following table will help you decide if SMS is required to meet your Application distribution needs.

**Table 2.5: Choosing Between Group Policy and SMS for Distribution**

| Issue | Active Directory Group Policies | SMS |
|---|---|---|
| Reporting | Windows Installer events and messages are stored on the local computer, rather than in a central location. If an administrator requires reporting on which users or computers were updated, they need to connect to each computer and view the event logs. | SMS provides for centralized reporting and management. |
| Distribution | Active Directory requires an administrator to manually create and manage software distribution points. If multiple installation points are required, administrators need to synchronize them. Windows 2000 Distributed File System (DFS) can be used to streamline this process.<br>Each distribution point requires a separate Group Policy. | SMS uses a distribution point hierarchy which is automatically synchronized and managed. SMS also compresses the packages prior to delivery to remote install points. |

*(continued)*

| Issue | Active Directory Group Policies | SMS |
|---|---|---|
| Targeting | Targeting with Group Policy is based on organizational unit membership and policy application. | SMS targets collections based on inventory values. Collections are dynamic and query-based. |

## Automating the Creation of Installers

If you are an independent software vendor (ISV), or if you need to deploy a large number of .NET-based applications for your enterprise, you might want to automate the creation of your setup routines. For example, you might want to build your own application for creating setup programs to standardize deployment practices, or you might want to write scripts that create installers for multiple applications. If the other factors described in this chapter bring you to the conclusion that an .msi (or .msm) file is appropriate for your applications, you can automate the creation of your setups by using the automation interface provided by the Windows Installer, rather than creating your installers using Visual Studio .NET.

For in-depth information about the Windows Installer automation interface, see the "Automation Interface" section of the Platform SDK on MSDN .

## Tools for Creating Windows Installer Packages

In addition to the deployment tools included in Visual Studio, installation tools that support Windows Installer are available from third-party vendors. These tools may support additional Windows Installer authoring features that are not available in Visual Studio deployment projects.

Third party tools for creating Windows Installer Packages include:

- **InstallShield Developer**. A Windows Installer setup-authoring solution that provides complete control of the Windows Installer service and full support for .NET-based application installations. The Developer edition provides the option to create .NET installations directly from within the Microsoft Visual Studio .NET integrated development environment (IDE) or from the traditional InstallShield Developer IDE.

  InstallShield Developer features Visual Studio .NET project wizards, dynamic links with Microsoft Visual C#® .NET development tool and Visual Basic® .NET development system projects, .NET COM interop support, .NET assembly installation configuration, and .NET Framework distribution in addition to a Windows Installer direct table editor. For more information on this product, see the InstallShield Web site (*http://www.installshield.com/isd/*).

- **Wise for Visual Studio .NET**. A tool that operates directly within Microsoft Visual Studio .NET. It merges the installation and application development lifecycles so that applications and installations are designed, coded, and tested together. For more information, see the Wise Web site (*http://www.wise.com/visualstudio.asp*).

## Deploying a Simple Collection of Build Outputs

For many Web applications, and some more simple Windows-based applications, it is appropriate to use a file copy mechanism to deploy the application to a server. In this case you deploy your application as a simple collection of build outputs, rather than by applying any further packaging. The term *build outputs* refers to the application files that comprise your application, such as ASPX files, executables, DLLs, configuration files, graphics, and other resources.

The advantages of packaging your application files as a collection of build outputs include:

- **Ease of deployment**. The build outputs and other files can simply be copied to the target computer.
- **Ease of update**. Updated files can simply be copied once again to the target computer.

Unlike Windows Forms-based applications, Web applications are usually installed by an administrator or other skilled IT professionals. In many cases, they are **not** installed, uninstalled, or repaired using Add/Remove Programs in Control Panel. Additionally, rolling back the installation is not usually required if a problem occurs within the setup routine—it is often easier for the administrator to fix minor problems, such as modifying IIS settings or managing the failed copy of a certain file, than it is to roll back the entire process and start again.

However, while this type of deployment can work well for more simple applications, you would preferably use one of the other techniques if you need to do any of the following:

- Make changes to the Registry
- Add, remove, or change Windows Services
- Make security policy changes
- Change service or user accounts
- Change COM components or updates to strong named assemblies (Private or global assembly cache)
- Ensure that all dependencies are noted and registered

For more complex applications, such as those that include shared assemblies or those that rely on specific IIS settings to be in place, using Windows Installer technology may still be a better choice.

## Distributing a Simple Collection of Build Objects

Distribution mechanism for a simple collection of build objects can include any of the following:

- Microsoft Application Center 2000 (for Web farm deployment)
- Copy Project functionality of Microsoft Visual Studio .NET 2003 (only suitable for Web-based applications)
- File Copy Distribution

We will discuss each of these in turn:

### Microsoft Application Center 2000

If you are deploying Web-based or server-based applications, in many cases the content and components will exist on a Web cluster of up to 12 servers. When deploying your application you need to ensure that the software is distributed to all of the servers in the cluster and synchronized across each element of the cluster. You can use Application Center to distribute and synchronize any of the following throughout your Web cluster:

- Web sites, virtual directories (and their associated ISAPI filters), and ASP.NET applications (including XML Web Services)
- Configuration files (such as *web.config*, application configuration files, and machine.config)
- Private assemblies
- Discovery files
- Symbols
- HTTP modules and HTTP handlers
- Localized content
- Secure Sockets Layer (SSL) Certificates
- Server Configuration information (such as IIS metabase settings, CLB configuration, and NLB configuration)

---

**Note:** Many of these elements require administrative privileges to be installed. Application Center will manage this, pushing the components out to the appropriate machines with no requirement to log on locally to each machine.

---

Application Center is designed for managing and deploying content and components to a server environment. It is best suited for deploying build outputs and content files for Web applications and COM+ applications.

You can also use the deployment and synchronization features of Microsoft Application Center to manage SSL certificates. Application Center detects that your application relies on a certificate and automatically configures each member of your cluster accordingly.

**Note:** Certificates should be backed up onto portable media and stored in a well-known place for emergencies or for configuring new servers.

By contrast Application Center is not suitable for deploying applications packaged in Windows Installer files or other executable setup programs (for example desktop applications, Windows service applications, or other client-based applications). Nor is it designed to deploy SQL Server databases and manage their schemas, or message queues. If you are deploying these types of technologies, you must consider using other distribution mechanisms.

By default, Application Center cannot install assemblies into the .NET global assembly cache for members of your Web farm or application farm clusters. However, a patch to resolve this issue has been published. For more details see the Microsoft Knowledge Base Article 326950, "INFO: Application Center 2000 GAC Replication Support for Windows 2000". This functionality will also be included in Application Center 2000 Service Pack 2.

If you are going to use Application Center 2000, you must install the product on all the servers in your Web farm or application server farm. Once Application Center is deployed, it will automatically configure your server applications (and their constituent parts) when you distribute content to cluster members. This prevents you from having to specify detailed actions for your deployment. For example, if you replace an existing COM component using Application Center, it will stop the IIS Service, remove the registration of the old component, replace the component files, register the new component and restart the IIS Web Service. Without Application Center you would need to perform each of these tasks manually, or script them.

One of the major advantages of using Application Center to deploy Web applications and server applications in your clustered server environment is that you can use it to manage the different stages of your application lifecycle. You can move your applications from your development environment to your test servers, from your test servers to your staging environment, and finally from your staging server(s) to your production cluster with the same mechanism. Using Application Center to move your solutions between these different environments provides you with consistency advantages. For example, having successfully moved your application from development to the test environment, and then from the test servers to your staging servers, you can have confidence that, as you deploy your application to the final production environment, no unexpected changes or problems are introduced that arise from the actual process of distributing the application to the production environment. This makes it easier to track down any issues that arise in the live environment.

#### Visual Studio 2003

If you are distributing a Web application, you can use the **Copy Project** command in Visual Studio .NET 2003 to create the new Web application on a target server. You can choose to include:

- Only the files required to run to the application, including build outputs, DLLs and references from the bin folder, and any files with the **BuildAction** property set to **Content**.
- All project files, including build outputs, DLLs, and references from the bin folder, and all other files.
- All files in the project folder and subfolders.

The latter two options include the project file and source files. For this reason, when deploying to a production environment, you should generally choose the first option. However, this does mean that although a new virtual directory is created on the target server, IIS settings will not be copied from the original environment to your target server. Instead the new virtual directory inherits the default settings from the Web site. You will need to apply the appropriate settings separately, either by developing and running IIS scripts or by manually applying the setting your Web application requires.

When using Copy Project, the target server must be prepared to receive the files in the project. You should create a file share on the server, or install Front Page Extensions so that the Web site can be created directly by Visual Studio .NET 2003.

---

**Note:** For more information about using the **Copy Project** command, see "Deployment Alternatives" on MSDN.

---

#### File Copy

A simple file copy can be used to deploy some .NET-based applications. However, this type of distribution is only viable when there is no need to modify the registry or perform other tasks, such as stopping IIS when changes have occurred.

XML Web services and ASP.NET Web applications lend themselves very well to copy-type deployment. Previously, these types of applications were built using IIS and traditional COM components. Installing the components for these applications usually involved registering the component during the deployment process by using the Regsvr32.exe utility. Updating existing components was far more troublesome, and required stopping the IIS service, using the Regsvr32.exe utility to remove the registration of the old component, copying and registering the new component (again using Regsvr32.exe), and finally starting the IIS service again. These steps are no longer needed for XML Web services and ASP.NET Web applications. IIS does not place exclusive file locks on .NET assemblies and can detect changes to your solution files, automatically using the new versions on the fly. Also, because assemblies are self-describing, they don't have to be registered.

Using a simple copy mechanism to deploy your applications suffers from some drawbacks. These include:

- You require an additional step to install shared assemblies into the global assembly cache. You can install an assembly into the global assembly cache by using the Gacutil.exe utility with the **/ir** switch, which installs assemblies into the global assembly cache with a traced reference. These references can be removed when the assembly is uninstalled by using the **/ur** switch.
- You cannot automatically configure IIS directory settings for your Web application. You will have to define these manually or run a script to perform this configuration.
- You cannot automatically manage the rollback or uninstall of your application.
- You cannot include launch conditions, such as those available in Windows Installer files.
- You cannot automatically manage conditional deployment scenarios.
- You cannot automatically manipulate registry settings and file associations—you will manually have to edit the registry on your target computer or import .reg files to ensure appropriate settings are in place.
- You cannot take advantage of Windows Installer features, such as automatic repair (although this is usually not a major issue for Web applications).

---

**Note:** You can also use drag-and-drop in Windows Explorer to move shared assemblies into the Global Assembly Cache folder. However you should avoid this method as it does not implement any reference counting. Without reference counting, the uninstall routine of another application can result in the premature removal of a shared assembly from the global assembly cache, which would break your application that relies on that assembly.

---

**Note:** For more information on using File Copy to deploy .NET-based applications, see the MSDN article, "Determining When to Use Windows Installer Versus XCOPY." For further details see the More Information section at the end of this chapter.

---

If you are deploying more complex applications as a collection of build objects, you will generally have to supplement the file copy operations with scripts and utilities to complete the installation. For more details on this see Chapter 3, "Implementing the Deployment of .NET Framework-based Applications."

## Host Header Issues for Web Applications

Although for many scenarios you can consider packaging your Web applications in a Web setup Windows Installer file, there is one situation in which deploying a collection of build outputs is much more appropriate. This scenario is where you need to deploy your Web application to a Web server that distinguishes between sites based on host headers.

Host header names can be used to host multiple domain names from one IP address. To distinguish one Web site from another on the same computer, IIS uses the following three elements:

- TCP/IP address
- TCP port
- Host header name

As long as at least one of these three items is unique for each Web site, IIS can manage multiple sites. When IIS receives a request for a Web page, it looks at the information sent in by the browser. If the browser is HTTP 1.1 compliant (for example, Internet Explorer 3.x and later, or Netscape Navigator 3.x and later), the HTTP header contains the actual domain name requested. IIS uses this to determine which site should answer the request. Although Web setup projects allow you to specify which port to install the application on, they do not currently support deployment of a Web application to a site differentiated by host headers. Therefore, you need to consider alternatives to using Web setup projects to package your Web applications if you need to deploy in this scenario.

One approach that you might consider is to copy the build outputs of your Web application to the appropriate location on the target Web server, and then build a separate **standard** Windows Installer file for deploying the components that have more complex setup requirements. The person running the installer needs to ensure that the components are installed into the correct folders.

# Summary

There are many planning considerations to think about before you determine how to deploy your .NET-based application. Use the information contained in this chapter, and the following chapter as the basis for making an informed decision on deploying your .NET-based applications.

# More Information

For more information on preventing configuration file settings from overriding those in a parent directory:
Locking Configuration Settings
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconlockingconfigurationsettings.asp*

For more information on configuring Internet Explorer Applications:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconconfiguringieapplications.asp*

For information about the remoting elements you can use in your configuration file:
Remote Object Configuration
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconremoteobjectschannels.asp*

For more information on specifying remoting information directly in client and server code:
Programmatic Configuration
(*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconprogrammaticconfiguration.asp*).

To obtain the Visual J# .NET Redistributable Package:
*http://msdn.microsoft.com/vjsharp*

For more information on specifying permissions that the assembly requires to run:
Assembly Security Considerations
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconassembliessecurityconsiderations.asp*

For more information on the security implications of using the AllowPartiallyTrustedCallersAttribute Class
See the .NET Framework Class Library on AllowPartiallyTrustedCallersAttribute
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemsecurityallowpartiallytrustedcallersattributeclasstopic.asp*

For more information on delaying the signing of assemblies:
Delay Signing an Assembly
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpcondelayedsigningassembly.asp*

For more information about strong naming assemblies with attributes:
Signing an Assembly with a Strong Name
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconassigningassemblystrongname.asp*

For more information about specifying assembly locations:
Specifying an Assembly's Location
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconspecifyingassemblyslocation.asp*

For more information about how the common language runtime (CLR) locates assemblies:
How the Runtime Locates Assemblies
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconhowruntimelocatesassemblies.asp*

For more information about redirecting assembly binding:
Redirecting Assembly Versions
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconassemblyversionredirection.asp*

For more information on using Windows Forms User Controls in Web Applications:
Host Secure, Lightweight Client-Side Controls in Microsoft Internet Explorer
*http://msdn.microsoft.com/msdnmag/issues/02/01/UserCtrl/UserCtrl.asp*

For more information about registering WMI schemas for your instrumented applications with the ManagementInstaller class:
Registering the Schema for an Instrumented Application
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconregisteringtheschemaforaninstrumentedapplication.asp*

For information on COM wrappers
.NET Framework Developers Guide
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcomwrappers.asp*

For more information about Tlbimp.exe:
Type Library Importer (Tlbimp.exe)
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrftypelibraryimportertlbimpexe.asp*

For more information about proxies:
Deploying Application Proxies
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cossdk/htm/pgdeployingapplications_65gz.asp*

For information on the IIS 6.0 Application Pool Settings equivalent to the Process Model settings in ASP.NET:
Mapping ASP.NET Process Model Settings to IIS 6.0 Application Pool Settings
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconaspnetprocessmodelsettingequivalencetoapplicationpoolsettings.asp*

For more information on the architectural differences between IIS 5.0 and IIS 6.0:
Technical Overview of Internet Information Services (IIS) 6.0
*http://www.microsoft.com/windowsserver2003/techinfo/overview/iis.mspx*

For information about working with and registering HTTP handlers and HTTP modules:
HTTP Runtime Support
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconhttpruntimesupport.asp*

For more information about using SSL:
Untangling Web Security: Getting the Most from IIS Security
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dniis/html/websec.asp*

For more information about merge modules:
Introduction to Merge Modules
*http://msdn.microsoft.com/library/en-us/vsintro7/html*
*/vbconWhatYouNeedToKnowAboutMergeModules.asp*

Information about managing the security requirements for downloaded CAB files:
Writing Secure Managed Controls
*http://msdn.microsoft.com/library/en-us/cpguide/html*
*/cpconwritingsecuremanagedcontrols.asp*

For more information Satellite Assemblies:
Creating Satellite Assemblies
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html*
*/cpconcreatingsatelliteassemblies.asp*

For more information about symbol management, see the following:

- Bugslayer: Symbols and Crash Dumps
  *http://msdn.microsoft.com/msdnmag/issues/02/06/Bugslayer/Bugslayer0206.asp*
- How to Get Symbols
  *http://www.microsoft.com/ddk/debugging/symbols.asp*
- INFO: Use the Microsoft Symbol Server to Acquire Debug Symbol Files
  *http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q311503*
- INFO: PDB and DBG Files – What They Are and How They Work
  *http://support.microsoft.com/default.aspx?scid=kb;en-us;Q121366*
- Production Debugging for .NET Framework Applications
  *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html*
  */DBGrm.asp*

For more information about no-touch deployment of .NET-based applications:
Death of the Browser?
*http://msdn.microsoft.com/library/en-us/dnadvnet/html/vbnet10142001.asp*

and

Security for Downloaded Code
*http://msdn.microsoft.com/library/en-us/dnadvnet/html/vbnet12112001.asp*

For more instructions on using SMS to deploy applications:
SMS product documentation
*http://www.microsoft.com/smserver/techinfo/productdoc/default.asp*

For in-depth information about the Windows Installer automation interface:
See the "Automation Interface" section of the Platform SDK on MSDN
*http://msdn.microsoft.com/library/en-us/msi/auto_8uqt.asp*

For more details on using Application Center to replicate assemblies that are installed into the .NET global assembly cache:
Microsoft Knowledge Base Article 326950, "INFO: Application Center 2000 GAC Replication Support for Windows 2000"
*http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B326950*

For more information about using the Copy Project command:
Deployment Alternatives
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbconDeploymentAlternatives.asp*

For more information on using File Copy to deploy .NET-based applications:
Determining When to Use Windows Installer Versus XCOPY
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/XCOPYWinInst.asp*

# 3

# Implementing the Deployment of .NET Framework-based Applications

By now you should have learned about the different mechanisms for deploying .NET-based applications. In this chapter we go into more detail, showing you how to deploy your applications, either using no-touch deployment, Windows Installer deployment, or deploying a simple collection of build objects.

## No-Touch Deployment

As mentioned in Chapter 2, "Planning the Deployment of .NET Framework-based Applications", for Windows Forms-based applications, you may choose no-touch deployment as the most appropriate deployment mechanism.

The .NET Framework installation provides a mechanism to hook Internet Explorer 5.01 and later to listen for .NET assemblies that are being requested. During a request, the executable is downloaded to a location on disk called the assembly download cache. A process named IEExec then launches the application in an environment with constrained security settings.

To deploy an application using no-touch deployment, you need to simply recompile the application with the build output set to be the production Web server. Once the application is built, when a user browses to the location of the application, Internet Explorer will attempt to run the application. The application is now pulled down using HTTP and installed into the assembly download cache. Before the application is run, the security policy will be checked to ensure the application has permission to conduct its operations.

**Note:** The client attempts will only attempt to run the application if it has at the Framework installed and Internet Explorer v.5.01 or later installed. Otherwise it will provide you with the options of opening or saving the application.

If you are going to use this method to deploy your Windows Forms-based applications, you have to determine how your users will launch the application. One approach is to provide the user with a URL to the Internet location. In this case, your application and any assemblies that are required are downloaded and then run in a security sandbox—they are downloaded to the .NET Framework assembly cache download folder and executed from there with either Internet or intranet zone privileges (which are minimal). With this approach no install is necessary at all on the client computer, as all code is downloaded as needed. Your application will automatically update whenever changes occur on the Web server. If files have changed, the newer versions are downloaded as needed, just as with normal Web browsing.

**Note:** If a user launches the executable from a URL and then sets Internet Explorer to work offline, then the application is also offline—no checks will be made for updated application files on the Web server until Internet Explorer is set back to online. This can lead to problems if an assembly is required while the user is offline, but that assembly has not yet been downloaded because it was not accessed in a previous session.

An alternative to launching the application from a URL is to distribute an application stub to the users. This stub simply contains code that loads assemblies from a Web server using the **Assembly.LoadFrom()** method. In these circumstances, you can manage security for the downloaded assemblies to grant them more privileges than those usually allowable for URL-launched applications. While changes on the server will still result in the application being updated on the client, any changes to the stub will need to be redistributed to each client computer.

By default, an application stub will not load assemblies from the cache without first checking the versions on the Web server. Instead, it will fail with an exception. This means that you should ensure that you have a reliable connection to the Web server, or write code to provide an offline mode for the application.

The main disadvantage of the application stub approach is that there is still some code to distribute to the client. However, this will generally be much simpler than distributing a full application and often a simple file copy is appropriate.

**Note:** For more information on using no-touch deployment, see the links at the end of this chapter.

## Security Considerations

Users do not require administrative permission over their computers in order to install Windows Forms-based applications by no-touch deployment. However, it is possible to control the security of the applications when they run, using code access security. This model works by matching applications to the permissions they should be assigned. At runtime, the common language runtime gathers evidence on an assembly. Evidence could take the form of what Internet Explorer zone the code came from—local disk, intranet, Internet, trusted sites, untrusted sites—the URL the code originated from, any private key it is signed with, its hash value, an Authenticode publisher signature, and so on. Using this evidence, the CLR assigns the assembly to the appropriate code groups, or categories. Each code group has a permission set assigned to it, dictating which permissions the assembly should get, such as the ability to read from or write to the local disk, to access networked resources, to print, to access environment variables, and so on.

If necessary, you can change security permissions on the client computer to grant more permission to an existing code group or create new code groups from scratch. This is done using the CasPol tool or an MMC snap-in called the .NET Framework Configuration Tool. Each of these tools requires administrative permissions over the client machine to be useful.

One area where you may have to alter security settings to support no-touch deployment is if you have configuration files that need to be deployed with your application. In these circumstances you need to enable anonymous access for the directory that contains the config file. In many cases anonymous access will already be enabled, but if it is not, you should validate that this does not infringe on your overall security policy.

Another important security consideration is that any assemblies downloaded to the client computer can only call back to the domain from where it was downloaded. So, for example, a downloaded assembly from servera.microsoft.com cannot call a Web service on serverb.internal.microsoft.com. This security precaution prevents, for example, people from distributing code that performs denial-of-service attacks on some other server.

For more information on code access security, see *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication* (Microsoft Press, ISBN: 0-7356-1890-0) or on MSDN.

# Installer Packages

As mentioned in Chapter 2, "Planning the Deployment of .NET Framework-based Applications," Windows Installer packages can be created in a variety of applications. For the purposes of this section, we use Visual Studio .NET 2003 to show how the Windows Installer packages are created.

**Note:** If you are going to create your packages using a different application, you should consult that documentation for further information.

Visual Studio .NET 2003 creates Windows Installer packages from deployment projects. There are four types of deployment project used to create different types of installer packages. The following table shows the different types of package and the projects that correspond to them, along with details of what they may contain:

**Table 3.1: Types of Deployment Project**

| Package | Deployment Project | Can Contain |
|---|---|---|
| Windows Applications and Services | Standard Setup Project | Project Output Groups<br>Files and Folders<br>Assemblies<br>Application Resources<br>Merge Modules<br>CAB Files<br>Dependencies<br>Registry Settings<br>Project Properties<br>Custom Actions<br>User Interface Design Settings |
| Web Applications and Services | Web Setup Project | Project Output Groups<br>Files and Folders<br>Assemblies<br>Application Resources<br>Merge Modules<br>CAB Files<br>Dependencies<br>Registry Settings<br>Project Properties<br>Custom Actions<br>User Interface Design Settings |
| Merge modules | Merge Module Project | Project Output Groups<br>Files and Folders<br>Registry Settings<br>Assemblies<br>Application Resources<br>Custom Actions<br>Settings |

| Package | Deployment Project | Can Contain |
|---|---|---|
| CAB files | CAB File Project | Project Output Groups<br>ActiveX controls<br>Controls activated by a Web service |

To create Windows Installer files, you need to add some or all of the elements listed in the table to the appropriate deployment project. In this section we discuss each of the steps to follow to create an installer package. We focus on setup projects (standard and Web) as these allow you to create Windows Installer packages for Windows-based and Web-based applications and services. However, we also cover the specific issues related to creating merge module projects and CAB file projects.

**Note:** You can create deployment projects in Visual Studio .NET 2003 by using the setup wizard. For more information on the setup wizard, see "Distributing the Visual J# .NET Redistributable Package" on MSDN.

## Adding Project Output Groups

The most important elements to add to your setup projects are one or more project output groups. These represent the application itself as built in Visual Studio .NET 2003. A project output group may include any or all of the following:

- Documentation Files (generated by C# projects)
- Project Output (the executable)
- Localized Resources
- Debug Symbols
- Content Files (for example html and aspx pages)
- Source Files

In some cases you can add all the information you need to a Windows Installer package simply by adding several project output groups. In others, a number of other steps are required, as detailed in the rest of this module.

► **To add a project output group to a deployment project**

1. Select a target folder in the **File System Editor**.
2. On the **Action** menu, point to **Add**, and then click **Project Output**.
3. In the resulting **Add Project Output Group** dialog box, select a project from the **Project** list.
4. Select the type of output from the outputs list. You can select multiple outputs from the list.
5. Optionally, select a different configuration from the **Configuration** list.
6. Click **OK** to apply the selection and close the window.

**Note:** The Project Output command is also available from the Project menu. Choosing this command from the Project menu rather than the Action menu causes the project outputs to be placed in the Application folder rather than in the target folder you have selected. This will also happen if you add the project from Solution Explorer rather than File System Editor.

The advantage of adding project output groups rather than the individual elements of a project is that much of the work has already been done for you. This means that it is easier to ensure that you add everything you need and Visual Studio .NET is already aware of any dependencies.

**Note:** In some cases you will not want to add all of the files in a project output group. For example, if you decide to add content files, it may not be appropriate to add all of those files to your production servers. You can add a filter to the project output group to control which files will actually be deployed. For more information see the article "Excluding Files from a Project Output Group" on MSDN. See More Information at the end of this chapter for further details.

## Adding Files

After adding project output groups to your deployment project, you may need to add additional files. Your setup project should include all the files you need to add during the setup process. This will include some or all of the following:

**Table 3.2: Files that May Be Added to a Setup Project**

| Potential Items in a Package | Standard Setup Project | Web Setup Project |
|---|---|---|
| Files and Folder | ✓ | ✓ |
| Executables | ✓ | ✓ |
| Libraries | ✓ | ✓ |
| Data Files | ✓ | ✓ |
| Services | ✓ | ✓ |
| COM/COM+ Components | ✓ | ✓ |
| Event Log Resources | ✓ | ✓ |
| Message Queue Resources | ✓ | ✓ |
| Performance Counters and Categories | ✓ | ✓ |
| Debug Symbols | ✓ | ✓ |
| Security Policies | ✓ | ✓ |
| .NET Configuration files | ✓ | ✓ |
| Web Pages | | ✓ |

| Potential Items in a Package | Standard Setup Project | Web Setup Project |
|---|---|---|
| Web Forms | | ✓ |
| Web Services | | ✓ |
| Discovery Files for Web Services | | ✓ |
| XML Schema Definition Files | | ✓ |

You can use File System Editor in Visual Studio .NET 2003 to add files to a deployment project, to specify the locations where files will be installed on a target computer, and to create shortcuts on a target computer.

► **To add files to a deployment project**

1. In **Solution Explorer**, select a deployment project.
2. On the **View** menu, point to **Editor**, and then click **File System**.
3. Select a target folder in the **File System Editor**.
4. On the **Action** menu, point to **Add**, then click **File…**
5. In the resulting **Add Files** dialog box, select a File, and click **Open**.
6. Repeat steps 3-5 as required.

You can add special folders and custom folders to deployment projects. Special folders correspond to predefined Windows folders. The physical location of these folders can vary from one computer to another: for example, the System folder may be located in C:\Windows on one computer, D:\Windows on another, and C:\Winnt on a third. Regardless of the physical location, Windows recognizes the folder as the System folder by reading special attributes. Using special folders in a deployment project allows you to choose a destination folder on a target computer without knowing the actual path to that folder.

Custom folders represent other folders on a target computer. Unlike special folders, custom folders do not necessarily depend on existing folders on the target computer, but rather allow the setup package to create new folders at install time.

► **To add special folders to a deployment project**

1. In **Solution Explorer**, select a deployment project.
2. On the **View** menu, point to **Editor**, and then click **File System**.
3. Select the **File System on Target Machine** computer icon, right-click and select **Add Special Folder**, and then select the folder you wish to include (such as the Global Assembly Cache).
4. Add files and additional folders as necessary.

**Note:** For more information on Special Folders and custom folders, see "Special Folders and Custom Folders" on MSDN

## Adding Configuration Files

Configuration files are deployed in the same way as any other files. However, they are of particular interest as they may contain information specific to the deployment of a .NET-based application.

As mentioned in Chapter 3, "Planning the deployment of .NET Framework-based Applications," one of the major challenges of configuration file deployment is that they may vary during the different stages of the application lifecycle. You need to ensure that the right configuration files are used at each stage.

A developer can use Visual Studio.NET to manage this process by specifying a configuration override file in the application project. The override file allows you to specify which configuration file should be built into a Windows Installer file when you build the setup project for your application. Exactly how you specify the file will vary depending on whether the application is developed in Visual Basic or C#.

### Configuration Override File in Visual Basic Projects

To define the configuration override file option in Visual Basic projects, you should specify the appropriate configuration file on the **Deployment** page of the **Configuration Properties** section in the **Property Pages** dialog box. For example, if you are about to build a Windows Installer file for deploying your application to the production environment, you might specify *Production.app.config* or *Production.web.config* as the override file for your application's project.

When you come to build the deployment project, Visual Studio .NET 2003 packages the configuration override file with your other build outputs and renames it to the required name for you in the Windows Installer file. For example, if you are packaging an ASP.NET application, Visual Studio names the configuration override file *web.config*. When you run the installer, the configuration file is deployed along with your application.

### Configuration File Override in C# Projects

The override file setting is not available in the property pages of C# applications, but you can still use configuration override files for your C# solutions. You need to edit the .csproj file to achieve this. The following example shows how to specify a configuration override file in the .csproj file for a C# Web application.

```
…
…
<Config
  Name = "Release"
  AllowUnsafeBlocks = "false"
  BaseAddress = "285212672"
  CheckForOverflowUnderflow = "false"
  ConfigurationOverrideFile = "production.web.config"
  DefineConstants = "TRACE"
  DocumentationFile = ""
```

```
  DebugSymbols = "false"
  FileAlignment = "4096"
  IncrementalBuild = "false"
  Optimize = "true"
  OutputPath = "bin\"
  RegisterForComInterop = "false"
  RemoveIntegerChecks = "false"
  TreatWarningsAsErrors = "false"
  WarningLevel = "4"
/>
…
…
```

**Note:** In this example, the **ConfigurationOverrideFile** entry is set for the Release section—it is possible to modify the .proj file to specify different override files for different build configurations.

## Adding Assemblies

The different elements of an assembly can reside in separate files, or, they can be grouped together in a single PE file, such as a .dll or an .exe file. For example, you might want large graphics or seldom used types to reside in separate files—that way they will only be loaded when required by the application.

**Note:** Instrumented assemblies need to be added to your project as custom actions and so are covered in that section.

As mentioned in Chapter 2, assemblies that are used by a single application should be deployed as a private assembly:

► **To add a private assembly**

1. Select a target folder in the **File System Editor**.
2. On the **Action** menu, point to **Add**, and then click **Assembly**.
3. In the resulting **Component Selector** dialog box, browse to the assembly you want to add, click **Select** for each assembly you wish to apply, and click **Close** to apply the selections and close the window.

**Note:** The Component command is also available from the Project menu. Choosing this command from the Project menu rather than the Action menu causes the assembly to be placed in the Application folder rather than in the target folder you have selected.

Assemblies that will be used by multiple applications or used by a distributed application will often be installed in the global assembly cache. However, before you can install assemblies in the global assembly cache, you first need to ensure that the file system includes a reference to the global assembly cache folder itself. (By default it only includes references to three folders on the target machine: the Application folder, the user's desktop, and the user's Programs menu.)

► **To add a reference to the global assembly cache**

1. Right-click on **File System** on **Target Machine**.
2. Click **Add Special Folder**.
3. Select **Global Assembly Cache Folder**.

Once the reference has been added, you are now in a position to add any strong-named assemblies to the global assembly cache.

► **To create a shared assembly**

1. In the **Files Editor**, right-click **Add**, and select the **Global Assembly Cache Folder**.
2. Select the **Global Assembly Cache Folder**, right-click and point to **Add…**, Click either **Assembly** or **Project Output Group** (that contains the assemblies you want added to the global assembly cache).
3. Select the **Assembly** or **Project Output Group** and click **OK**. For each choice you will have another dialog box appear:

   For **Assemblies**:

   a. In the resulting **Component Selector** dialog box, browse to the assembly you want to add, and click **Select** for each assembly you wish to apply.
   b. Click **OK** to apply the selections and close the window.

   For **Project Output Groups**:

   a. In the resulting **Add Project Output Group** dialog box, select a project from the **Project** list.
   b. Select the type of output from the **Outputs** list. You can select multiple outputs from the list.
   c. Optionally, select a different configuration from the **Configuration** list.
   d. Click **OK** to apply the selection and close the window. You will now see the assemblies in the **Global Assembly Cache** folder.

---

**Note:** For more information on when to use the global assembly cache or for strong-naming assemblies, see Chapter 2, "Planning the Deployment of .NET Framework-based Applications."

---

## Adding COM Components

Any COM components required by your .NET-based application should either already exist on the target computers or be deployed along with your application. If you are using Visual Studio .NET setup and deployment projects, the COM dependencies of your .NET assemblies are detected and included in the detected dependencies list. However, any other dependencies of the COM object cannot be automatically detected and must be manually included into the setup project for deployment.

► **To set a COM component to register automatically on install**

1. Select the COM component, right-click and click **Properties**.
2. Under the **Register** property select one of the five options:
   a. **vdsrpDoNotRegister**: The item requires no registration.
   b. **vsdrpCOM**: The item will be registered as a COM object.
   c. **vsdrpCOMRelativePath**: The item will be registered as an isolated COM object.
   d. **vsdrpCOMSelfReg**: The item will be self-registered as a COM object when it is installed. Not available for assemblies.
   e. **vsdrpFont**: The file will be registered as a font file when it is installed.

Along with your COM+ components, you also need to deploy the interop assembly that allows it to communicate with your .NET-based applications. You can either place the interop assembly in the global assembly cache or deploy it as a private assembly. As COM components have computer-wide visibility, you should normally install interop assemblies in the global assembly cache so that the interop assembly for the component is visible to all .NET applications on that computer.

**When you** deploy a primary interop assembly to development computers, you should always register the primary interop assembly with Regasm.exe on the development computer. This places entries into the registry that associate the primary interop assembly with the COM object. In addition to using Regasm.exe, you must then place the primary interop assembly in the global assembly cache on the development computer. When a developer then adds a reference to the COM component in Visual Studio .NET 2003, the fact that there is a primary interop assembly associated with this COM component is detected and Visual Studio .NET 2003 does not generate an alternate interop assembly. At run time, the primary interop assembly is found in the global assembly cache and loads properly on the development computer.

For more information about deploying all required COM components, see the following articles on MSDN:

- "Deploying COM+ Applications"
- "Application Deployment Using Microsoft Management Technologies"

## Adding Security Policy

Security policy is managed by administrators at three different levels: enterprise, computer, and user. In addition, developers can specify security policy in code for their application domains.

Security policy can be easily distributed and applied using Windows Installer files. The .NET Framework configuration tool (Mscorcfg.msc) provides a wizard for creating Windows Installer files for your security policy. The wizard creates a Windows Installer file that corresponds to one of the three configurable policy levels

(enterprise, computer, and user), but not all of them concurrently. If you are deploying security policy for all three configurable levels, you must create three different Windows Installer files and deploy them individually.

The wizard creates the Windows Installer file using the current policy settings of the computer where the wizard executes. For example, to create a user policy for deployment to a group of users, you configure the user policy on your current computer, create the Windows Installer file with the wizard, and then return the user policy of the current computer to its original state.

You need to ensure that the user account under which the policy is installed has adequate privileges to access the configuration files you are modifying. For example, if you are currently logged on using an account that does not have permission to modify the enterprise configuration file, and the Windows Installer file needs to modify that file, the installation does not succeed. Note that the Windows Installer package does not produce an error if the current account does not have sufficient permission to modify the configuration file.

If you need to deploy security policy with your application, you can create a nested setup whereby the security policy installer is launched from inside the application installer.

► **To create the policy installer package**

1. Open the **Microsoft .NET Framework 1.1 Configuration** tool.
2. Expand the **Runtime Security Policy** folder.
3. Make sure the policies on this machine are the same as what should be published by the installer package.
4. Select the **Runtime Security Policy** folder again.
5. Under **Tasks,** click **Create Deployment Package**.
6. Click your policy level (**Enterprise**, **Computer** or **User**) and indicate the location and filename of the installer package the wizard is about to create.
7. Click **Next**.
8. Click **Finish**.

## Managing Merge Modules

The main advantage of using merge modules is that you only need to define the setup logic for the assemblies in the .msm file once, rather than for each instance of the application. The .msm file needs only to be added to the Windows Installer file once, even though multiple applications in the installer file depend on their components. If you need to upgrade the assemblies in the .msm file, you need to only create a new .msm file and include that in your Windows Installer file for redeployment.

**Note:** For more information about updating and redeploying components packaged in .msm files, see Chapter 4, "Maintaining .NET Framework-based Applications."

Here are some recommendations for using merge modules:

- For components that are shared across multiple applications (or have the potential to be shared) you should package each component in its own .msm file. This allows you to deploy that .msm file with the installer package of the application that uses the component without needing to repackage the component every time. While it is possible to put multiple components into a single merge module, it is best to create a separate merge module for each component to avoid distributing unnecessary files.
- Installer packages can include multiple applications, allowing you to install a suite of applications in a single step. In this case, the installer package should include merge modules for all components used by any of the included applications. If a merge module is used by more than one application it needs to be added to the installer package only once.
- You should capture all of the dependencies for a particular component. After you distribute a merge module, it should never be modified. Instead, you should create a new merge module for each successive version of your component in your merge module to ensure that the component is installed correctly.
- Each merge module contains unique version information that is used by the Windows Installer database to determine which applications use the component, preventing premature removal of a component. For this reason, a new merge module should be created for every incremental version of your component. A merge module should never be updated after it is included in an installer package.
- You should generally use merge modules for applications developed in-house, not for software that you ship externally, as you cannot prevent it becoming part of any other application. For most code that you are shipping externally, you should ship as an .msi instead.

In some cases you will be provided with existing merge module projects that you need to add to your setup project.

► **To add an existing merge module project to a solution**

1. On the **File** menu, point to **Add**, and then click **Existing Project**.
2. In the resulting **Add Existing Project** dialog box, browse to the location of the merge module project and click **Open**.

In other cases the work will not already have been done for you and you will need to create a new merge module.

► **To create a new merge module**

1. On the **File** menu, point to **Add**, and then click **New Project**.
2. In the resulting **Add New Project** dialog box, select the **Setup and Deployment Projects** folder.
3. Click **Merge Module Project**.
4. Add the files and components you require.
5. Click **OK**.

► **To add an existing merge module (from a third party developer) to a deployment project**

1. Right-click the deployment project, point to **Add**.
2. Click **Merge Module…**
3. Select the merge module (*.msm), and click **Open**.

---

**Note:** Deployment projects for applications that reference the .NET Framework automatically add a merge module (dotnetfxredist_x86_*xxx*.msm, where *xxx* represents the language code) in the detected dependencies folder. This merge module cannot be redistributed; it exists for internal use by the project to prevent individual .NET assemblies from being listed. It is excluded by default; attempting to include it will cause a build error.

---

By default, files in your merge modules are installed into the folder locations you specify when you build the .msm file. In some cases, you may want to allow the developer who uses your merge module some flexibility in deciding where files should be installed for their application. For example, if an assembly in a merge module is used by multiple applications, the developer may want to install it in the global assembly cache; otherwise, they would install it in the application directory.

To allow other developers to retarget your files to a different location, you should place them in the Module Retargetable folder in your merge module project. When the resulting merge module is added to another deployment project, the author of that project can choose a location for your files by setting the **Module Retargetable Folder** property exposed by your built merge module in their installer package.

You can provide a default location for files in the Module Retargetable folder by setting its **DefaultLocation** property. You can set this to any of the following:

- [CommonFilesFolder]
- [FontsFolder]
- [GAC]
- [ProgramFilesFolder]
- [SystemFolder]
- [WindowsFolder]
- [TargetDir]

The **[TargetDir]** setting corresponds to the application folder for the solution that is deployed with the installer package that includes your merge module. The other folders in the list should be self-explanatory.

### Setting Merge Module Project Properties

In addition to the properties that merge module have in common with installer packages, they also support a **ModuleSignature** property. This property specifies a unique identifier for the merge module. The **ModuleSignature** property contains the name of the merge module followed by a globally unique identifier (GUID). Each released version of a merge module must have a unique **ModuleSignature** property in order to avoid versioning problems. You should never manually edit the GUID portion of this property—you should use the GUID generation facilities in the **Module Signature** dialog box.

### Building a Merge Module

As with project properties, merge module projects support a subset of the build options available for installer packages. These include:

- Output file name
- Compression
- Authenticode signature

For more information about these options, see the Build Your Windows Installer File section later in this chapter.

For step-by-step instructions on how to create and use merge modules, see "Creating or Adding a Merge Module Project" on MSDN.

## Managing CAB Files

In some cases you will be provided with existing CAB projects that you need to add to your setup project.

► **To add an existing cab project to a solution**

1. On the File menu, point to **Add**, and then click **Existing Project**.
2. In the resulting **Add Existing Project** dialog box, browse to the location of the CAB project, and click **Open**.

In other cases you may wish to group existing files and components into a new CAB project.

► **To create a new CAB project**

1. On the **File** menu, point to **Add**, and then click **New Project**.
2. In the resulting **Add New Project** dialog box, select the **Setup and Deployment Projects** folder.

3. Click **Cab Project**.
4. Add the files and components you require.
5. Click **OK**.

To add a CAB file to a deployment project, you can import the *.cab file as either a single file or as the project output of a CAB project. (See earlier steps for how to add files or project output groups.)

Properties of CAB projects allow you to specify a level of compression and implement Authenticode signing, which allows administrators to grant higher trust to these controls without reducing their security with regard to other intranet/Internet code. You can also set the following properties for your CAB file project:

- **FriendlyName**. Specifies the public name for the CAB file.
- **Version**. Specifies the version number of the CAB file. As with the other types of setup projects, the **Version** property should be changed for each released version of your CAB file.
- **WebDependencies**. Specifies the URL, friendly name, and version of other CAB files that must be installed in order for the selected CAB project to be successfully installed. If this property is set, all dependencies are automatically downloaded and installed when the CAB file is run. You can specify multiple Web dependencies using the **Web Dependencies** dialog box, which is accessible from the **Properties** window.

---

**Note:** For more information about managing the security requirements for downloaded CAB files, see "Writing Secure Managed Controls" on MSDN.

---

Unlike the other deployment project types, there are no editors provided for working with CAB projects. Files and project outputs can be added to a CAB project in Solution Explorer, and properties can be set in the **Properties** window or in the project property pages.

## Adding Dependencies

Deployment projects in Visual Studio .NET automatically detect dependencies and add them whenever a project output group, assembly, or merge module is added to the project. For assemblies, all dependencies on other assemblies are detected. However, if the assembly references an unmanaged component (for example, a COM .dll), any dependencies of the unmanaged component will not be detected. Likewise, COM components added directly to a deployment project may have dependencies that are not detected.

**Note:** Rather than adding assemblies directly to a deployment project, it is best to add a project output group that contains the assembly. The deployment tools can more reliably detect dependencies for a project output group.

You need to determine all of the possible dependencies for your COM components and include those files in the deployment project. You should check the documentation for the component or contact the component's author to get a list of dependencies. Alternatively, you can obtain a merge module (an .msm file) for the COM component to deploy the component and its dependencies—the developer who created the .msm file should already have defined and included the dependencies for their component and included them in the .msm file.

## Adding Launch Conditions

One of the more powerful features of Windows Installer is the ability to set conditions for installation, allowing you to customize installations on a case-by-case basis. The deployment tools in Visual Studio .NET support conditional deployment through launch conditions. Using the Launch Conditions Editor you set the **Condition** property, which is then compared against the environment on the target computer. For example, you might want to install different files based on operating system version, customize registry settings based on the value of an existing key, or even halt installation if a dependent application is not already installed on the target computer. Launch conditions can be set to check for the operating system version, existence of files, registry values, Windows Installer components, the CLR, and IIS. Launch conditions are specified in the Launch Conditions Editor.

► **To add a launch condition**

1. On the **View** menu, point to **Editor**, and then click **Launch Conditions** when a deployment project is selected in **Solution Explorer**.
2. Right-click on **Requirements on Target Machine** and select one of the following choices. For each choice, two entries are created: One entry in the **Search Target Machine** folder and one in the **Launch Conditions** folder. The entry in the **Search** folder specifies where the package should look for a needed entry and the **Condition** folder specifies what conditions should be found to make the condition true.
   a. **Add File Launch Condition**. This launch condition allows you to search for a particular file as a condition. In the properties of the search you can input a specific file, including location, date, size and version, or search for a range of values. You can also specify the depth of the search, which determines how many levels of subfolders should be included in the search.
   b. **Add Registry Launch Condition**. This launch condition will allows you to search for a particular registry entry as a condition. In the properties of the search you can input a specific registry location and key as your value.

c. **Add Windows Installer Launch Condition**. This creates a pre-built search function that will check to make sure that the target machine has Windows Installer 2.0 or later. The GUID of the installer package needs to be included in the properties of the search function.

d. **Add .NET Framework Launch Condition**. This creates a pre-built search function that will check to make sure that the target machine has a version of the runtime supported by the application. Which versions are supported is determined by the **<SupportedRuntimes>** property in the application configuration file. Nothing needs to be done to this entry since the information is automatically populated.

e. **Add Internet Information Services Condition**. This creates a pre-built search function that will check to make sure that the target machine has IIS 4.0 or greater. Since ASP.NET applications work on IIS 5.0 or later you should modify the **Condition** property from **REGISTRYVALUE >= "#4"** to **REGISTRYVALUE >= "#5"**. This ensures that only IIS 5.0 or later is supported.

3. In the **Launch Conditions** folder under the **InstallUrl** property, you can specify the location where the user can download files if a launch condition evaluates to **false**. If this property contains a value, the user is presented with a dialog box containing **Yes** and **No** buttons, plus the message specified in the **Message** property. If the user selects **Yes**, they will be redirected to the location specified in the property; in either case the installation will be terminated immediately.
4. In the **Launch Conditions** folder under the **Message** property, you can specify the error message that will be displayed if the launch condition is not met. For the automatically populated conditions (IIS and .NET Framework) a message is provided.

For more information about working with launch conditions, see "Launch Condition Management in Deployment" on MSDN.

For more information about using the **Condition** property, see "Deployment Conditions" on MSDN.

One commonly used launch condition is for applications that use J#. As mentioned in Chapter 2, "Planning the Deployment of .NET Framework-based Applications," for a J# application to run, you have to deploy the J# redistributable package. For more information on adding a launch condition for the J# redistributable package, see "Distributing the Visual J# .NET Redistributable Package" on MSDN.

Another common launch condition is for applications that require MDAC. As mentioned in Chapter 1, "Introduction", MDAC is prerequisite for many Framework applications. For more information on adding an MDAC launch condition, see "Adding a Launch Condition for Microsoft Data Access Components" on MSDN.

## Adding Registry Settings

Registry keys can be added to a deployment project using the Registry Editor in a Visual Studio .NET 2003 deployment project.

► **To add a registry entry to the deployment project**

1. In **Solution Explorer**, select a setup project.
2. On the **View** menu, point to **Editor**, and then click **Registry**.
3. Under the list select the path (HKLM for example) and create a new registry key to be added.

If a key doesn't exist in the registry of a target computer, it will be added during installation. Keys can be added beneath any top-level key in the Registry Editor.

---

**Note:** For instructions on how to add registry keys to your projects, see "Adding and Deleting Registry Keys in the Registry Editor" on MSDN.

---

The Registry Editor can be used to specify values for new or existing registry keys in the registry of a target computer. You can add string, binary, and DWORD values. The values are written to the registry during installation. The values you specify overwrite any existing values.

---

**Note:** By modifying the registry table directly in the Orca database editor tool you can also add REG_EXPAND_SZ and REG_MULTI_SZ values to the registry. For more information on this see "Registry Table" in MSDN.

---

Registry keys and values can be added to a deployment project by importing a registry file (.reg) into the Registry Editor. This allows you to save time by copying an entire section of an existing registry in a single step. Registry files can be created using tools such as the Registry Editor (Regedit.exe) included with Windows 2000.

You can also use the Registry Editor to specify a default value for any registry key. For more information on this see "Creating a Default Registry Value in the Registry Editor" on MSDN.

---

**Note:** As with managing files and folders, if a user is running the installer package directly, the user must have the appropriate permissions to modify the registry; otherwise, installation fails when the Windows Installer attempts to create, modify, or delete registry keys and values. For example, the user must have local administrative privileges if your installer creates, deletes, or modifies keys and values in the **HKEY_LOCAL_MACHINE** hive.

---

### COM Calling to .NET

When deploying .NET assemblies that will be used by COM, you must make the .NET assemblies visible to COM by registering the .NET assembly. This is done by creating an entry in the registry that points to Mscoree.dll, which then loads and executes the .NET assembly.

After the .NET assembly is registered, it must be located where it can be found by the .NET runtime. This means it must be placed in one of the following locations:

- The global assembly cache
- The application base directory, which is usually the .exe directory or the bin directory in a Web application

## Adding File Associations

If your application includes (or creates) files as part of its normal operation, then you might want those files to be associated with your application. For example, you might want any file that your application creates to open with your executable file if the user attempts to open the file. The File Types Editor is used to establish file associations on the target computer, by associating file extensions with your application and specifying the actions allowed for each file type.

► **To add a file association**

1. In **Solution Explorer**, select a deployment project.
2. On the **View** menu, point to **Editor**, and then click **File Types**.
3. Set the file association with the primary output, and click **OK**.

---

**Note:** For more information about editing file associations, see "File Types Management in Deployment" on MSDN.

---

## Setting Project Properties

To have your installer package contain all of the information needed to install correctly, you should manipulate properties that control how your Windows Installer files interact with Windows Installer. While this can be done at any stage of the package creation process, the more property information you can discover and include beforehand, the easier the creation process will be.

Deployment projects have two categories of properties: general project properties and configuration-dependent properties. General properties apply to all project configurations, whereas configuration-dependent properties apply to a specific project configuration.

► **To set general project properties**

1. In **Solution Explorer**, select a deployment project.
2. Right-click and select **Properties**.
3. In the **Solution Property Pages** dialog box, select the **View** menu, and click **Property Pages**.

► **To set configuration-dependent project properties**

1. On the **View** menu, choose **Property Pages**.
2. Select a configuration from the **Configuration** list.
3. Select a category from the list of categories. The properties for the selected category will be displayed.

> **Note:** If most of your deployment properties are common across all configurations, you can avoid duplicate effort by choosing **All Configurations**. Once the common properties are set, choose another configuration and set any unique properties.)

These are the Windows Installer file properties you can manipulate with Visual Studio .NET 2003:

- **AddRemoveProgramsIcon**. Specifies an icon to be displayed in the **Add/Remove Programs** dialog box on the target computer. This property has no effect when installing on operating systems such as Windows 98 or Windows NT, where icons are not displayed in the **Add/Remove Programs** dialog box.
- **Author**. Specifies the name of the author of an application or component. The **Author** property is displayed on the **Summary** page of the **Properties** dialog box when an installer file is selected in Windows Explorer. After the application is installed, the property is also displayed in the **Contact** field of the **Support Info** dialog box, which is accessible from the **Add/Remove Programs** dialog box.
- **Description**. Specifies a free-form description for an installer file. The **Description** property is displayed on the **Summary** page of the **Properties** dialog box when an installer file is selected in Windows Explorer. After the application is installed, the property is also displayed in the **Support Info** dialog box, accessible from the **Add/Remove Programs** dialog box.
- **DetectNewerInstalledVersion**. Specifies whether to check for later versions of an application during installation. If this property is set to **True** and a later version is detected at installation time, installation ends. For more information about working with this property, see Chapter 4, "Maintaining .NET Framework-based Applications."
- **Keywords**. Specifies keywords used to search for an installer. The **Keywords** property is displayed on the **Summary** page of the **Properties** dialog box when an installer file is selected in the Windows Explorer.
- **Localization**. Specifies the locale for the run-time user interface.

- **Manufacturer**. Specifies the name of the manufacturer of an application or component. The **Manufacturer** property is displayed in the **Publisher** field of the **Support Info** dialog box, accessible from the **Add/Remove Programs** dialog box. It is also used as a part of the default installation path (C:\Program File\Manufacturer\Product Name) displayed during installation.
- **ManufacturerUrl**. Specifies a URL for a Web site containing information about the manufacturer of an application or component. The **ManufacturerUrl** property is displayed in the **Support Info** dialog box, accessible from the **Add/Remove Programs** dialog box.
- **ProductCode**. Specifies a unique identifier for an application. This identifier must vary for different versions and languages. Windows Installer uses the **ProductCode** property to identify an application during subsequent installations or upgrades; no two applications can have the same **ProductCode** property. To ensure a unique **ProductCode** property, you should never manually edit the GUID; instead, you should use the GUID generation facilities in the **Product Code** dialog box. For more information about working with this property, see Chapter 4, "Maintaining .NET Framework-based Applications."
- **ProductName**. Specifies a name that describes an application or component. The **ProductName** property is displayed as the description of the application or component in the **Add/Remove Programs** dialog box. It is also used as a part of the default installation path (C:\Program Files\Manufacturer\Product Name) displayed during installation.
- **RemovePreviousVersions**. Specifies whether an installer removes earlier versions of an application during installation. If this property is set to **True** and an earlier version is detected at installation time, the earlier version's uninstall function is called. The installer checks **UpgradeCode**, **PackageCode**, and **ProductCode** properties to determine whether the earlier version should be removed. The **UpgradeCode** property must be the same for both versions, whereas the **PackageCode** and **ProductCode** properties must be different. For more information about working with this property, see Chapter 4, "Maintaining .NET Framework-based Applications."
- **RestartWWWService**. For Web setup projects only, this specifies whether Internet Information Services stop and restart during installation. Restarting may be required when deploying an application that replaces ISAPI DLLs or Active Template Library (ATL) Server components that are loaded into memory. ASP.NET components do not require a restart when replacing components that are loaded into memory. If this property is not set to **True** and a restart is required, the installation completes only after the World Wide Web Publishing service is restarted.
- **SearchPath**. Specifies the path that is used to search for assemblies, files, or merge modules on the development computer.

- **Subject**. Specifies additional information describing an application or component. The **Subject** property is displayed on the **Summary** page of the **Properties** dialog box when an installer file is selected in the Windows Explorer.
- **SupportPhone**. Specifies a phone number for support information for an application or component. The **SupportPhone** property is displayed in the **Support Information** field of the **Support Info** dialog box, accessible from the **Add/Remove Programs** dialog box.
- **SupportUrl**. Specifies a URL for a Web site containing support information for an application or component. The **SupportUrl** property is displayed in the **Support Information** field of the **Support Info** dialog box, accessible from the **Add/Remove Programs** dialog box.
- **Title**. Specifies the title of an installer. The **Title** property is displayed on the **Summary** page of the **Properties** dialog box when an installer file is selected in the Windows Explorer.
- **UpgradeCode**. Specifies a shared identifier that represents multiple versions of an application. This property is used by Windows Installer to check for installed versions of the application during installation. The **UpgradeCode** property should be set for only the first version; it should never be changed for subsequent versions of the application, nor should it be changed for different language versions. Changing this property prevents the **DetectNewerInstalledVersion** and **RemovePreviousVersions** properties from working properly. For more information about working with this property, see Chapter 4, "Maintaining .NET Framework-based Applications."
- **Version**. Specifies the version number of an installer. The **Version** property should be changed for each released version of your installer. For more information about versioning, see Chapter 4, "Maintaining .NET Framework-based Applications."

### Create Localized Installers

The Visual Studio deployment tools include several features that allow you to distribute different versions of your application for different locales. You need to create a separate installer for each localized version of your application.

To create a localized installer file, you set the **Localization** property of the deployment project to one of the supported languages (listed in the drop-down list in the Properties window). The **Localization** property setting determines the language for the default text displayed in the installation user interface dialog boxes, such as button captions and instructions for proceeding with the installation. You cannot see the translated text in the Visual Studio .NET 2003 IDE; you can see only the translated text if you build and run the installer.

► **To set the localization property**

1. Open the **Solution Explorer** and select the deployment project.
2. Right-click and select **Properties**.
3. In the **Solution Property Pages** dialog box, select the **View** menu and click **Property Pages**.
4. In the **Localization** property, select the language that this project supports.

Text that is provided by properties is not translated. For example, the **ProductName** property that determines the name displayed in the title bar of the installation dialog boxes is not translated into the chosen locale, so you need to enter the localized **ProductName** in the Properties window for each localized deployment project. Other deployment project properties that you may need to localize include:

- Author
- Description
- Keywords
- Manufacturer
- ManufacturerUrl
- Subject
- SupportPhone
- SupportUrl
- Title
- AddRemoveProgramsIcon (if the icon contains text)

Additional properties that may need to be localized include the **Name** and **Description** properties for shortcuts in the File System Editor, the **Name** and **Description** properties for file types and actions in the File Types Editor, and the **Message** property for conditions in the Launch Conditions Editor.

---

**Note:** If the core files for your application are the same for all locales, you should consider putting the core files in a merge module and adding the merge module plus any locale-specific files to the installer for each locale. Registry settings, custom actions, and file types can be set in the merge module project so that you do not need to recreate them for each project.

For an example of a localized Installer package, see "Localizing a Windows Installer Package" on MSDN.

---

## IIS Settings

You can use the project properties of a Web setup project to deploy IIS settings along with your solution. You specify these settings by using the Properties window in the Visual Studio .NET Web setup project, when the Web Application folder (or a sub-folder) is selected. The following describes properties that control the IIS virtual directory settings:

- **AllowDirectoryBrowsing**. Sets the **IIS Directory browsing** property for the selected folder. This setting corresponds to the **Directory browsing** check box on the **Directory** page of the **Internet Information Services Web Properties** dialog box, and can be set to either **True** or **False**.
- **AllowReadAccess**. Sets the **IIS Read** property for the selected folder. This setting corresponds to the **Read** check box on the **Directory** page of the **Internet Information Services Web Properties** dialog box and can be set to either **True** or **False**.
- **AllowScriptSourceAccess**. Sets the **IIS Script source access** property for the selected folder. This setting corresponds to the **Script source access** check box on the **Directory** page of the **Internet Information Services Web Properties** dialog box and can be set to either **True** or **False**.
- **AllowWriteAccess**. Sets the **IIS Write** property for the selected folder. This setting corresponds to the **Write** check box on the **Directory** page of the **Internet Information Services Web Properties** dialog box and can be set to either **True** or **False**.
- **ApplicationProtection**. Sets the **IIS Application Protection** property for the selected folder. This setting corresponds to the **Application Protection** selection on the **Directory** page of the **Internet Information Services Web Properties** dialog box. It can be set to:
  - **vsdapLow**. The application runs in the same process as IIS.
  - **vsdapMedium**. The application runs in an isolated pooled process in which other applications are also run.
  - **vsdapHigh**. The application runs in an isolated process separate from other processes.
- **AppMappings**. Sets the **IIS Application Mappings** property for the selected folder. This setting corresponds to the **Application Mappings** list on the **App Mappings** page of the **Internet Information Services Application Configuration** dialog box.
- **DefaultDocument**. Specifies the default (startup) document for the selected folder. This setting corresponds to the list of default documents on the **Documents** page of the **Internet Information Services Web Properties** dialog box. You can specify a comma-separated list to specify multiple default documents — the order of precedence for the default documents in taken from their relative positions in your comma-separated list.

- **ExecutePermissions**. Sets the **IIS Execute Permissions** property for the selected folder. This setting corresponds to the **Execute Permissions** list on the **Directory** page of the **Internet Information Services Web Properties** dialog box. You can set it to:
  - **vsdepNone**. Only static files, such as HTML or image files, can be accessed.
  - **vsdepScriptsOnly**. Only scripts, such as Active Server Pages scripts, can be run.
  - **vsdepScriptsAndExecutables**. All file types can be accessed or executed.
- **Ports**. Sets the port where a Web service is located on the target computer.
- **LogVisits**. Sets the **IIS Log Visits** property for the selected folder. This setting corresponds to the **Log visits** check box on the **Directory** page of the **Internet Information Services Web Properties** dialog box and can be set to either **True** or **False**.
- **Virtual Directory Name** The alias of the virtual directory. This virtual directory will be created if it doesn't already exist.

For more information about programmatically defining these IIS 6.0 Settings, see "Programmatic Administration Guide."

For more information about IIS Security in ASP.NET applications see "Authentication in ASP.NET: .NET Security Guidance" on MSDN.

## Adding Custom Actions

Custom actions allow you to perform actions at the end of the installation that cannot be handled by the standard process. The custom action will typically add files, components, or project output groups and you may add scripts to control the nature of the install. For example, you might want to create a local database on the target computer during installation. You could create an executable file that creates and configures the database, and then add that executable file as a custom action in your deployment project.

The Custom Actions Editor in Visual Studio .NET is used to manage custom actions in a deployment project. A deployment project can contain multiple custom actions, and you can control the order of those actions.

► **To add a custom action to a deployment project**

1. On the **View** menu, point to **Editor**, and click **Custom Actions**.
2. Select a folder in the **Custom Actions Editor**.
3. On the **Action** menu, click **Add Custom Action**.
4. In the **Select item in Project** dialog box, select a folder and select the .dll file, .exe file, or project output that contains the custom action.

5. If the item hasn't previously been added to the deployment project, click the **Add File**, **Add Output**, or **Component** button to add the item as a custom action. This also adds the item to your project.

**Note:** If you use the **Add File**, **Add Output**, or **Component** button to add an item to the **Select item in project** dialog box and subsequently cancel the dialog box, the items are still added to the deployment project. If you don't want the items in the deployment project, you can remove them using Solution Explorer.

By default custom actions are run after the actual installation is complete, and those custom actions do not have access to properties used to control installation. However, using the Orca tool, you can cause custom actions to occur during the installation process:

► **To have a custom action occur during the installation process**

1. Open the installer package using Orca.exe (provided in the Installer SDK).
2. Go to the **CustomAction** table and locate the name of the custom action that you want to occur during the installation (Located under the **Action** column).
3. Go to the **InstallExecuteSequence** table and locate the custom action in the **Action** column. In the corresponding **Sequence** column, change the value to a number higher than the event you want the custom action to follow. For example, you would change the custom action to **4010** if you wanted the custom action to occur after the files are installed (which by default is Event number **4000**). (It is important to note that the **CustomActionData** value is not necessary for actions running during the installation process as the condition values are available.)

Conditions can be placed on any custom action using the **Condition** property. This allows you to run different custom actions based on conditions that exist on a target computer during installation. For example, you might want to run different custom actions, depending on the operating system version on the target computer.

For more information about custom actions, see "Custom Actions" on MSDN.

For more information about working with custom actions, see "Adding and Removing Custom Actions in the Custom Actions Editor" on MSDN.

For more information about passing data from your installer to a custom action, see "Walkthrough: Passing Data to a Custom Action" on MSDN.

There are a number of different elements that would typically be added to your Installer package as custom actions. These include:

- Adding Files
- Modifying IIS Settings
- Adding Pre-defined Installation Components
- Adding Instrumented Assemblies

- Adding Serviced Components
- Pre-compiling ASP.NET applications

We will look at each of these in turn:

### Adding Files

In some cases you will want to add files at the end of an installation rather than earlier on. For example, you may create a log file based on information that the user has entered in the user interface and wish to add this to the files deployed on the server. In this case, a custom action would be appropriate.

### Modifying IIS Settings

Most IIS settings cannot be set with properties of the Web setup project. These include specifying directory security settings (for anonymous access, basic authentication, or Windows authentication), and specifying custom errors.

You can include custom actions in your Windows Installer file for applying the required settings.

When creating a custom action to set IIS 6.0 settings you have two different methods you can employ. They are:

- Create a script using the IIS ADSI Provider.
- Create a script using the IIS WMI Provider.

The following table compares the architecture and features of the IIS ADSI provider with those of the IIS WMI provider.

**Table 3.3: Comparison of IIS ADSI provider with IIS WMI provider**

| Issue | IIS ADSI provider | IIS WMI provider |
|---|---|---|
| Query capabilities | No. ADSI has no provision for queries. | Yes. By querying on metabase key types, the IIS WMI provider returns only the data you need. |
| Object model and access routes | COM: Scripts and programs. | COM: Scripts, programs, and UI tools; for example, the WMI Object Browser. |
| Extensible schema | Yes. ADSI provider supports metabase schema extensions. | No. The IIS WMI provider can return existing schema extensions, but cannot extend the metabase schema. |

| Issue | IIS ADSI provider | IIS WMI provider |
|---|---|---|
| Association or containment of related data | Properties are related to IIS metabase keys by containment. You can use the ADSI container object methods of the IIS Admin Objects to manipulate keys in the IIS metabase. You can create, delete, and move keys by creating, deleting, and moving IIS Admin Objects within container objects. You can also enumerate contained objects, such as virtual directories or servers with container object methods. ADSI supports property inheritance. | Properties are related to IIS metabase keys by containment or by association. Also, in conjunction with other providers, WMI supports associations with managed objects not in the metabase. An association in WMI describes a relationship between classes and is in itself defined by a class. This powerful concept allows management information about an entire system of associated components to be viewed and traversed for tasks such as troubleshooting. Navigating associations to other classes is not limited by containment. |

IIS 6.0 provides created scripts which use the WMI provider to perform certain functions, those functions are:

- **iisweb.vbs**. Script to create and manage Web sites on computers running a member of the Windows Server 2003 family with IIS 6.0.
- **iisftp.vbs**. Script to create and manage File Transfer Protocol (FTP) sites on computers running a member of the Windows Server 2003 family with IIS 6.0.
- **iisvdir.vbs**. Script to create, delete, and list Web virtual directories on computers running a member of the Windows Server 2003 family with IIS 6.0.
- **iisftpdr.vbs** Script to create, delete, and list FTP virtual directories on computers running a member of the Windows Server 2003 family with IIS 6.0.
- **iisback.vbs**. Script creates and manages backup copies of the IIS configuration (metabase and schema) of a remote or local computer.
- **iiscnfg.vbs**. Script imports and exports all, or selected, elements of an IIS metabase on a local or remote computer, or it copies the entire IIS configuration (metabase and schema) to another computer.
- **iisext.vbs**. Script can enable and list applications; add and remove application dependencies; enable, disable, and list Web service extensions; and add, remove, enable, disable, and list individual files.

## Adding Pre-defined Installation Components

As mentioned in Chapter 2, installation components are installed using a two step process. First the developer adds the installation component to a project and builds the application. Second you take the project output from the compiled application

and add it as a custom action. One important deployment consideration for installation components is that there is a restriction on using custom actions to install assemblies—they cannot run on assemblies that will be installed into the global assembly cache.

This means that all installation components that are added to assemblies should be placed in private assemblies. To achieve this you should create a separate private assembly that contains only installation components. This groups your installation components together and minimizes the number of assemblies that need to be added as custom actions in your setup project. Although this means that an assembly is deployed that is technically not used after installation is complete, it should not cause any problems for your application. In any case, you can then set the assembly to uninstall upon completion.

Another option is to include the installation components into existing assemblies. This is not a recommended method since it leaves unnecessary code in the program (that could be possibly exploited) after installation and may in fact interfere with the existing assembly's functionality.

► **To add a pre-defined installation component to a deployment project**

1. Select the setup project in **Solution Explorer**. On the **View** menu, point to **Editor**, and click **Custom Actions**.
2. In the **Custom Actions Editor**, select the **Install** node. On the **Action** menu, click **Add Custom Action**.
3. In the **Select Item in Project** dialog box, double-click the **Application Folder**.
4. Select the Primary output from project containing the pre-defined installation components, and click **OK**.
5. Add any property settings to the custom action as appropriate (for more information see the earlier section on Custom Actions in this chapter).

There are some specific considerations you should bear in mind when adding particular pre-defined installation components.

#### Adding Event Logging

The **EventLog** installer deploys the settings you have associated with **EventLog** components. These settings include:

- **Log**. The name of the log to read from or write to.
- **Source**. The name to use when writing to the log.

When you deploy your solution to the production environment, the installer creates the log you specify in the **Log** property; it also creates the event source that you specify in the **Source** property. An event source is simply a registry entry that indicates what logs the events. It is often the name of the application or the name of a subcomponent. Because event sources are necessary only when writing to an event log, it is not necessary to use an event log installer when you will be only reading, but not writing, to a log.

When deploying event logs with your application, you should be aware that:

- The **EventLog** installer can install event logs only on the local computer.
- If you set the **Source** property for an existing log, but the specified source already exists for that log, the **EventLogInstaller** deletes the previous source and recreates it. It then assigns the source to the log you are using.
- The name you use as a source for your log cannot be used as a source for other logs on the computer. An attempt to create a duplicated **Source** value throws an exception. However, a single event log can have many different sources writing to it.
- Event logs are supported only on the following platforms:
  - Windows NT Server 4.0
  - Windows NT Workstation 4.0
  - Windows 2000 family
  - Windows XP Home Edition
  - Windows XP Professional
  - Windows 2003 Server family
- The **EventLog** installer can only create event sources if it has administrative rights to do so. This should not cause problems if you install event logs and event sources using the **EventLogInstaller** class with Windows Installer files, because the installation is probably running with the Administrator security account.

For instructions on using the **EventLogInstaller** class, see "Walkthrough: Installing an Event Log Component" on MSDN.

#### Adding Message Queues

The **MessageQueue** installer creates and configures any message queues required by the application, along with settings such as path, journal settings, label, and so on. With message queues an application can send and receive messages, explore existing queues, create and delete queues, and perform a variety of other operations using a simple programming model.

#### Adding Performance Counter Categories and Counters

The **PerformanceCounter** installer allows you to install and configure a custom performance counter that your application needs in order to run.

When deploying performance counters with your application, you should be aware that:

- Performance counters are supported only on the following platforms:
  - Windows NT Server 4.0
  - Windows NT Workstation 4.0
  - Windows 2000 family

  - Windows XP Home Edition
  - Windows XP Professional
  - Windows 2003 Server family
- Performance counters need to be created by security accounts with administrative privileges — the easiest way to ensure this is to have your Windows Installer file run by an administrator.

#### Adding Windows Services

When developers design a project that will install Windows services, they use two types of installation components to install Windows service applications:

- **Service** Installer. You will use one **Service** Installer per service contained in your project. You will use this component to specify settings such as start type.
- **ServiceProcess** Installer. You will use only one **ServiceProcess** Installer for your Windows service application, regardless of how many services it contains. This component controls installation settings for the executable file itself, such as the security account used to run the service application.

For more information about creating and installing Windows service applications, see "Adding Installers to Your Service Application" on MSDN.

### Adding Instrumented Assemblies

As mentioned in Chapter 2, it is the responsibility of the application developer to ensure that instrumented assemblies are installed correctly by using an existing pre-defined installation class in the application project, or adding an installer class to the project.

Once the application project has been compiled, you can take the appropriate steps to ensure that the instrumented are deployed properly to the target computers:

► **To add an instrumented assembly to a deployment project**

1. Select the setup project in **Solution Explorer**. On the **View** menu, point to **Editor**, and select **Custom Actions**.
2. In the **Custom Actions Editor**, select the **Install** node. On the **Action** menu, click **Add Custom Action**.
3. In the **Select Item in Project** dialog box, double-click the **Application Folder**.
4. Select the Primary output from project containing the appropriate pre-defined installation component or installer class.
5. Add any property settings to the custom action as appropriate (for more information see the earlier section on Custom Actions in this chapter).

To install successfully, your installer classes must be run by a process with administrative privileges. The easiest way to ensure this happens is to have your installer

classes run as custom actions as part of a Windows Installer file — typically, Windows Installer files are run by users who are members of the local administrators group, so required privileges are already in place.

For more information about registering WMI schemas for your instrumented applications with the **ManagementInstaller** class, see "Registering the Schema for an Instrumented Application" on MSDN.

### Adding Serviced Components

Serviced components can be registered dynamically at runtime, but in some cases the process using a serviced component will not have the required privileges for dynamic registration to occur. You can register a serviced component programmatically using an installer class to deploy the serviced component. If you add the installer classes as custom actions to your setup project, then the Windows Installer files for your Web applications are typically run by an administrator, so the required privileges are met.

However, unlike the predefined installation components, you need to write the installation code for your components yourself—for example, you need to override the **Install** and **Uninstall** methods for your installer class. With these methods, you use the **RegistrationHelper** class to register (and unregister) your component. For an example of how to do this, see "Deploying N-Tier Applications with Visual Studio .NET Setup Projects" on GotDotNet.

For an example of how to create and use installer classes, see "Walkthrough: Using a Custom Action to Create a Database During Installation" on MSDN.

For more details about the RegistrationHelper class, see "Understanding Enterprise Services in .NET" on the .NET Framework Community Web site, GotDotNet.

### Pre-Compiling ASP.NET Applications

When an ASP.NET application is requested for the first time, two compilations take place:

- **Batch compile**. When an ASP.NET file (such as .asmx or .aspx) is requested for the first time, ASP.NET performs a "batch compile" on that directory. It compiles all of the code in the ASP.NET files in that directory (but not subdirectories) into MSIL.
- **JIT compile**. Like any assembly, the first time a method is accessed it is compiled from MSIL to machine code to execute. This occurs on a per method basis.

These two compilations cause a delay in the first request to a Web application. To avoid the delay, you can use a custom action to trigger either or both of these compilations in advance.

For information on how to pre-compile your ASP.NET application during a Windows Installer installation, see "Walkthrough: Using a Custom Action to Pre-Compile an Assembly During Installation" on MSDN.

Keep in mind that pre-compilation is not just a deployment issue. These compilations occur when the application is first started. This means that whenever the computer is rebooted, IIS is restarted, or the application is unloaded for any reason, both of these compilations occur again the next time the Web application runs. In addition to the Web server being restarted, ASP.NET applications can be unloaded and restarted for a variety of reasons, including:

- ASP.NET can be configured to restart the application periodically by using the **restartQueueLimit** setting in the **processModel** section of your configuration files.
- Every time you update any part of your Web application, ASP.NET effectively creates a new application instance for reflecting your changes.

If you choose to pre-compile your application to avoid the initial performance hit for your users, you should create a process (either manually or through script) separately from your deployment steps so that you can execute that process whenever the application is unloaded and needs to be started. For example, you might create a script that is run on a scheduled basis, or you might create a Windows service that monitors the status of your Web server and runs the script in response to your application being unloaded.

## Design the User Interface of Windows Installer Files

To define the steps your users must take to install your application, you can specify that a number of predefined user interface dialog boxes be presented during installation to present or gather information. You specify the dialog boxes to be displayed in the User Interface Editor.

► **To open the User Interface Editor**

1. In **Solution Explorer**, select a deployment project.
2. On the **View** menu, point to **Editor**, and click **User Interface**.

In addition to using the dialog boxes provided with Visual Studio .NET setup and deployment projects "as is," you can customize them. To do this, you need to add a dialog box to your project within Visual Studio .NET 2003, and then modify that dialog box outside of Visual Studio using a tool such as Orca.

This section can be critical as this is where you create the "User Experience." At the same time, it is important when creating these interface dialogs to know what information you need from the user. For example if you need to know a username and password to access a remote server to apply an add-in, or simply want to know what name they want to use for a virtual directory.

As custom actions are run after the actual installation is complete, they do not have access to properties used to control installation. This includes any information you may have gathered from the user. If you need to pass information from the installer

to a custom action, you can do so by setting the **CustomActionData** property for the field that contains the user gathered data.

For more information about each of the dialog boxes you can add to Windows Installer, see "Deployment Dialog Boxes" on MSDN.

For more information about Orca and the other Windows Installer Platform SDK tools, see the Windows Installer Platform SDK section later in this chapter.

## Building the Installer File

The Windows Installer file containing your setup logic is created when you build the setup project. The contents of the Windows Installer file, and whether any other files are created, are determined by the settings you choose for the project. You can control these settings in the **Property Pages** dialog box. (You can view this dialog box by right-clicking your setup project in the Solution Explorer and then clicking **Properties**).

### Choosing How the Files Are Packaged

In most cases, you will choose to package your files in a single setup file (an .msi). This allows you to distribute one file that contains all the files needed for installation. You can specify how files will be packaged within a Windows Installer file. The following options are available:

- **In setup file**. This allows you to distribute one file that contains all of the files needed for installation. The .msi file also contains the logic needed to complete the installation.
- **As loose uncompressed files**. All of the files required by your application are placed in the same directory as the .msi file (and subdirectories if your solution includes them). The .msi file itself contains the setup logic necessary to complete the installation and relies on the other files being present at install time. This option can be useful because it allows users to retrieve individual files, perhaps from the installation CD, after installation is complete if those files are damaged or have been deleted from their computers.
- **In cabinet file(s)**. To distribute more manageable file sizes, you should use the cabinet file(s) option. You can choose the maximum file size for your CAB files, which can make distribution of these files easier than for one large installer file. For example, if you plan to distribute files on a series of floppy disks, set the maximum size to 1440 KB (1.44 MB). Your files will be packaged in one or more CAB files in the same directory as the .msi file. Similar to the loose uncompressed files option, the .msi file contains the setup logic necessary to complete the installation and relies on the CAB files being present at install time.

### Compressing the Installer File

To manage the size of the installer file, you can set either of two levels of compression:

- **Optimized for speed**. Files are compressed to install faster, but result in a larger file size.
- **Optimize for size**. Files are compressed to a smaller size, but may result in a slower build and installation.

These settings apply only to CAB files and to solutions packaged in a single .msi file, and not to loose, uncompressed files. You should consider optimizing for size when users install your solution over a slow dial-up link. For scenarios where size is not an issue, such as when the installer will be distributed on CD or DVD, you might consider optimizing for speed.

### Authenticode Signing

The Internet itself cannot provide any guarantee about the identity of the software creator. Nor can it guarantee that any software downloaded was not altered after its creation. Browsers can exhibit a warning message explaining the possible dangers of downloading data of any kind, but they cannot verify that code is what it claims to be. One approach to providing guarantees of the authenticity and integrity of files is attaching digital signatures to those files. A digital signature attached to a file positively identifies the distributor of that file and ensures that the contents of the file were not changed after the signature was created.

Visual Studio .NET deployment tools make it possible for you to sign an installer, a merge module, or a CAB file using Authenticode. You must first obtain a digital certificate in order to use these features of Visual Studio .NET deployment tools. The Signcode.exe utility available in the Microsoft Platform SDK can be used to obtain a digital certificate.

You can set your installer file to be digitally signed by selecting the Authenticode Signature check box in the project properties. You must then provide one of the following:

- **Certificate file**. Specifies an Authenticode certificate file (.spc) that be used to sign the files.
- **Private key file**. Specifies a private key file (.pvk) that contains the digital encryption key for the signed files.
- **(Optional) Timestamp server URL**. Specifies the Web location for a timestamp server used to sign the files.

You can obtain digital certificates from a number of certificate authorities, such as VeriSign or Thawte.

More information about digital certificates is available in the "Platform SDK CryptoAPI Tools Reference."

For more information about using Authenticode signing for your Visual Studio .NET setup projects, see "Deployment and Authenticode Signing" on MSDN.

## Other Windows Installer Package Considerations

If you choose to deploy your Framework applications using Windows Installer packages, there are a number of other things to consider before you deploy the application in a production environment.

### Documenting Installer Creation

One of the advantages of creating Windows Installer packages is that the uninstall process is generally straightforward – you simply remove the program using Add/Remove Programs. However in some cases an uninstall will fail using Windows Installer. You should therefore document thoroughly the process used to create your Windows Installer package. This will allow you to create a manual uninstall process that you can test prior to deploying the application.

### Administrative Installation

As mentioned in Chapter 2, "Planning the Deployment of .NET Framework-based Applications", you may wish to deploy your application to one or more network servers and have users install the application from that point. Windows Installer can perform an administrative installation of an application to a network installation point. An administrative installation installs a source image of the application onto the network that is similar to a source image on a CD. If the user can choose to *run from source* when installing over the network, the installer uses most of the application files directly over the network.

---

**Note:** For more information on Administrative Installations, see "Administrative Installations" in the Platform SDK.

---

### .NET Framework Bootstrapping

If the Framework is not pre-installed on your target computers, but you are deploying a Framework application, you need to provide a method to install the Framework so that your application can run. One option is to use a launch condition to direct the user to a URL where the Framework can be installed. Another is to bootstrap the Framework to your application. To do this you should:

- Package the redistributable .NET Framework (Dotnetfx.exe) with your application.
- Run a check that determines whether the .NET Framework is installed on the target computer.
- Install the .NET Framework if necessary.
- Install your application.

You cannot include the .NET Framework in your Windows Installer files. Instead, you must develop a separate bootstrapping application that performs the tasks listed above. A sample application (Setup.exe) that performs these tasks can be downloaded from "Microsoft .NET Framework Setup.exe Bootstrapper Sample" on MSDN.

The Setup.exe bootstrapping application sample demonstrates how to create a setup program that verifies whether the .NET Framework is installed. The code checks for a specified version number of the .NET Framework in the **HKLM\SOFTWARE\Microsoft\NETFramework\policy\v1** registry key. You can determine which version number should be checked for by examining the properties of the redistributable .NET Framework, Dotnetfx.exe. (For more information about Dotnetfx.exe, see Chapter 2, "Planning the Deployment of .NET Framework-based Applications.") The code performs a comparison between the build number in the registry key and the build number of the .NET Framework being hosted by the application. If there is not a matching build number in the registry key in this location, Setup.exe performs a silent install of the .NET Framework included with your application, and then runs your Windows Installer file.

You can use the sample for your own applications by downloading the precompiled application and simply modifying the Setup.ini file that ships with the sample. Alternatively, you can download the source code for the sample and modify it to meet your specific needs.

For more information about how to work with this sample, see "Redistributing the .NET Framework" on MSDN.

### Windows Installer Bootstrapping

If you are not certain whether the target computers have Windows Installer 2.0 available, you should include the Windows Installer bootstrapping application with your .msi file. For your .msi files to install your application successfully, they require that the target computer have the Windows Installer 2.0 service present. A bootstrapping application includes the files needed to install Windows Installer on the target computer, if it is not already installed, before continuing with your application setup. You have two main options for including the bootstrapping application:

- **Windows Installer bootstrapping application**. A bootstrapping application is included for installation on a Windows-based computer. Your .msi file automatically checks for the Windows Installer service on the target computer and, if it is not present or is an incompatible version, launches the setup of the Windows Installer service prior to installing your application.

  To use this bootstrapper in Visual Studio .NET 2003, go to the deployment project's property page and by default the Windows Installer 2.0 bootstrapper is included

- **Web bootstrapping application**. A bootstrapping application is included for download over the Web. As in the case of the Windows Installer bootstrapping application, this option launches the setup of the Windows Installer service prior to installing your application, if necessary. The only distinction is that you don't distribute the bootstrapping application with your .msi file, but instead make it available for download from a Web server.

  A Web bootstrapping application has authentication built in, so using this approach rather than CAB files for distributing the bootstrapping application is recommended. If you choose this option, you can use the **Settings** option to specify the Web location where your application and the Windows Installer executable files are available for download.

  **Note:** In some cases, browser content expiration settings can cause the Web bootstrapping application to fail to download. For more information about how to remedy this problem, see Knowledge Base article Q313498, "BUG: Error 1619 When You Install a Package That Uses Web Bootstrapper."

Every installation that attempts to use the Microsoft Windows Installer begins by checking whether the installer is present on the user's computer, and if it is not present, whether the user and computer are ready to install Windows Installer. Two setup applications, InstMsiA.exe and InstMsiW.exe, are available with the Windows Installer SDK that contains all of the logic and functionality to install Windows Installer.

**Note:** InstMsiA.exe is the American National Standards Institute (ANSI) version of the installer redistributable package, whereas InstMsiW.exe is the Unicode version. The version used by the bootstrapping application depends on the platform to which the Windows Installer service is being installed. InstMsiA.exe is used for Windows 95, Windows 98, and Windows Me, and InstMsiW.exe is used for Windows NT 4.0 and later.

You can include these files with your Windows Installer file to ensure that the correct version of the Windows Installer service is present before your .msi file runs.

However, a bootstrapping application must manage this process. If Windows Installer is not currently installed, the bootstrapping application must query the operating system to determine which version of the InstMsi is required. After the installation of Windows Installer has initiated, the bootstrapping application must handle return codes from the InstMsi application and handle any reboots that are incurred during the Windows Installer installation.

Visual Studio .NET setup projects make this bootstrapping process very simple. All you need to do is set some project properties for your .msi file, and the build process creates the appropriate bootstrapping application. On the property pages for your

deployment project, you can specify one of the following bootstrapping application options:

- **None**. No bootstrapping application will be included.
- **Windows Installer Bootstrapper**. A bootstrapping application will be included for installation on a Windows-based computer.
- **Web Bootstrapper**. A bootstrapping application will be included for download over the Web. A bootstrapping application has authentication built in, so using a bootstrapping application rather than .cab files is the preferred method for downloading.

When you build your setup project, the bootstrapping application (Setup.exe and associated .ini file) and the InstMsi file are included in the build output—users need to run Setup.exe rather than the .msi file to install your application.

For more information about the process for installing bootstrapping applications, see "Bootstrapping" in the Platform SDK.

## Manipulating Installer Packages

Merge modules (.msm files) and .msi files contain databases that define setup components and logic for your application or component. The Orca database editor is a table-editing tool available in the Windows Installer SDK that you can use to edit your .msi or .msm files. You can use Orca if you need to edit the database tables directly to control setup logic and components for an .msi file or .msm file. Although you can edit any .msi file or .msm file using Orca, you should do so for third-party installers only under instruction from the software vendor.

A more common use for Orca is to create patch creation properties (PCP) files. You can then use MsiMsp.exe to create a Windows Installer patch (MSP) file from your PCP. The MSP patch file can then be applied to your application.

Orca.exe and MsiMsp.exe are both available in the Microsoft Windows Installer SDK. To download the Microsoft Windows Installer SDK, see "Samples, Tools, and Documentation (x86) v1.2 for Windows 2000, Windows 9x, and Windows NT 4.0" on MSDN.

For a complete list of other utilities included in the Windows Installer Platform SDK and for more in-depth information, see the Windows Installer Platform SDK documentation.

### Use Orca to Create Nested Installations

In certain circumstances, you might need to launch one .msi file from another. This is known as a nested installation. Windows Installer technology supports nested installs, but the setup and deployment projects do not currently natively support them. However, you can use Orca or another .msi file editing tool to add a nested installation action to the custom action table of your primary .msi file in order to have that .msi file launch another one. Nested installations are performed in the same transactional context as the launching .msi file, so failure in the nested installation rolls back all work carried out by the main .msi file. However, there are some

issues with nested installations that make them suitable in certain circumstances only. For example:

- Nested installations cannot share components.
- An administrative installation cannot also contain a nested installation.
- Patching and upgrading may not work with nested installations.
- The installer may not properly calculate the disk space required for the nested installation.
- Integrated progress bars cannot be used with nested installations.

For more information about nested installations, see "Nested Installation Actions" on MSDN.

For step-by-step instructions on creating nested installation, see Knowledge Base article Q306439, "HOWTO: Create a Nested .msi Package."

#### Automating the Creation of Installers

If you are an ISV, or if you need to deploy a large number of .NET-based applications for your enterprise, you might want to automate the creation of your setup routines. For example, you might want to build your own application for creating setup programs to standardize deployment practices, or you might want to write scripts that create installers for multiple applications. If the other factors described in this chapter bring you to the conclusion that an .msi (or .msm) file is appropriate for your solutions, you can automate the creation of your setups by using the automation interface provided by the Windows Installer, rather than creating your installers using Visual Studio .NET.

For in-depth information about the Windows Installer automation interface, see the "Automation Interface" section of the Platform SDK on MSDN.

## Collection of Simple Build Objects

As mentioned in Chapter 2, Windows Installer represents the most effective solution for deploying all but the most simple of .NET-based applications. However there are circumstances in which a simple file-copy distribution method may be appropriate, mainly in the case of ASP.NET applications. If you do use a file copy distribution method, there are some special considerations you need to take into account. These include the deployment of:

- Files and Folders
- Assemblies
- Application Resources
- CAB Files
- Registry Settings

We will cover each of these in turn:

## Files and Folders

If you use a copy operation to deploy the build outputs for your Web application, you must make sure that all required files are included. Your Web application probably consists of more than just the build outputs—it contains the .aspx files, graphics and other resources, and so on. You can ensure that all required files are deployed by writing a script or deployment utility that interrogates the Visual Studio .NET project file (*.csproj or *.vbproj) to see which other files are required for your project. The project file is actually an XML-based text file, so retrieving the required information from it is a simple matter. It has a **<Files>** element that lists the other files used by the solution and specifies each file's build action, which you use to determine whether it should be deployed. The following is an example of the **<Files>** element in a project file:

```
<Files>
 <Include>
  <File RelPath = "AssemblyInfo.cs" SubType = "Code" BuildAction = "Compile" />
  <File RelPath = "Global.asax" SubType = "Component" BuildAction = "Content" />
  <File RelPath = "Global.asax.cs" DependentUpon = "Global.asax" SubType = "Code"
        BuildAction = "Compile" />
  <File RelPath = "web.config" BuildAction = "Content" />
  <File RelPath = "WebForm1.aspx" SubType = "Form" BuildAction = "Content" />
  <File RelPath = "WebForm1.aspx.cs" DependentUpon = "WebForm1.aspx"
        SubType = "ASPXCodeBehind" BuildAction = "Compile" />
 </Include>
</Files>
```

The **BuildAction** attribute can be one of the following:

- **None**. The file is not included in the project output group and is not compiled in the build process. An example is a text file that contains documentation, such as a Readme file.
- **Compile**. The file is compiled into the build output. This setting is used for code files.
- **Content**. The file is not compiled, but it is included in the Content output group. For example, this setting is the default value for an .aspx, .htm, or other kind of Web file or graphic.
- **Embedded Resource**. This file is embedded in the main project build output as a DLL or executable file. It is typically used for resource files.

Your deployment script or utility should iterate through the **<File>** elements contained in the **<Include>** section of the **<Files>** node, and should retrieve the **RelPath** values for each file that has a **BuildAction** set to **Content**. Any file that has the **BuildAction** set to **Compile** or **EmbeddedResource** is included in the build outputs

themselves. (Of course, you should also remember to include the build output in your copy operation.) Those files set to **None** are not typically required by your solution.

## Assemblies

There are some different considerations for deploying assemblies depending on whether they are private or shared, and if they are instrumented.

### Private Assemblies

If your private assemblies are not strong-named, you can use a simple copy operation, such as XCOPY, to install them into the production environment. Private assemblies are self-describing and, unlike COM components, do not require information to be added to the registry.

If your private assembly is strong-named, then it does need to be registered. You should the regsvcs.exe tool to register your assembly.

### Deployment of Shared Assemblies

You can install an assembly into the global assembly cache without using Windows Installer technology, by using the Gacutil.exe utility. If you use the Gacutil.exe tool, you should normally use the **/ir** switch, which installs assemblies into the global assembly cache with a traced reference. These references can be removed when the assembly is uninstalled by using the **/ur** switch.

As with any form of reference counting, using traced references with shared assemblies will only work properly provided it is implemented consistently. If you implement reference counting, you should make sure that you always do so, or it is of little or no use.

---

**Note:** Theoretically you can deploy an assembly into the global assembly cache by simply dragging and dropping it to the global assembly cache folder in Windows Explorer. However, this method does not implement any reference counting and so should generally be avoided. In any case the new assembly will not be used until you modify the reference to the strong name in any application that refers to it.

---

### Deployment of Instrumented Assemblies

As mentioned earlier, instrumented assemblies have special requirements as they require a registration stage, when the schema is discovered and registered in the repository. In practice, this means you need to deploy an instrumented assembly you need to run the Installutil.exe utility against the *.dll file corresponding to the assembly.

For more information about using the Installutil.exe utility, see “Installer Tool (Installutil.exe)” on MSDN.

## Application Resources

If you are deploying application resources, such as message queues, event logs, and performance counters, these need to be registered. To register these resources, you should run the installutil.exe utility against the *.dll file corresponding to the application resource.

## COM/COM+ Objects

If your Web application includes legacy COM objects, they need to be installed and registered properly before they can be used. You can use the RegSvr32 tool to manually register your COM libraries.

## IIS 6.0 Settings

If you use the Visual Studio .NET **Copy Project** command, a new virtual directory is created for you on the target Web server. However, for simple copy operations, the IIS settings are not copied from your development virtual directory and applied to the production copy of your solution. The new virtual directory inherits the default settings from the Web site. Again, you need to apply the appropriate settings separately, either by developing and running IIS scripts or by manually applying the setting your Web application requires.

## Serviced Components

You will need to give special consideration to packaging your Web applications if they include serviced components. You can either have your serviced components registered dynamically, or you can use installer classes to ensure that they are installed correctly. If you use installer classes, and you are not packaging your solution in a Windows Installer file, you need to run them with the Installutil.exe tool.

## Applying Security Policy

Rather than deploying security policy with Windows Installer files, you can use scripts and batch files along with the Code Access Security Policy tool (Caspol.exe) to apply security policy settings. If you use this approach, you need to disable the prompt for your users that security policy has been changed. You can achieve this by including the following command as the first entry in your batch file:

```
Caspol –pp
```

For more information about using this tool, see "Code Access Security Policy Tool (Caspol.exe)" on MSDN.

For more information about administering security policy, see "Security Policy Administration Overview" on MSDN

For more information about specifying security policy in code for application domains, see "Setting Application Domain-Level Security Policy" on MSDN.

For more information about using the Mscorcfg.msc wizard, see ".NET Framework Configuration Tool (Mscorcfg.msc)" on MSDN.

For answers to frequently asked question regarding the deployment of security policy, see ".NET Framework Enterprise Security Policy Administration and Deployment" on MSDN.

### Registry Settings

Use the Regsvcs.exe utility from the .NET Framework SDK to manually register your assembly containing serviced components.

As mentioned earlier in this chapter, to register a .NET assembly you need to create an entry in the registry that points to Mscoree.dll, which loads and executes the .NET assembly. If you are not using Windows Installer technologies, you can use the Regasm.exe tool to register the .NET assembly. There is a **Codebase** switch on the Regasm.exe tool that creates a **Codebase** entry in the registry. The **Codebase** entry specifies the file path for an assembly that is not installed in the global assembly cache. This switch is mainly used only in development and you should not specify this option if you subsequently install the assembly that you are installing into the global assembly cache.

## Summary

Deploying applications under the .NET Framework can be much easier than in previous development environments. However, some .NET-based applications are very complex with lots of interactions. You need to be sure that you can deploy these applications effectively to your environment. You should use the advice given in this chapter to help you perform the steps required to deploy your applications correctly.

## More Information

For more information on No Touch Deployment:
No Touch Deployment in the .NET Framework
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vbtchno-touchdeploymentinnetframework.asp*

For more information on code access security:
*Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication* (Microsoft Press, ISBN: 0-7356-1890-0) or on MSDN
*http://msdn.microsoft.com/library/en-us/dnnetsec/html/secnetlpMSDN.asp?frame=true*

For more information on the Visual Studio setup wizard:
Distributing the Visual J# .NET Redistributable Package
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vjsharp/html/vjsamDistributingVisualJNETFramework.asp*

For more information on preventing files in a Project Output Group from being deployed:
Excluding Files from Project Output Groups:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxtskexcludingfilesfromprojectoutputgroup.asp*

For more information on Special Folders:
Special Folders and Custom Folders
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vbconCustomFoldersSpecialFolders.asp*

For more information about deploying all required COM components:
Deploying COM+ Applications
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cossdk/htm/pgdeployingapplications_5xmb.asp*

and

Application Deployment Using Microsoft Management Technologies
*http://www.microsoft.com/windows2000/techinfo/howitworks/management/apdplymgt.asp*

For step-by-step instructions on how to create and use merge modules:
Creating or Adding a Merge Module Project
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskcreatingoraddingmergemodule.asp*

For more information about managing the security requirements for downloaded CAB files:
Writing Secure Managed Controls
*http://msdn.microsoft.com/library/en-us/cpguide/html/cpconwritingsecuremanagedcontrols.asp*

For more information about working with launch conditions:
Launch Condition Management in Deployment:
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vxconLaunchConditionManagementInDeployment.asp*

For more information about using the Condition property:
Deployment Conditions
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vxconDeploymentConditions.asp*

For more information on adding a launch condition for the J# redistributable package:
Distributing the Visual J# .NET Redistributable Package
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vjsharp/html/vjsamDistributingVisualJNETFramework.asp*

For more information on adding an MDAC launch condition:
Adding a Launch Condition for Microsoft Data Access Components
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxtskaddinglaunchconditionformicrosoftdataaccesscomponents.asp*

For instructions on how to add registry keys to your projects:
Adding and Deleting Registry Keys in the Registry Editor
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskAddingDeletingRegistryKeys.asp*

For more information about modifying the registry table
Registry Table Information
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/registry_table.asp*

For instructions on how to specify values for registry keys
Adding and Deleting Registry Keys in the Registry Editor
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskAddingDeletingRegistryValues.asp*

For more information about creating default registry values:
Creating a Default Registry Value in the Registry Editor
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskCreatingDefaultRegistryValue.asp*

For instructions on how to import registry files into your deployment projects:
Importing Registry Files in the Registry Editor
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskImportingRegistryFiles.asp*

For more information about editing file associations:
File Types Management in Deployment
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbconTheAssociationsEditor.asp*

For an example of a localized Installer package:
Localizing a Windows Installer Package
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/localizing_a_windows_installer_package.asp*

For more information about programmatically defining these IIS 6.0 Settings:
Programmatic Administration Guide
*http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/windowsserver2003/proddocs/server/prog_prog.asp*

For more information about IIS Security in ASP.NET applications:
Authentication in ASP.NET: .NET Security Guidance
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/authaspdotnet.asp*

For more information about custom actions:
Custom Actions
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbconCustomActions.asp*

For more information about working with custom actions:
Adding and Removing Custom Actions in the Custom Actions Editor
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskAddingRemovingCustomActions.asp*

For more information about passing data from your installer to a custom action:
Walkthrough: Passing Data to a Custom Action
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vxwlkWalkthroughPassingDataToCustomAction.asp*

For an example of how to create and use installer classes:
Walkthrough: Using a Custom Action to Create a Database During Installation
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxwlkwalkthroughusingcustomactiontocreatedatabaseduringinstallation.asp*

For more information about using the Installutil.exe utility:
Installer Tool (Installutil.exe)
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconinstallerutilityinstallutilexe.asp*

For instructions on using the EventLogInstaller class:
Walkthrough: Installing an Event Log Component
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbwlkWalkthroughCreatingEventLogInstallers.asp*

For more information about creating and installing Windows service applications:
Adding Installers to Your Service Application
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbtskAddingInstallersToYourServiceApplication.asp*

For more information about registering WMI schemas for your instrumented applications with the ManagementInstaller class:
Registering the Schema for an Instrumented Application
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconregisteringtheschemaforaninstrumentedapplication.asp*

For an example of how to register and unregister your component using the RegistrationHelper class:
Deploying N-Tier Applications with Visual Studio .NET Setup Projects
*http://www.gotdotnet.com/team/xmlentsvcs/deployntier.aspx*

For more details about the RegistrationHelper class:
Understanding Enterprise Services in .NET on the .NET Framework Community Web site, GotDotNet
*http://www.gotdotnet.com/team/xmlentsvcs/espaper.aspx*

For information on how to pre-compile your ASP.NET application during a Windows Installer installation:
Walkthrough: Using a Custom Action to Pre-Compile an Assembly During Installation
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxwlkwalkthroughusingcustomactiontoprecompileassermblyduringinstallation.asp*

For more information about each of the dialog boxes you can add to Windows Installer:
Deployment Dialog Boxes
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbconDeploymentDialogs.asp*

You can obtain digital certificates from a number of certificate authorities, such as VeriSign (*http://www.verisign.com/*) or Thawte (*http://www.thawte.com/*).

For more information about digital certificates:
Platform SDK CryptoAPI Tools Reference
*http://msdn.microsoft.com/library/en-us/security/security/cryptoapi_reference.asp*

For more information about using Authenticode signing for your Visual Studio .NET setup projects:
Deployment and Authenticode Signing
*http://msdn.microsoft.com/library/en-us/vsintro7/html/vxconDeploymentAuthenticodeSigning.asp*

For more information on Administrative Installations:
Administrative Installations in the Platform SDK
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/administrative_installation.asp*

To download a sample application (Setup.exe) that bootstraps the .NET Framework:
Microsoft .NET Framework Setup.exe Bootstrapper Sample
*http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/msdn-files/027/001/830/msdncompositedoc.xml*

For more information about how to modify the Setup.exe Bootstrapper sample:
Redistributing the .NET Framework
*http://msdn.microsoft.com/library/en-us/dnnetdep/html/redistdeploy.asp*

To fix browser content expiration settings can cause the Web bootstrapper application to fail to download:
Knowledge Base article Q313498, "BUG: Error 1619 When You Install a Package That Uses Web Bootstrapper
*http://support.microsoft.com/default.aspx?ln=EN-US&pr=kbinfo&#mskb*

For more information about the process for installing bootstrapping applications:
Bootstrapping in the Platform SDK
*http://msdn.microsoft.com/library/en-us/msi/boot_4puv.asp*

To download the Microsoft Windows Installer SDK:
Samples, Tools, and Documentation (x86) v1.2 for Windows 2000, Windows 9x, and Windows NT 4.0 on MSDN
*http://msdn.microsoft.com/downloads/sample.asp?url=/MSDN-FILES/027/001/457/msdncompositedoc.xml*

For more information about nested installations:
Nested Installation Actions
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/cact_260j.asp*

For step-by-step instructions on creating nested installation:
Knowledge Base article Q306439, "HOWTO: Create a Nested .msi Package"
*http://support.microsoft.com/default.aspx?ln=EN-US&pr=kbinfo&#mskb*

For in-depth information about the Windows Installer automation interface:
Automation Interface" section of the Platform SDK on MSDN
*http://msdn.microsoft.com/library/en-us/msi/auto_8uqt.asp*

For more information about using the Caspol.exe tool:
Code Access Security Policy Tool (Caspol.exe
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfcodeaccesssecuritypolicyutilitycaspolexe.asp*

For more information about administering security policy:
Security Policy Administration Overview
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconsecuritypolicyadministrationoverview.asp*

For more information about specifying security policy in code for application domains:
Setting Application Domain-Level Security Policy
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconsettingapplicationdomainlevelsecuritypolicy.asp*

For more information about using the Mscorcfg.msc wizard:
.NET Framework Configuration Tool (Mscorcfg.msc)
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconnetframeworkadministrationtoolmscorcfgmsc.asp*

For answers to frequently asked question regarding the deployment of security policy:
.NET Framework Enterprise Security Policy Administration and Deployment
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/entsecpoladmin.asp*

For more information on deploying applications using no touch deployment:
Security and Versioning Models in the Windows Forms Engine Help You Create and Deploy Smart Clients
*http://msdn.microsoft.com/msdnmag/issues/02/07/NetSmartClients/default.aspx*

For a list of articles on Windows Forms-based applications:
Articles List for Windows Forms Applications
*http://www.windowsforms.net/Default.aspx?tabindex=3&tabid=40#Deployment*

# 4

# Maintaining .NET Framework-based Applications

Application deployment isn't over once applications have been successfully installed on the target computers. Your applications may receive minor or major updates, new applications may be installed alongside existing applications, and new versions of the Framework may be installed. All of these can present challenges, and you need to be aware of the issues involved.

In this chapter we start by looking at updating applications, using the methods for deployment already discussed in the previous chapters—no-touch deployment, Windows Installer deployment, and deploying a simple collection of build objects, plus a methodology specifically aimed at providing automatic updates to your applications. Then we examine the various scenarios you are likely to encounter when running multiple applications alongside each other, and multiple versions of the Framework.

## Upgrading .NET Framework-based Applications

Occasionally, you need to upgrade Framework applications that have already been deployed to the production environment. Typically, applications are upgraded in order to:

- Apply bug fixes to remedy problems with your application.
- Provide enhancements to existing features.
- Provide new features or capabilities.

In some cases you will decide to upgrade an application to take advantage of the new functionality of a new release of the Framework. For example version 1.1 makes some enhancements in the area of code access security. You may wish to adjust your code to take advantage of this.

One of the major advantages of the .NET architecture is that it provides comprehensive support for modifying or upgrading applications after they have been deployed.

## Planning an Upgrade Strategy

In many cases you will upgrade your solutions using similar mechanisms to those with which you originally deployed your application. For example, if you originally packaged your application in a Windows Installer file, then you will typically upgrade application files with an updated Windows Installer file or a Microsoft patch (.msp) file, managed by Windows Installer. If you distribute that solution with SMS, you will often re-deploy your solution using the same approach. However, there will undoubtedly be some situations a different deployment strategy is appropriate for an upgrade. For example, it is possible that a change to a configuration file or Web file might best be implemented simply by copying the new version, even though the application might originally have been packaged in a Windows Installer file and distributed with Active Directory Group Policy for software deployment.

---

**Note:** Chapter 2, "Planning the Deployment of .NET Framework-based Applications" provides information to help determine a deployment strategy for your application. You can also use this information to help you choose the correct deployment strategy for your upgrade.

---

## Levels of Upgrade

Your upgrade strategy may vary depending on the scale of the update. Upgrades are categorized into three different levels:

- **Major upgrades**. These contain new features or significant changes. A major upgrade of Appv1.0 would generally be known as Appv2.0.
- **Minor upgrades**. These contain minor product changes. A minor upgrade of Appv1.0 would typically be known as Appv1.1.
- **Small updates**. These contain patches or bug fixes. Typically a version number will not be incremented by a small update.

You can use the version numbers of applications to manage certain aspects of their deployment. For example, in some cases there is a particular order in which updates to an application need to be applied. By defining version numbers it is easy to differentiate between version numbers and therefore define the appropriate order for updates. You may also use version numbers to ensure that older versions of an application are not installed over newer ones, or that an upgrade removes previous versions of an application. This is most easily achieved if you use Windows Installer packages for your applications and application updates. For more details, see the Windows Installer section later in this chapter.

## Upgrade Options

There are four major choices when it comes to deploying an application upgrade. These are:

- No-touch deployment
- Windows Installer package deployment
- Deploying a simple collection of build outputs
- Configuring Automatic Updates

We will look at each of these in turn:

## No-Touch Deployment Upgrades

One of the main benefits to an application using no-touch deployment is how easy it is to upgrade the application. If you change any files on the Web server, the next time the client launches the application, it will automatically update, downloading the newer files as needed. This happens because the .NET Framework automatically checks the timestamp of files that are called, to see if they need to be downloaded again or if they can simply be run from the assembly cache download folder or the Temporary Internet Files folder.

You have two options for updating an application deployed by no-touch deployment:

- Manually copy any modified files onto the Web server.
- Recompile the application with the build output set to be the production Web server.

Generally the first option is most suitable when just a few files are modified. However, for more significant changes, it often is more sensible to simply redeploy the application by recompiling it.

Application code is automatically updated when the user makes a request back to the Windows Form. The .NET Framework will automatically check the timestamp of the assembly to see if it needs to download again or if it can simply be run from the user's assembly download cache.

Although issuing updates using a no-touch deployment method is generally very straightforward, clients may potentially have problems during an upgrade. If you update the directory while clients are using the application, it is possible that a client could download old code initially and then attempt to download further code which has since been updated. This will lead to unpredictable results and will often cause your application to fail. To get around this problem, the simplest solution is to deploy any significant updates to a separate directory on the Web server and when deployment is complete, change any pointers to the new location.

## Upgrading Using Windows Installer

Windows Installer offers the most comprehensive solution for updating .NET-based applications. It contains some functionality specifically aimed at solving some of the problems of updating applications, including version control.

### Windows Installer Upgrade Strategies

If you are planning to upgrade your application using Windows Installer technology, you have three choices for implementing the upgrade:

- Create a completely new deployment project and add your project outputs and other files, just as you did for the first release of the application.
- Upgrade the existing deployment project by ensuring changes to your application are reflected in the setup, and then rebuild that version.
- You can build a patch package (.msp) and apply it to the currently installed application.

We will discuss each of these in turn:

#### Creating a New Deployment Project

In cases when you are creating a significant change to an application, it may be most appropriate to create a brand new deployment project rather than trying to modify an existing one. In some situations the changes to your deployment project will be so substantial that it is easier to start again than try to modify what is already there. However, there are some issues with creating a completely new project in this way.

Probably the most significant is the relationship between your new deployment project and the previous one. Windows installer uses various versioning and upgrade properties and GUIDs to associate Windows Installer files with applications that are already installed. If you create a new setup project, then Visual Studio .NET assigns new GUIDs, which prevents upgrade management features (such as removing earlier versions and preventing earlier versions from installing over later ones) from working. You can, of course, apply the original GUIDs to your new setup project, but this is an error-prone process, because of the format of the GUIDs.

The actual process of creating a new deployment project for a latest version of the application is no different to creating a deployment project for the first version. For more details on creating deployment projects, see Chapters 2 and 3 of this guide.

#### Updating an Existing Deployment Project

For most minor upgrades and many major upgrades, updating an existing deployment project is easier than creating a completely new one. For example, you will already have designed the user interface, registry settings, file associations, custom actions, launch conditions, and so on, for your original application. If you are simply shipping new versions of your executable file or DLLs, you will not want to step

through these processes again. Instead, you can simply update your project output or individual files, and those other features that do not need changing will remain in place. In this case the correct GUID is associated with the application, and so upgrade management features should work properly without you having to modify the package.

#### Patch Packages

Patch packages, as the name suggests, are most suitable for minor upgrades or small updates. The main advantage of a patch package over a Windows Installer package (new or modified) is that patch package only contains changes to your application and is therefore normally much smaller. This can be particularly useful when the original application is very large or in low-bandwidth environments.

Windows Installer can use a patch package to patch both local and administrative installations. A patch package is a storage file, similar to a descriptor file, and has an .msp file name extension and its own class identifier (CLSID). A patch package does not include a database like a regular installation package. Instead it contains at minimum one database transform that adds patching information to the database of its target installation package. The installer uses this information to apply the patch files that are stored in the patch package.

Visual Studio .NET does not currently support patching an installed application using Windows Installer technology. You need to use Windows Installer Platform SDK tools (such as Orca.exe and MsiMSP.exe), or third-party utilities, to create patch files that can be applied to an application that was originally installed with a Visual Studio .NET setup project.

Developers create a patch package by generating a patch creation file and using the Msimsp.exe tool to call the **UiCreatePatchPackage** function in Patchwiz.dll. Both Msimsp.exe and Patchwiz.dll are provided in the Windows Installer SDK.

Because the application of a patch to a Windows Installer package results in a new .msi file installing much of the original code, the new .msi file must remain compatible with the layout of the original source. This means that when you create a patch package you must follow the following guidelines:

- Use an uncompressed setup image to create a patch. For example, an administrative image or an uncompressed setup image from a CD. This is because the patch creation process is unable to generate binary patches for files compressed in cabinets.
- Do not change the primary keys in the file table between the original and new .msi file versions.
- Do not move files from one folder to another.
- Do not move files from one CAB file to another.
- Do not change the order of files in a CAB file.

► **To create a patch package**

1. Create an uncompressed setup image from either a CD or administrative installation.
2. Copy that uncompressed setup image to a second location, creating a second uncompressed image.
3. Add the additional files, components and settings to the second uncompressed image.
4. Create a Patch Creation Properties File (*.pcp). (A blank patch creation properties file, template.pcp, is provided with the Windows Installer SDK.)
5. Use the Orca.exe tool to modify the .pcp file.

   The following database tables are required in every .pcp file:

   - Properties Table
   - ImageFamilies Table
   - UpgradedImages Table
   - TargetImages Table

   The following are optional tables:

   - UpgradedFiles_OptionalData Table
   - FamilyFileRanges Table
   - TargetFiles_OptionalData Table
   - ExternalFiles Table
   - UpgradedFilesToIgnore Table
6. Run the Msimsp.exe tool (also in the Windows Installer SDK) using the following command line: **`Msimsp.exe -s <pcpfilepath> -p <mspfilepath>.`**

At this point you now have an .msp file that is the difference between the first and second uncompressed installation images.

To apply the patch package to a local installation the following command line can be used:

```
msiexec /p <*.msp> REINSTALL=ALL REINSTALLMODE=omus
```

For more information on patch packages, see "A Small Update Patching Example" on MSDN.

For more information on Windows Installer SDK tools, see the Windows Installer Development Tools portion of the Platform SDK on MSDN.

For more information on .pcp file properties see the UpgradedImages Table (PATCHWIZ.DLL) portion of the Platform SDK on MSDN.

## Versioning Installer Packages

The versioning features of Windows Installer files provide solutions to many of the issues associated with managing and upgrading multiple versions of the same application. By versioning your Windows Installer packages, you can ensure that older versions of an application are not installed over newer versions, and you can also make sure that previous versions of an application are removed when a new one is installed.

It is important to note that version numbers on Windows installer packages are maintained separately to version numbers of the applications they install, and can therefore be different. However, to avoid confusion as to which application version is contained within which .msi file, you should consider using the version number of each released application to identify the associated installer. For example, if your application has versions 1.0, 1.5, 1.51, and 2.0, you can use those same numbers for the version property of the Windows Installer file in which they are packaged. This makes it easier for you to determine which installer should be used for which application version.

The full version number of a Framework application is formatted in the following way:

```
<major version>.<minor version>.<build number>.<revision>
```

However, Windows Installer will only recognize the first three numbers when differentiating between versions.

Developers that build applications will often just specify the version number to be of the form:

```
<major version>.<minor version>.*
```

This causes the build number and revision numbers to be created automatically to reflect the number of days since Jan 1st 2000, and the half the number of seconds since midnight respectively. This kind of versioning is very appropriate when an application is in development, but is often not appropriate for the final build of a particular version.

The versioning properties of Windows Installer files built with Visual Studio .NET 2003 Deployment projects are described in Table 4.1 on the next page.

**Table 4.1: Versioning Properties**

| Property | Description |
|---|---|
| **Version** | Represents the version of your Windows Installer file. Because the version property is used by Windows Installer to manage upgrades, you should change this property for each released version of your installer (unless you are creating a small update. When you change the **Version** property, Visual Studio prompts you that you should also update the **ProductCode** and **PackageCode** properties and can automatically update these two other properties for you. However, you will not always want to do so. |
| **ProductCode** | Specifies a unique identifier for an application, represented by a string GUID. You should update this property when you release a major upgrade version of your application; otherwise, it prevents the **DetectNewerInstalledVersion** and **RemovePreviousVersions** properties from working properly. Windows Installer uses the **ProductCode** to identify an application during subsequent installations or upgrades so no two major versions of an application should have the same **ProductCode**. To ensure a unique **ProductCode**, you should never manually edit the GUID — instead, you should use the GUID generation facilities in the **ProductCode** dialog box available from the Properties window. |
| **PackageCode** | The package code is a GUID identifying a particular Windows Installer package. The package code associates an .msi file with an application. The **PackageCode** property is not accessible in the Visual Studio .NET Properties window — instead it is created and managed when you build your setup project. This property manifests itself as the revision number of the file — you can view this revision number on the **Summary** tab of the properties sheet for your Windows Installer file in Windows Explorer. When you rebuild your project, Visual Studio .NET also changes the **PackageCode** property for you. |
| **UpgradeCode** | Specifies a unique identifier that is shared by multiple versions of the same application. The **UpgradeCode** should be set for the first version and should never be changed for subsequent versions of the application. Changing this property prevents the **DetectNewerInstalledVersion** and **RemovePreviousVersions** properties from working properly. (see Note) |
| **DetectNewerInstalledVersion** | Specifies whether to check for later versions of an application during installation on a target computer. The **UpgradeCode** and **ProductCode** are used to determine if a later version is present on the target computer. If this property is set to **True** and a later version number is detected at installation time, this version is not installed over the later version. |

| Property | Description |
|---|---|
| **RemovePreviousVersions** | Specifies whether an installer will remove earlier versions of an application during installation. If this property is set to **True** and an earlier version is detected at installation time, the earlier version's uninstall function is called. The installer checks **UpgradeCode** and **ProductCode** properties to determine whether the earlier version should be removed. The **UpgradeCode** must be the same for both versions, whereas the **ProductCode** must be different for the earlier version to be uninstalled. |

Exactly which of these properties you alter will depend to a large extent on the nature of the upgrade you are installing.

**Note:** If you modify the UpgradeCode from the value used to install the original version of your application; you cannot use the upgraded .msi file to upgrade that installation. Windows Installer service identifies and manages upgrades for installed applications by the UpgradeCode property. If you change the UpgradeCode in the upgraded Installer package, the Windows Installer service does not upgrade your existing application when you run it—instead, it installs a new application that is separate from the one already present. Consequently, properties such as DetectNewerInstalledVersion and RemovePreviousVersions have no effect.

#### Versioning Major Upgrades

Major upgrades are distinguished by a new version number, along with both a new **ProductCode** property and a new **PackageCode**. To create a major upgrade, you update the **Version** property for your Windows Installer file. Visual Studio .NET 2003 then prompts you that the **PackageCode** and **ProductCode** properties should also be updated and offers to do this for you. You should accept this offer if you want to create a major upgrade. However, as for all upgrade types, you should never update the **UpgradeProperty** of your Windows Installer file.

The reason for changing the **Version** property is to allow Windows Installer to differentiate between different versions of your application. This allows Windows Installer can do that, to manage the order in which updates can be applied. For example, if you create an upgrade to update version 1.0.0 of your solution to 2.0.0, and then you create another upgrade to update version 2.0.0 to version 3.0.0, Windows Installer can enforce the correct order by checking the version before applying the upgrades. You can also prevent the upgrade for updating version 2.0.0 to 3.0.0 from being applied to version 1.0.0.

The reason for changing the **ProductCode** is to allow Windows Installer to implement features such as automatically removing earlier versions of your application as you install the new version.

### Versioning Minor Upgrades

Minor upgrades will have a new version number. A minor upgrade requires that you change both the **Version** and **PackageCode** properties, but not the **ProductCode** or the **UpgradeCode**. When you change the **Version** property, Visual Studio .NET prompts you that you should also change the **PackageCode** *and* **ProductCode** and offers to do so for you. You should decline this offer. The **PackageCode** will still be updated when you rebuild the solution, but the **ProductCode** (and **UpgradeCode**) properties will remain unchanged.

---

**Note:** It might seem sensible that changing the version from 1.0 to 1.1 constitutes a minor upgrade whereas changing from 1.0 to 2.0 represents a major upgrade. However, while you will often choose your version numbers to reflect this numbering scheme, the elements of the Version property you change are irrelevant to the type of upgrade you are supplying — the difference between a minor and major upgrade is actually whether the ProductCode property changes as well the Version.

---

Because the **ProductCode** property is not changed, you must adhere to some strict requirements if you are to provide a minor upgrade. For a list of these requirements, see "Changing the Product Code" on MSDN.

Also, because the **ProductCode** will not have changed for a minor upgrade, you cannot simply run the upgraded Windows Installer file to update your application. If you attempt this, the Windows Installer service detects that the product has already been installed and displays the message "Another version of this product is already installed. Installation of this version cannot continue. To configure or remove the existing version of this product, use Add/Remove Programs on the Control Panel." However, if you run your Windows Installer file from the command line, and provide certain switches, you do not need to use the **Add/Remove Programs** utility to uninstall the application before running your upgrade. The following is an example of a typical command line that a user (or deployment tool such as SMS) needs to specify to run a minor upgrade without first uninstalling the application:

```
Msiexec /i d:\MyApp.msi REINSTALL=ALL REINSTALLMODE=vomus
```

Effectively, this command line reinstalls the existing product, rather than treating it as a major upgrade (because the **ProductCode** is the same). Crucially, the reinstall is performed with your new Windows Installer package, rather than the original one that will have been cached by the Windows Installer service. The "v" in the **vomus** switch specifies that the installation should be run from your new source, rather than from the cached Windows Installer file that was originally used to install the application, and it also specifies that your new Windows Installer file should replace the original installer in the cache.

For more information about the command-line switches, see the following topics in the Windows Installer Platform SDK:

- REINSTALL property
- REINSTALLMODE Property

#### Versioning Small Updates

To create a small update, you change the **PackageCode** property of the installer, but not the **ProductCode**, the **Version**, or the **UpgradeCode** properties. You can create a small update by rebuilding a modified Windows Installer file using Visual Studio .NET. Rebuilding an .msi file with Visual Studio .NET automatically creates a new **PackageCode** for the installer, regardless of whether you have changed the **Version** or **ProductCode** properties.

Because small updates do not change the **Version** or the **ProductCode** properties of an installation, you can use them only if your modifications comply with some strict requirements. For a list of these requirements, see "Changing the Product Code" on MSDN.

As with minor upgrades, you cannot simply run a small update and have it replace your existing application. This is, again, because the **ProductCode** has not changed and Windows Installer detects that the product has already been installed. As for minor upgrades, Windows Installer displays the message "Another version of this product is already installed. Installation of this version cannot continue. To configure or remove the existing version of this product, use Add/Remove Programs on the Control Panel." Again, as for minor upgrades, you can reinstall the application from the command line, using the following syntax:

```
Msiexec /i d:\MyApp.msi REINSTALL=ALL REINSTALLMODE=vomus
```

There will undoubtedly be many situations where small updates will *not* suit your needs. For example, you may want to develop a series of upgrades that can be applied in sequence (which cannot be achieved with small updates because the **Version** property will not have been changed). In this case, you should consider creating either a major or minor upgrade for your application. Furthermore, if you have made modifications that do not comply with the requirements for retaining the **ProductCode** property, you cannot create a small update—you should instead create a major upgrade for your solution.

### Upgrading Merge Modules

Unlike Windows Installer files, merge modules (.msm files) are not installed on their own. Instead, they are included in Windows Installer files. The components and setup logic you add to your merge module are incorporated with that of the Windows Installer file when the installer is built. Therefore, your merge modules do not interact directly with the Windows Installer service.

If you modify the components you have included in a merge module, you need to:

- Update and rebuild the .msm file.
- Make the new .msm file available to developers for inclusion in their .msi files.

The developers who use your merge module then need to replace the previous version with your new .msm file. Because their application has essentially been modified by the inclusion of your new .msm file, they will need to rebuild and redistribute their installer to the production environment in order for changes to take effect. Depending on the nature of the change, the developer should then decide whether this is a major upgrade, minor upgrade, or small update and you will need to update the properties for the Windows Installer package appropriately.

Merge modules have their own versioning properties, independent from .msi files. These are:

- **ModuleSignature**. Each released version of a merge module must have a unique **ModuleSignature** in order to avoid versioning problems when they are added to .msi files. You should never manually edit the GUID portion of this property. Instead you should use the GUID generation facilities in the **Module Signature** dialog box, accessible from the Properties window in Visual Studio .NET.
- **Version**. Each time you update and rebuild your merge module, you should update the **Version** property.

The **Version** property is useful for allowing the developer to distinguish between different releases of your merge module. The **ModuleSignature** property is used by Visual Studio and other installer creation tools to uniquely identify the merge modules when they are incorporated in .msi files.

### Upgrading CAB Files

CAB files are used to package .NET managed controls (or ActiveX controls). CAB files are typically referenced from a Web page. They are automatically downloaded, and their controls extracted and installed, by Web browsers. As such, they are not installed as applications with the Windows Installer service. When you update the controls that have been distributed in this way, you will need to rebuild your CAB file. Like .msi and .msm files, CAB files support the **Version** property. To avoid version conflict, you should update this property for every new release of your CAB.

### Distributing Upgrades

As with initial deployment, there are a number of ways in which you can distribute your upgrades. These are:

- Group Policy functionality of Active Directory
- Systems Management Server (SMS)
- Other Methods (including placing files on a network server, Web server or distributing media)

We will discuss each of these in turn:

#### Group Policy Functionality of Active Directory

If you originally deployed your application using a Windows Installer file and are publishing or assigning software to users or computers using Group Policy, you can redistribute your upgraded application using a similar approach. You can upgrade your applications by:

- Creating a new msi for your application, modifying an existing msi, or applying a patch package to an existing msi.
- Using Group Policy Software Installation to specify that your new package is an upgrade to the existing package.
- Choose whether the existing package should be uninstalled, or whether the upgrade package can upgrade over the existing package. Typically, the uninstall option is for replacing an application with a completely different one. The upgrade option is for installing a later version of the same product while retaining the user's application preferences, document type associations, and so on.
- Choosing whether or not the upgrade is to be mandatory.

For more information about deploying and updating applications with Group Policy, see the Step-by-Step Guide to Software Installation and Maintenance.

#### Systems Management Server (SMS 2.0)

If you originally deployed your application using a Windows Installer file by using SMS, you can redistribute your upgraded application using the same mechanism.

To upgrade a Windows Installer application, you need to complete two processes:

1. Create a new image for your application, either by patching the original source files, or by creating a new software package. Then update the package's distribution points to propagate the new version of the source files. Any clients installing the application after you upgrade the source files and update the package's distribution points will install the upgraded application.
2. Upgrade existing clients by creating a new program in the existing package. Then, send an advertisement for this new program to a collection of clients that have installed the original version of the application.

For more information about deploying and updating applications with SMS, refer to the SMS product documentation and the white paper, "Deploying Windows Installer Setup Packages with Systems Management Server 2.0."

### Other Methods

There are a number of other methods that are used to distribute upgrades in the form of Windows Installer packages. The following table outlines their uses:

**Table 4.2: Methods of Distributing Windows Installer Packages**

| Method | Description | Comments |
|---|---|---|
| Web/FTP/Network Server | New Installer package (or a patch) is placed on a server and the link sent out to users so they can download it. | If you are creating a fresh or updating an existing msi, you can replace the old files with the new ones, or create a fresh share for the new installation.<br>If you are using a patch package, you can direct the users to a share containing the patch, or in the case of an administrative install, apply the patch directly to the application by using the msiexec command line tool. This will ensure that anyone installing the application later will also be installing the updates as well. |
| E-mail | New Installer package or patch package sent using e-mail. | Sending the package or patch by e-mail is not recommended since this increases the workload on your e-mail system and security filters may not allow you to pass certain types of files using e-mail. Instead you should send a link to a location where the user can apply the patch or install the new msi. |
| CD/DVD | New Installer package or patch burned to CD/DVD. | May be the most appropriate method where bandwidth is low. |

## Updating a Simple Collection of Build Outputs

If you distributed your application to the production environment as a collection of build outputs and other files, you can generally upgrade your solutions by copying the new build outputs or files to the appropriate locations. However there are some issues you should be aware of when updating a collection of build outputs.

### Using Shadow Copy Functionality

The .NET Framework supports s*hadow copying*, a process in which .NET copies each assembly into a temporary shadow copy directory before opening the assembly (.dll or .exe) file. This mechanism gives you the ability to recompile or to re-deploy new versions of assemblies while the application is in use. The new versions of your assemblies will be used the next time the user starts the application.

In ASP.NET applications, all that happens automatically—the ASP .NET worker process does the work for you. So to update a Web application you can simply copy

the updated files to the correct location, and ASP.NET automatically handles shutting down the running application and copying the shadow files. (It's only that seamless as long as your application doesn't use any assemblies requiring registration for COM interoperability, or assemblies that must be registered in the global assembly cache.)

### Updating Assemblies

You can normally update private assemblies by simply copying the new version of the assembly over the old one. However, although you can use simple copy operations for the initial deployment of a strong-named assembly, it is not possible to update it in this way and have your application (or other assemblies) use it automatically. The strong name of the assembly is stored in the manifest of any assembly that references it, and different versions of a strong-named assembly are considered to be completely separate assemblies by the CLR. Unless you specify otherwise, the CLR does not load a different version of a strong-named assembly than the one your application was originally built against. For more details on how to update strong named assemblies, see Running Assemblies Side by Side later in this chapter.

### Distribution Strategies for an Upgrade to a Collection of Build Outputs

Although you may have to consider the update of strong named assemblies separately, for the most part, upgrading a collection of build outputs can be achieved in the same way that they are initially deployed. You can copy the new files to their destination folder using any of the following mechanisms:

- Using Microsoft Application Center 2000 (for Web farm deployment).
- Copy Project functionality of Microsoft Visual Studio .NET 2003 (only suitable for Web-based applications)
- File copy distribution

We will discuss each of these in turn:

#### Application Center

Application Center provides broad support for updating your server environments. You can update content, configuration settings, and COM+ applications on the cluster controller, and then have Application Center synchronize the other members of your cluster with the controller.

For ease of administration, you can have Application Center automatically synchronize both content and configuration settings on the members of your cluster. You can use either of the two following automatic synchronization methods:

- **Change-based synchronization**. Application Center detects when content or configuration settings have changed on the cluster controller, and automatically replicates those changes to the other members of the cluster immediately. COM+ applications are not automatically updated.

- **Interval-based synchronization**. You can set a synchronization interval that governs when Application Center replicates changes that have occurred on the cluster controller to the other members of the cluster. The default interval is 60 minutes.

If you need a finer level of control over the synchronization of your clustered server environment, you can use the manual synchronization features of Application Center. There are three different types of manual synchronization:

- **Cluster synchronization**. All cluster members in the synchronization loop are synchronized with the controller and all applications are synchronized (except for COM+ applications).
- **Member synchronization**. Only the specified cluster member is synchronized with the controller. All applications (except for COM+ applications) are synchronized.
- **Application synchronization**. A specified application on all cluster members in the synchronization loop is synchronized with the cluster controller.

For more information about synchronization and synchronization loops, see the Application Center documentation.

#### Updating COM+ Applications with Application Center

When you update COM+ applications or COM components, you need to stop any process that is currently using those components. This is because processes lock components they are using. After you update your COM component or COM+ application, you typically restart the process that you previously stopped, so that it can it resume its operations and take advantage of your updated functionality. In the Web environment, this typically means stopping and restarting the IIS Web service when you update COM/COM+ applications. These steps need to be taken for all COM/COM+ applications, regardless of how you have updated them.

You can use Application Center to ease the process of stopping and restarting services when you update COM+ applications. Application Center deploys COM+ applications to member servers when they are initially added to the cluster. However, rather than automatically synchronizing COM+ applications across your cluster when subsequent changes are made, Application Center allows you to defer this synchronization to off-peak periods. By deferring the synchronization of COM+ applications, you can minimize the effect of stopping and restarting the services on your cluster members.

To upgrade COM+ applications with Application Center, you can deploy them using the New Deployment Wizard or with Application Center command line operations. You should schedule this deployment for off-peak periods, because the Web service will be stopped and started on the cluster members, effectively leaving only the cluster controller available to respond to requests while synchronization takes place.

For more information about deploying and synchronizing COM+ applications, see the Application Center documentation.

#### Visual Studio 2003

If you are upgrading a Web application, you can use the Copy Project command in Visual Studio .NET 2003 to update the Web application on a target server. You may choose to modify the location of the files so that the new application can exist alongside the old, or you may wish to copy the files over the original location. Copy Project will not remove old files from the target directory, so you may need to do some manual cleanup after using the Copy Project command. For files that you are overwriting, you will be prompted to choose to overwrite them or not.

#### File Copy

In many cases where the updates to an application are relatively simple, file copy can be one of the most effective approaches for updating your application. If this is the case, deploying an update is simply a case of copying over the new files and removing any old files that are no longer required. For more details on using file copy as a deployment method, see Chapters 2 and 3 of this guide.

## Configuring Automatic Updates

One alternative to the various approaches for patching, repackaging, and updating applications is to move the responsibility of updating the application from administrators or users to the application itself. In this case, the client application can be designed to automatically download and install updates from a server. To achieve this, a developer can include code with an application so that it:

- Automatically checks for updates.
- Downloads updates if available.
- Upgrades itself by applying those updates.

In order to manage updates in this way, the automatic updater needs to know:

- **When to check for updates**. This can be defined by creating a thread that polls the server periodically for updates, on a time basis or when an event occurs.
- **Where to check for updates**. This can be defined by adding a path directly in the application or referring to a Web service that specifies the path.
- **How to check for updates**. The developer can force the application to check timestamps that have changed, or refer to a Web service that stores the files that are changed.
- **How to apply updates**. This is the real challenge of automatically updating applications. The application needs to update its own files while it is running—but those files will be locked *because* the application is running. The only way to unlock the files is to stop the application, but if you stop the application, it cannot perform the update.

## The Application Updater Application Block

Designing your applications for automatic updates is a potentially complex area, so to help you configure applications for automatic updates, Microsoft has created some sample code that you can use as a basis for designing your own automatic updates solution. The Application Updater Application Block is an evolution of the MSDN AppUpdater.

There are a number of solutions to the problem of defining how the application should be updated. The Application Updater Application Block assumes you will follow one of two strategies:

- Using a Windows service.
- Using a shim or façade EXE with self-update capability built into the application itself.

We will discuss each strategy in turn:

### Using a Windows Service

This method is suitable in situations where you are responsible for updating more than one application, as a single service can be responsible for maintaining multiple applications. It is also appropriate for updating other Windows services or nonclient applications, as it functions independently of the applications themselves.

This method is useful if you want to update functionality within an application, but without the application being aware of any changes. You can use this method for updating even nonclient applications, for example, Windows services.

The downside to this method is that you have to set up and maintain a service. Services need to be registered, so a simple file copy is not appropriate to install them. For more details on how to achieve this, see Chapter 3, "Implementing the Deployment of .NET Framework-based Applications." The Windows services method is not recommended if you are sending a smart client application to unknown customers over the Internet (for example, MSN Messenger). In this scenario the application itself is unable to determine that updates are available, and security considerations are likely to prevent you being able to push that information out to the client.

The following diagram shows how the Windows service is responsible for updating applications:

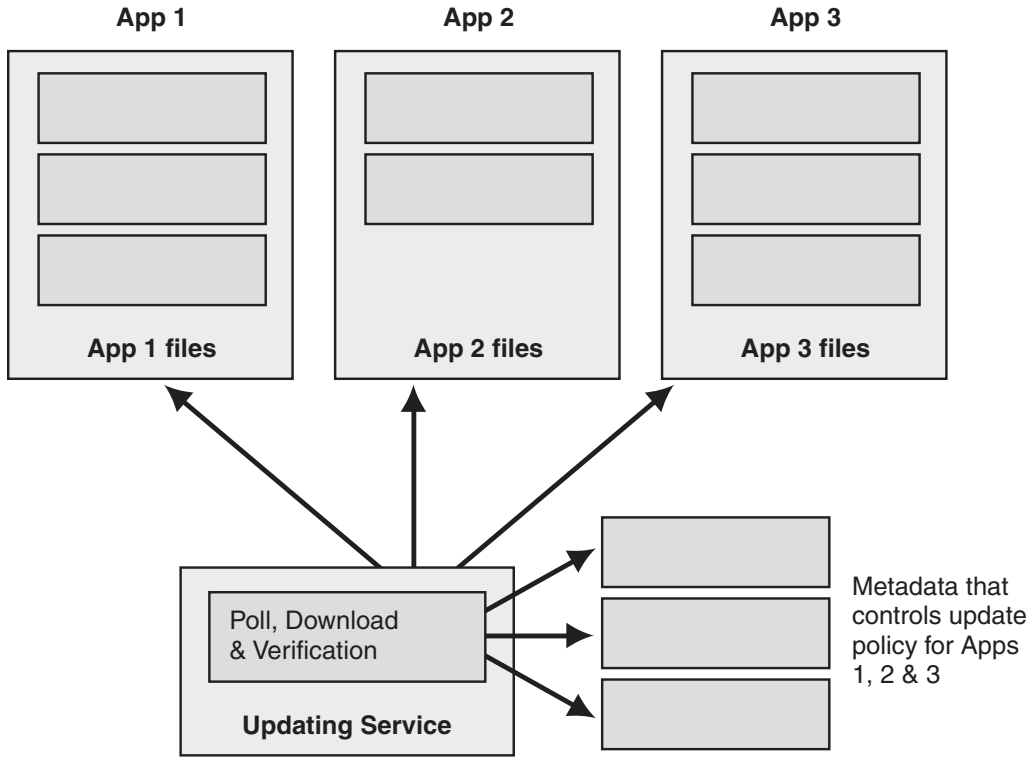


**Figure 4.1**
*Using a Windows service to update applications*

#### Using a Façade Executable

In this case, the user actually launches the façade executable, which is responsible for launching the real application, and relaunching it after updates have occurred. The façade application is much easier to deploy than a Windows service and can generally be done just by using a file copy method. In this scenario the application itself is responsible for obtaining the updates. As key files will often be locked the application generally does need to be shut down and restarted by the façade application.

This method gives you full control over the update of the application, including all functionality with the potential for integrating the polling of updates and notifications, and downloading updates in the application itself. However, unlike the case of a Windows service, a restart of the application is needed to complete the update.

The following diagram shows upgrade functionality built into the application.

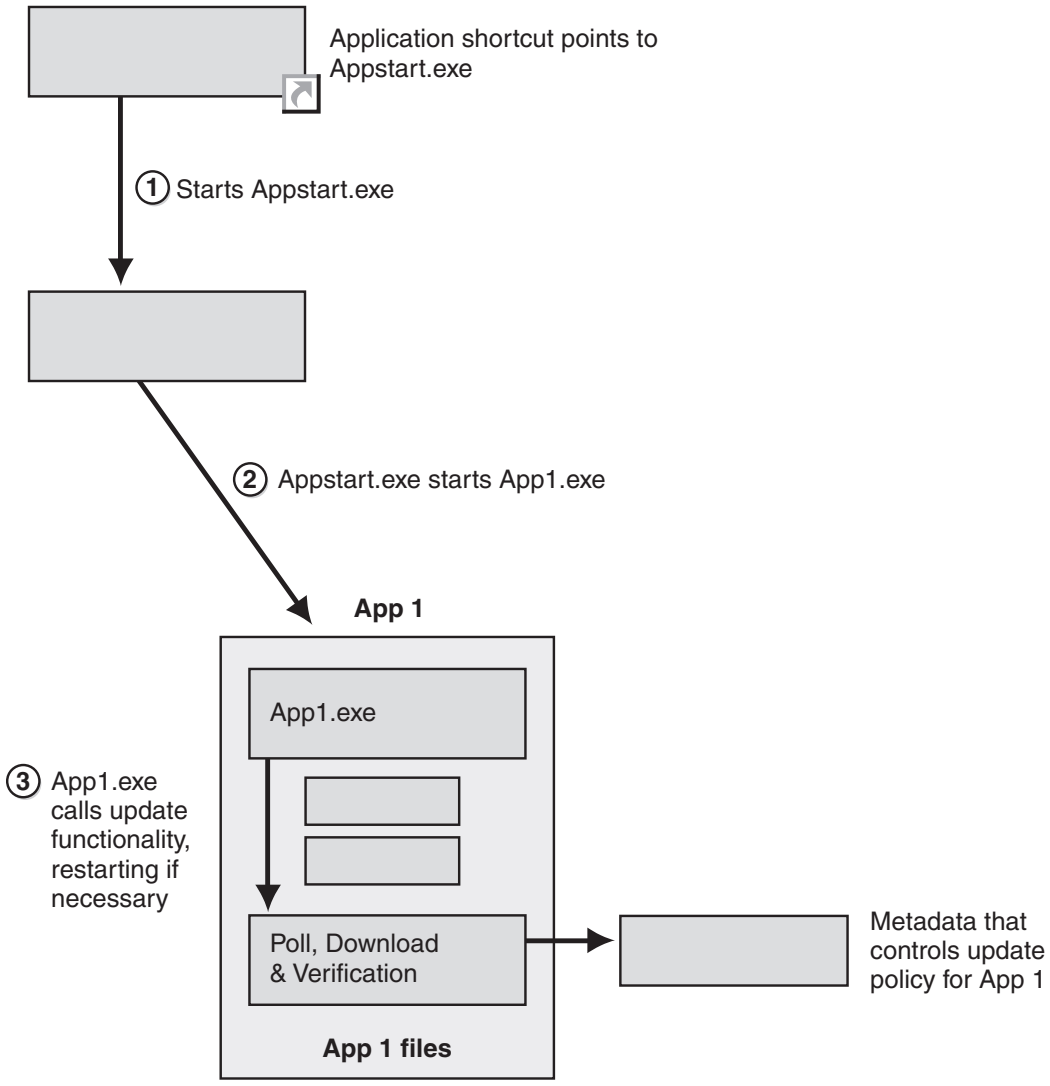


**Figure 4.2**
*Using a façade application*

**Note:** For more information, see the white paper ".NET Applications: .NET Application Updater Component" on the windowsforms.net site. This white paper provides sample code for a component that you can incorporate into your own applications. You will need to set a few properties, such as the location to check for updates, but apart from those minor modifications, you can actually use the sample component as is.

## Other Solutions for Automatic Updates

You may choose your own solution to the problem of configuring automatic updates, sometimes using the Application Updater Application Block as a starting point. For example, you may wish to produce a solution that does not involve having to restart the application post updates. Two possible solutions to this include:

- Using a façade application to perform the checks for updates.
- Using a shadow copy technique for the application update.

We will look at each of these in turn:

### Using a Façade Application to Perform the Checks for Updates

In this case, the façade application checks for updates before launching the true application. There is no file locking as the application has not yet been launched, and so once updates are complete, the true application can be run. If the façade application also checks for updates when the application is running—the application would still need to be restarted to resolve any file locks.

This solution is particularly useful when you cannot modify the source code of the application itself and do not wish to use a Windows service to perform the updates.

The diagram shows how the façade application can be used to check for updates.

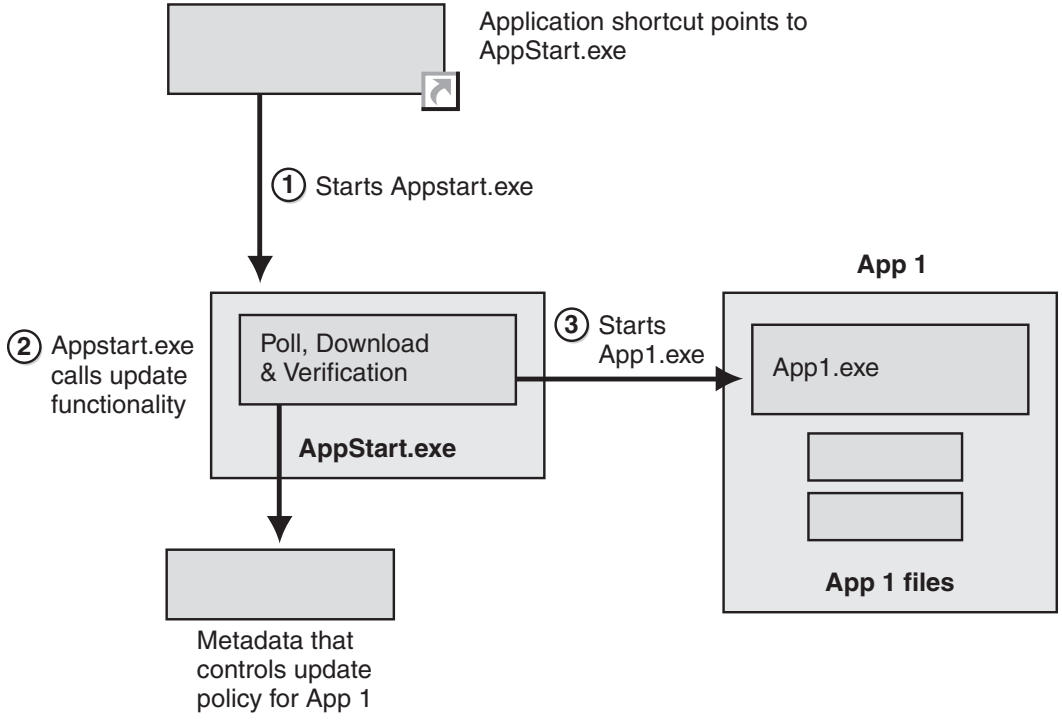


**Figure 4.3**
*Using a façade application to check for updates*

### Using a Shadow Copy Technique for the Application Update

If you get the application to shadow copy the appropriate files and use those copies, the originals can be updated without needing to restart the application.

The main barrier to using this method is one of complexity. However it does provide the ability to do updates on the fly with little or no notice to the user in most cases.

The following diagram shows how shadow copy functionality can help prevent the application from being restarted.

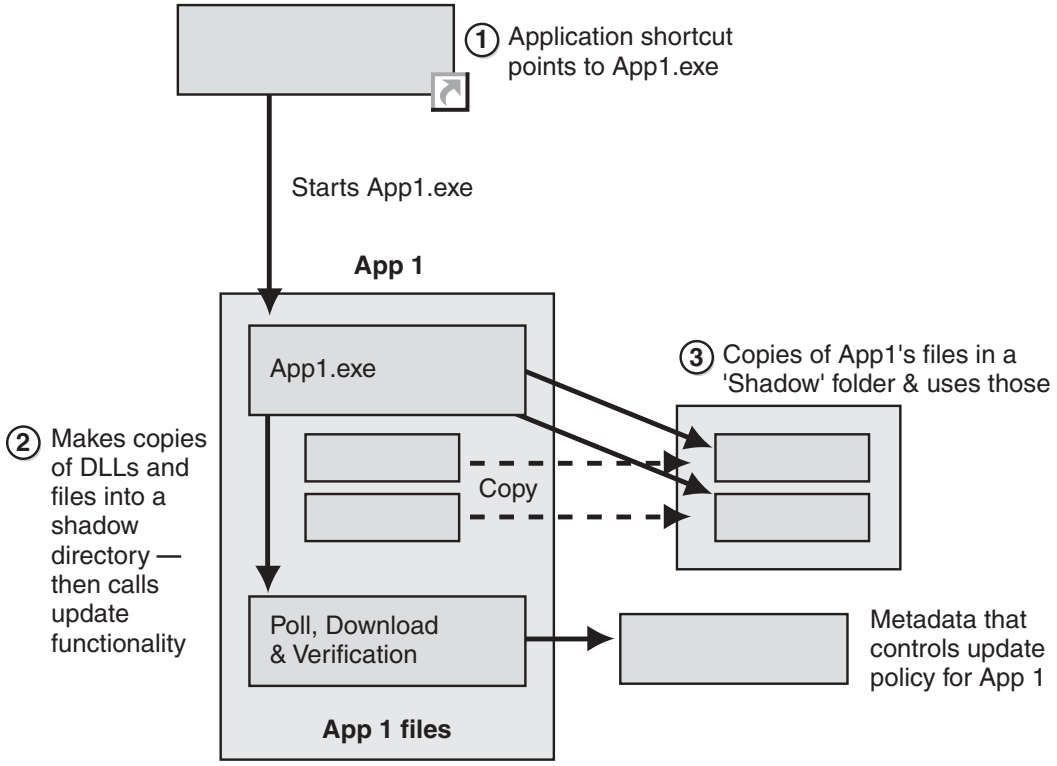


**Figure 4.4**
*Using shadow copy functionality to keep the application from being restarted*

## .NET Framework Side-by-Side Execution

So far in this guide we have looked at the relatively simple situation where applications are installed in isolation. There is no assumption that other applications will already exist on computers. However, in many cases you are likely to have multiple .NET-based applications running alongside each other. Those applications may

share assemblies, and in some cases you may find that they require different versions of the same assembly in order to run properly. Not only that, but applications may even be build on different versions of the Framework and may not function properly using another runtime.

Fortunately, one of the major benefits of the Framework is the ability to run multiple applications, using multiple versions of the same components, requiring multiple versions of the runtime itself. This functionality is known collectively as side-by-side execution.

The following illustration shows several applications using two different versions of the runtime on the same computer. Applications A, B, and C use runtime version 1.0, while application D uses runtime version 1.1.

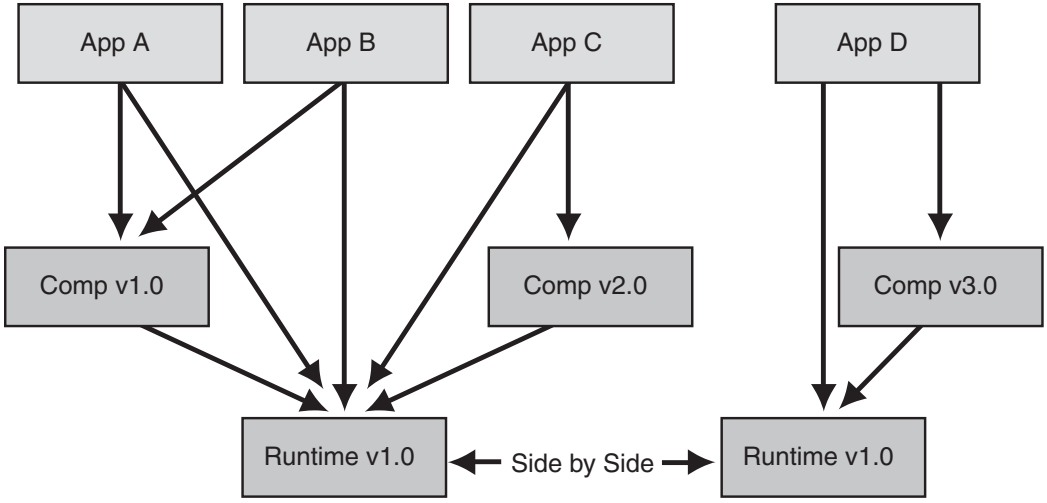


**Figure 4.5**
*Side-by-side execution of two versions of the runtime*

The .NET Framework consists of the common language runtime and about two dozen assemblies that contain the API types. The runtime and the .NET Framework assemblies may have different versions. For example, version 1.0 of the runtime is actually version 1.0.3705.0, while version 1.0 of the .NET Framework assemblies is version 1.0.3300.0.

## Running Assemblies Side by Side

Prior to Windows XP and the .NET Framework, DLL conflicts occurred because applications were unable to distinguish between incompatible versions of the same code. Type information contained in a DLL was bound only to a file name. An application had no way of knowing if the types contained in a DLL were the same types that the application was built with. As a result, a new version of a component

would often overwrite an older version and break applications. Side-by-side execution of the .NET Framework provides a number of features to eliminate DLL conflicts.

## Determining Which Assembly to Load

In order to understand how assemblies can exist side by side, it is important to know how the CLR determines which assembly to load. This will depend on a whole series of factors, including whether the assembly is strong named, how the assembly is referenced in the application which calls it, and where the assembly is located.

For detailed information on how assemblies are loaded, see "How the Runtime Locates Assemblies" on MSDN.

There are a number of ways in which you can notify the CLR to load an updated version of a strong named assembly. These include:

- Updating the application configuration file
- Use a publisher policy
- Updating the machine configuration file

We will discuss each of these in turn:

### Application Configuration File

You can instruct the CLR to load an updated version by providing binding redirects in your application's configuration file. You must then deploy not only the updated strong-named assembly, but also the updated application configuration file that contains the binding redirect information. The following code snippet shows an example of redirecting your application to use an updated assembly:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="myAssembly" publicKeyToken="32ab4ba45e0a69a1"
                                      culture="en-us" />
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

**Note:** The CLR attempts to locate assemblies in the global assembly cache before searching in the application base folder. Consequently, if you deploy a strong-named private assembly with your application, but it is already present in the global assembly cache, the CLR uses the shared version, rather than the private one, at run time. This should not lead to any problems, because assemblies with the same strong name should always be identical.

#### Publisher Policy

You can state that applications should use a newer version of an assembly by supplying a publisher policy file with the upgraded assembly. The publisher policy, which contains assembly redirections, is itself located in an assembly, and placed in the global assembly cache. If a publisher policy file exists, the runtime checks this file after checking the assembly's manifest and application configuration file. You should use publisher policies only when the new assembly is backward compatible with the assembly being redirected.

---

**Note:** New versions of assemblies that claim to be backward compatible can still break specific applications. When this happens, you can use the following setting in the application configuration file to make the runtime bypass the publisher policy: <publisherPolicy apply="no">.

---

You can create a publisher policy file to specify assembly redirection for upgraded versions of strong-named assemblies. A publisher policy file is an XML document that is similar to an application configuration file. You need to compile it into a publisher policy assembly and place it in the global assembly cache for it to take effect.

► **To create and deploy a publisher policy**

1. Create a publisher policy file.
2. Create a publisher policy assembly from your publisher policy file, using the Assembly Linker utility (AL.exe).
3. Add your strong-named shared assembly to the appropriate location.
4. Add the publisher policy assembly to the global assembly cache.

For more information about publisher policy files, see Creating a Publisher Policy File on MSDN:

#### Machine Configuration File

You can also redirect assembly versions with the machine configuration file. Unlike application settings, when determining binding redirects, the machine configuration file is checked after the application configuration file. This means that the machine configuration file overrides all of the individual application configuration files and publisher policies, and applies to all applications. As a result you do not often use the machine configuration file to store this information. However, there might be some cases when you want all of the applications on a computer to use a specific version of an assembly. For example, you might want every application to use a particular assembly version because it fixes a security hole. If an assembly is redirected in the machine configuration file, all of the applications using the earlier version will use the later version.

For more information about redirecting assembly binding, see "Redirecting Assembly Versions" on MSDN.

## Isolation

Using the .NET Framework, you can create applications and components that execute in isolation, an essential component of side-by-side execution. Isolation involves being aware of the resources you are using and safely sharing resources between multiple versions of an application or component. Isolation also includes storing files in a version-specific way.

### Assemblies in the Global Assembly Cache

Placing assemblies in the global assembly cache produces some significant benefits for side-by-side operations. These include:

- The global assembly cache supports side by side installation of multiple versions of a single component. Assemblies installed in an application directory can not support side by side since the directory can only contain one file of a given name and all versions of a component share the same file name.
- You can control the precise binding semantics used for code in the global assembly cache.
- Individual applications can choose to bind to any version of an assembly in the global assembly cache. Multiple applications can each bind to different versions of the same assembly in the global assembly cache.

After an assembly is installed into the global assembly cache, you cannot simply copy a new version and have your application use that updated assembly. As with all strong-named assemblies, applications contain the strong name (complete with version number) of the assembly that they reference in their own manifests. Instead of simply copying a new version of the strong-named assembly, you can either recompile against the newer version or provide binding redirection for the referencing application as well.

#### Version-Aware Code Storage

The .NET Framework provides version-aware code storage in the global assembly cache. The global assembly cache is a computer-wide code cache present on all computers with the .NET Framework installed. It stores assemblies based on version, culture, and publisher information, and supports multiple versions of components and applications residing in the same location. Unless specified elsewhere, each application will use the version of the assembly it used at compile time.

#### Testing Considerations for Side-by-Side Shared Assemblies

Although the Framework is designed to allow assemblies to run side by side without problems, whether they can do so or not will depend in part on how the assemblies were designed. Therefore there are a number of areas you should test to show that you are able to deploy the assemblies successfully side by side.

You should ensure that your installs (and uninstalls) work as expected. For example, if a registry key is created during the install of an assembly, and a second version of that assembly is installed that requires the same registry key, you need to determine how install and uninstall routines should behave with respect to that registry key. For this example, you want to ensure that the registry key is not removed if either assembly version is uninstalled, because this will break the remaining version.

You will also need to ensure that side-by-side execution works properly. For example, if multiple versions of an assembly rely on a registry key or any shared resource (such as a file, directory, and so on) and both versions are executing side by side, you will need to determine how the assemblies affect each other. You need to ensure that changes to shared resources by one version of the assembly will not adversely affect the other version(s).

Furthermore, if more than one version of an assembly is used by a process, you will need to determine if there are type issues. For example, let's say that you have an assembly which has had both version 1 and 2 installed into the global assembly cache. Now imagine that two controls are used in one application, and that one control references version 1 and the other control references version 2 of your assembly. You need to determine whether the assemblies can be used together in the same application in this way. The types in the different assemblies are considered to be different by the common language runtime, and so cannot be exchanged.

## Running Distributed Applications Side by Side

If your application is designed to run across multiple physical computers, versioning issues are complicated as you may have different versions of the Framework installed on different computers. It is therefore very important to thoroughly test any updates to your application on a distributed environment that matches your production environment.

For More Information on side by side issues for Distributed applications, see "Side-by-Side and Versioning Considerations for .NET Remoting" on MSDN.

## Running Multiple Versions of the Framework Side by Side

Versions 1.0 and 1.1 of the .NET Framework are designed to be compatible. A properly configured application built with the .NET Framework version 1.0 should run on version 1.1, and a properly configured application built with the .NET Framework version 1.1 can be configured to work with version 1.0 of the Framework, and should function with no problems, unless it refers to API features that were added in version 1.1.

---

**Note:** For more information on compatibility changes to the .NET Framework see, "Backwards Breaking Changes from version 1.0 to 1.1" on the GotDotNet Web site.

---

Versions of the .NET Framework are treated as a single unit consisting of the runtime and its associated .NET Framework assemblies (a concept referred to as assembly unification). You can redirect assembly binding to include other versions of the .NET Framework assemblies, but overriding the default assembly binding can be risky and must be rigorously tested before deployment.

### Determining Which Version of the Runtime to Load

The runtime determines the available runtime versions by enumerating the keys and values in the registry under **HKLM\SOFTWARE\Microsoft\.NETFramework\policy**. Each key identifies the major and minor version of the runtime. The values under each major and minor key identify the build number. For example, the key **HKLM\SOFTWARE\Microsoft\.NETFramework\policy\v1.1** with a value of 4322 indicates that version 1.1.4322 of the .NET Framework is installed.

A directory with the same version number as specified in the registry must also exist under the .NET Framework installation root. The directory names are preceded with the letter v. For example, version 1.1.4322 of the .NET Framework would be installed in <WinDir>\ Microsoft.NET\Framework\v1.1.4322\ where <WinDir> is the Windows directory.

The following are the .NET Framework versions currently available:

- Version 1.0 is v1.0.3705
- Version 1.1 is v1.1.4322

The runtime uses the both the application configuration file and the PE file header to determine which versions of the runtime are supported by the application. Assuming an application configuration file is present, the following process is used to determine the appropriate runtime version to load:

1. The runtime examines the **<supportedRuntime>** element in the application configuration file. If one or more of the supported runtime versions specified in the **<supportedRuntime>** element are present, the runtime loads the runtime version specified by the first **<supportedRuntime>** element. If this version is not available, the runtime examines the next **<supportedRuntime>** element and attempts to load the runtime version specified. If this runtime version is not available, subsequent **<supportedRuntime>** elements are examined. If none of the supported runtime versions are available, the runtime fails to load a runtime version and displays a message to the user (see step 4). See the .NET Framework General Reference on this topic for more information.

2. If no **<supportedRuntime>** element is present, the runtime examines the **<requiredRuntime>** element in the application configuration file. This element is used only for runtime version 1.0 applications. If the runtime version specified by the **<requiredRuntime>** element is present, the runtime loads it. If the specified version is not available, the runtime fails to load a runtime version and displays a message to the user (see step 4). If the application configuration file has no **<requiredRuntime>** element, the process continues to step 3. See the .NET Framework General Reference on this topic for more information.
3. The runtime reads the PE file header of the application's executable file. If the runtime version specified by the PE file header is available, the runtime loads that version. If the runtime version specified is not available, the runtime searches for a runtime version determined by Microsoft to be a suitable replacement for the runtime version in the PE header. If that version is not found, the process continues to step 4.
4. The runtime displays a message stating that the runtime version supported by the application is unavailable. The runtime is not loaded. If this process is conducted for a service or some other event that does not involve user interaction, and you have set the **HKLM\Software\Microsoft\.NETFramework\NoGuiFromShim** registry key to 1, then the message is written to the Event Log.

---

**Note:** After a runtime version is loaded, assembly binding redirects can specify that a different version of an individual .NET Framework assembly be loaded. These binding redirects affect only the specific assembly that is redirected.

---

## Running .NET-based Applications with Dependencies

In some cases you may find that you have an application compiled under one version of the Framework, but a dependent dll compiled under another. Of course this is not necessary, as it is possible to run multiple versions of DLLs on the same computer, and in the case of the global assembly cache, the same location. Add to that the fact that unless you specify otherwise the DLL used will be the one used at compile time, and you will see that having applications where there is this kind of discrepancy is fairly unusual.

However there are circumstances in which this kind of situation could arise. For example, an old version of a DLL could be removed for security reasons and existing applications simply reconfigured to use the new one. Under these circumstances the application determines which version of the Framework is run (only one version can run at a time for a particular application) and the dependent DLL has to attempt to run under that version.

The following table shows the behavior of a .NET-based application with a dependent DLL where different versions of the application, component, and Framework are available.

**Table 4.3: Side by Side Behavior of a Window Forms-based Application**

| Application Version | Only .NET Framework v1.0 installed | Only .NET Framework v1.1 installed | Both versions of the Framework installed |
|---|---|---|---|
| Application v1.0 Component v1.0 | No issues with compatibility | It will attempt to run with the v1.1 CLR[1] | Will run with v1.0 of the Framework |
| Application v1.0 Component v1.1 | Will run under the v1.0 CLR[2] | It will attempt to run with the v1.1 CLR[1] | Will run with v1.0 of the Framework |
| Application v1.1 Component v1.0 | Will run under the v1.0 CLR[3] | It will attempt to run with the v1.1 CLR[1] | Will run with v1.1 of the Framework[1] |
| Application v1.1 Component v1.1 | Will run under the v1.0 CLR[3] | No issues with compatibility | Will run with v1.1 of the Framework |

## Configuring a COM Application for Side-by-Side Execution

Application configuration files enable a COM application to bind to a specific managed component and specify which version of the runtime runs the component. COM application developers can create a .NET-based application configuration file and deploy it with their applications.

Managed and unmanaged applications use the identical configuration file schema to specify a version of the runtime and to bind to a specific component.

### Specifying the Runtime Version

If a COM application calls managed code without an application configuration file, the latest compatible runtime version installed on the computer is loaded by default. If this behavior does not satisfy the requirements of your COM application, you should indicate in a configuration file specific runtime versions that your application requires. For example, you can specify runtime version 1.1.4322, which loads the .NET Framework version 1.1.

1. Could break if any of the components are using functionality that has been broken in the new version of the Framework. For more information on the list of break points in the functionality, please see the link at the end of this chapter.
2. Could break if the components are using v1.1 APIs as they are not supported in the earlier version.
3. This is true if the **<supportedRuntime>** element is used, otherwise it will send an error.

► **To specify runtime version 1.1.4322**

1. Using an XML editor, create an application configuration file.
2. Insert the following standard header at the beginning of the file:

```
<?xml version ="1.0" encoding = "utf-8">
Insert the following XML elements into the file:
<configuration>
   <startup>
      <supportedRuntime version="v1.1.4322"/>
   </startup>
</configuration>
```

**Note:** COM components hosted by an extensible host, such as Microsoft Internet Explorer or Microsoft Office cannot control which version of the runtime is loaded. However, you can create an application configuration file for the host application which will determine which version of the runtime is used.

#### Specifying an Assembly Version

If a COM application calls a .NET assembly without using an application configuration file, the runtime loads the latest version of the assembly registered in the Windows registry that contains the type to be activated from COM. You can override this behavior by directing your application to bind to an earlier assembly version.

► **To redirect assembly binding to an earlier version**

1. Using an XML editor, create an application configuration file.
2. Insert the following standard header at the beginning of the file:

```
<?xml version="1.0" encoding="utf-8" ?>
```

3. Insert the following XML elements into the file. The **<bindingRedirect>** element with the **oldVersion** and **newVersion** attributes redirects binding for **myManagedAssembly** version 2.0 to version 1.0. For more information on the <bindingRedirect> element, see the .NET Framework General Reference.

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <dependentAssembly>
            <assemblyIdentity name="myManagedAssembly"
                              publicKeyToken="32ab4ba45e0a69a1"
                              culture="en-us" />
            <bindingRedirect oldVersion="2.0.0.0"
                             newVersion="1.0.0.0"/>
         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

4. You can redirect more than one assembly version by including multiple **<bindingRedirect>** elements within a **<dependentAssembly>** element. For more information on the **<dependentAssembly>** element, see the ".NET Framework General Reference."

#### Side-by-Side Issues with Serviced Components

As you update your serviced components, or if you have a scenario where multiple versions of the same component are on the same computer, you can encounter several issues caused by having these components installed side by side. To help you maintain your serviced components, you should adhere to the following advice:

- Do not use globally unique identifiers (GUIDs) as the **GuidAttribute** class or **ApplicationIDAttribute**. Instead use the **ApplicationNameAttribute**. If you do use a GUID, you need to hard code the GUID into the assembly, which requires you to manually change this GUID if you update your component and have it run alongside the original version; otherwise, the COM+ installation overwrites the settings for the earlier version.
- Remember that assembly versioning applies only to the assemblies themselves, not the COM+ application. There is no automatic way to version the application itself. If the newer version of a serviced component requires changes at the COM+ application level that will break previous versions of the component, then you need to use the application name to indicate versioning by installing the newer version of the serviced component into its own COM+ application.

For more information about serviced components, see "Understanding Enterprise Services (COM+) in .NET" on MSDN.

## ASP.NET Applications Running Side-by-Side

The .NET Framework allows you to install multiple versions of the runtime on the same computer. By default, when the .NET Framework is installed on a computer with an existing installation, all ASP.NET applications are automatically updated to use this version of the .NET Framework. The only exceptions are applications that are bound to an incompatible version of the runtime or to a later version of the runtime. Although later versions of the .NET Framework are designed to be backwards compatible, you might want to configure an ASP.NET application to use an earlier version.

### Script Maps for ASP.NET Applications

When multiple versions of the .NET Framework are installed on the same computer, each installation contains an associated version of the ASP.NET ISAPI (aspnet_filter.dll). An ASP.NET application uses the ASP.NET ISAPI to determine which version of the .NET Framework to use for the application. An ASP.NET application can be configured to use any of the installed ASP.NET ISAPI versions. To specify the ASP.NET ISAPI version to use for an ASP.NET application, a script map is registered in IIS for the application.

A script map associates a file extension and HTTP verb with the appropriate ISAPI for script handling. For example, when IIS receives a request for an .aspx file, the script map for the application directs IIS to forward the requested file to the appropriate version of the ASP.NET ISAPI for processing. The script map for each ASP.NET application is normally set in the IIS management console and can be applied directly to an application, or inherited from a parent application. By default, when the .NET Framework is installed, the script maps for all existing ASP.NET applications on the computer are automatically updated to use the ASP.NET ISAPI version associated with the installation, unless the application uses a later or an incompatible version.

To make it easier to reconfigure the script map for an ASP.NET application, each installation of the .NET Framework comes with an associated version of the ASP.NET IIS Registration tool (Aspnet_regiis.exe). By default, this tool is installed in the following directory:

**<*windir*>\Microsoft.NET\Framework*versionNumber***

Administrators can use this tool to remap an ASP.NET application to the ASP.NET ISAPI version associated with the tool.

---

**Note:** Because Aspnet_regiis.exe is associated with a specific version of the .NET Framework, administrators must use the appropriate version of Aspnet_regiis.exe to reconfigure the script map for an ASP.NET application. Aspnet_regiis.exe only reconfigures the script map of an ASP.NET application to the ASP.NET ISAPI version associated with the tool.

---

The tool can also be used to display the status of all installed versions of ASP.NET, register the associated version of ASP.NET, create client-script directories, and perform other configuration operations.

For more information about script maps and IIS configuration, see the documentation for IIS.

For more information on updating script maps for an ASP.NET application, see the "Configuring an ASP.NET Application for an ASP.NET Version" on MSDN.

#### Using Application Center to Manage Side-by-Side Instances

Although not limited to ASP.NET deployments, Application Center is useful at deploying ASP.NET applications and can make side-by-side deployment easier. Therefore, when deploying upgraded server applications to your cluster, you should consider using Application Center to implement side-by-side application instances. Side-by-side application instances allow you to retain an existing version of your application while you deploy a new version to the production environment and verify that it functions as expected. The following scenario explains how this works.

Imagine the application you need to upgrade has been installed into a folder called *CommerceApp2001*, and that this application has been deployed to all members of your cluster to implement a Web farm. Rather than directly upgrade this application

and have it synchronized across your cluster, you can install your upgraded application into a different folder, for example, *CommerceApp2002*, and create a new virtual directory that maps to this folder.

You can have this new virtual directory synchronized across your cluster and the application located in *CommerceApp2001* will not have been modified and will continue to handle requests from clients. At this stage, you will have two versions of the application both able to function in the production environment. You can then verify that the application in *CommerceApp2002* functions as expected in the real production environment, and you might consider advertising to selected customers the fact that your application has been upgraded by informing them of the URL for the new application. They can then provide you with valuable feedback on your upgraded application before you release it to a wider audience.

When you are satisfied that the solution functions as expected and you are confident that you can release it for use by all clients, you can perform the following simple operations:

- Modify the virtual directory settings on your cluster controller so that your original application now references the *CommerceApp2002* folder, rather than *CommerceApp2001*.
- Synchronize your application across the cluster—the mapping between virtual directory and local folder is stored in the IIS metabase and this setting will be replicated from the cluster controller to the other cluster members.

If it becomes necessary to revert to the original version, rolling back this change is simply a matter of setting the virtual directory to *CommerceApp2001* and then Application Center will synchronize the change across the cluster.

For more information about implementing your specific deployment scenarios with Application Center, see "Best Practices for Phased Deployment Using Application Center 2000."

### Security Considerations

Each installation of the .NET Framework has a separate security configuration file, there are no forward or backward compatibility issues with security settings. However, if your application depends on the additional security capabilities of ADO.NET included in the .NET Framework version 1.1, you will not be able to distribute it to a version 1.0 system.

### ASP.NET and Dependent Components

As with Windows Forms-based applications, there are circumstances where you may have an ASP.NET application and a dependent component compiled on different versions of the Framework. Unlike Windows-based applications the Framework itself will not be existing side by side, as the version used will be defined at the

virtual directory or Web level. The following table shows how the application and component will behave in these circumstances.

**Table 4.4: Side by Side of an ASP.NET Application**

| Application Version | Only .NET Framework v1.0 | Only .NET Framework v1.1 |
|---|---|---|
| Application v1.0<br>Component v1.0 | No issues with compatibility | It will attempt to run with the v1.1 CLR[1] |
| Application v1.0<br>Component v1.1 | Will run under the v1.0 CLR[2] | It will attempt to run with the v1.1 CLR[1] |
| Application v1.1<br>Component v1.0 | Will run under the v1.0 CLR[3] | It will attempt to run with the v1.1 CLR[1] |
| Application v1.1<br>Component v1.1 | Will run under the v1.0 CLR[3] | No issues with compatibility |

# Summary

Deploying applications successfully in a relatively simple environment is one thing, deploying those applications in environments where other applications already exist, and maintaining those applications through multiple upgrades can be quite another. This chapter should help you deal with the conflicts you may find between different applications and any difficulties that may arise from attempting to update your applications.

# More Information

For more information on Windows Installer SDK tools:
Windows Installer Development Tools
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/windows_installer_development_tools.asp*

For more information on .pcp file properties:
UpgradedImages Table (PATCHWIZ.DLL)
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/upgradedimages_table_patchwiz_dll_.asp*

1. Could break if any of the components are using functionality that has been broken in the new version of the Framework. For more information on the list of break points in the functionality, please see the link at the end of this chapter.
2. Could break if the components are using v1.1 APIs as they are not supported in the earlier version.
3. This is true if the **<supportedRuntime>** element is used, otherwise it will send an error.

For more information on patch packages:
A Small Update Patching Example
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/a_small_update_patching_example.asp*

For a list of the minor upgrade requirements:
Changing the Product Code
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/changing_the_product_code.asp*

For more information about deploying and updating applications with Group Policy:
see the Step-by-Step Guide to Software Installation and Maintenance
*http://www.microsoft.com/windows2000/techinfo/planning/management/swinstall.asp*

For more information on the REINSTALL property:
Platform SDK REINSTALL Property reference
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/reinstall_property.asp*

For more information on the REINSTALLMODE property:
Platform SDK REINSTALLMODE Property reference
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi/setup/reinstallmode_property.asp*

For more information about deploying and updating applications with SMS:
SMS product documentation and the white paper, "Deploying Windows Installer Setup Packages with Systems Management Server 2.0"
*http://www.microsoft.com/smserver/docs/deploymsi.doc*

For more information on façade executables:
.NET Applications: .NET Application Updater Component
*http://windowsforms.net/articles/appupdater.aspx*

For detailed information on how assemblies are loaded:
How the Runtime Locates Assemblies
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconhowruntimelocatesassemblies.asp*

For more information about publisher policy files:
Redirecting Assembly Versions
*http://msdn.microsoft.com/library/en-us/cpguide/html/cpconassemblyversionredirection.asp*

and

Creating a Publisher Policy File
*http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcreatingpublisherpolicyfile.asp*

For more information about redirecting assembly binding:
Redirecting Assembly Versions
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconassemblyversionredirection.asp*

For more information on side-by-side issues for distributed applications:
Side-by-Side and Versioning Considerations for .NET Remoting
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/versremote.asp*

For more information on compatibility changes to the .NET Framework:
Backwards Breaking Changes from version 1.0 to 1.1
*http://www.gotdotnet.com/team/changeinfo/Backwards1.0to1.1/default.aspx*

For more information on the <supportedRuntime> element:
See the .NET Framework General Reference
*http://msdn.microsoft.com/library/en-us/cpgenref/html/gnconsupportedruntimeelement.asp*

For more information of the <requiredRuntime element>
See the .NET Framework General Reference
*http://msdn.microsoft.com/library/en-us/cpgenref/html/gngrfrequiredruntime.asp*

For more information on the <bindingRedirect> element:
See the .NET Framework General Reference
*http://msdn.microsoft.com/library/en-us/cpgenref/html/gngrfbindingredirect.asp*

For more information on the <dependentAssembly> element:
See the .NET Framework General Reference
*http://msdn.microsoft.com/library/en-us/cpgenref/html/gngrfdependentassembly.asp*

For more information about serviced components:
Understanding Enterprise Services (COM+) in .NET
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/entserv.asp*

For more information about implementing your specific deployment scenarios with Application Center:
Best Practices for Phased Deployment Using Application Center 2000
*http://www.microsoft.com/applicationcenter/techinfo/deployment/2000/wp_phaseddeploy.asp*

For more information on updating script maps for an ASP.NET application:
Configuring an ASP.NET Application for an ASP.NET Version
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconconfiguringaspnetapplicationforaspnetversion.asp*

## Authors

Paul Slater (Lead Technical Author) - Wadeware LLC, Robert Hill (Lead Technical Contributor) - Implement.com, Jason Hogg (Program Manager) - Microsoft Corporation

## Collaborators

Many thanks to the following advisors who provided invaluable assistance:

Steve Hoag, Mike Wade, Dave Templin, David Guyer, Sean Draine, Jamie Cool, Greg Singleton, Dennis Angeline, Rishi Rana, Julio Castillo, Erich Pleny, David Blair (Metratech), Erik Olsen, David Fleischman, Duncan Mackenzie, Arash Ghanaie-Sichanie, Robert McIntyre, Chris Flaat, Chad Royal, Izzy Gryko, Mark Boulter, Sebastian Lange, Steven Pratschner, Jay Allen, Kenny Jones, Michael Stuart, Mike Kass, Pete Coupland (VMC Consulting Corporation), Riyaz Pishori, Sonja Keserovic, Sushil Chordia, Jeff Kercher, RoAnn Corbisier, Chris Sfanos, Andrew Mason, Edward Jezierski, Ron Jacobs, J.D. Meier, Claudette Iebbiano (CI Design Studio), Ryan Plant (Jet Blue)

Microsoft®

patterns & practices

*Proven practices for predictable results*

Patterns & practices are Microsoft's recommendations for architects, software developers, and IT professionals responsible for delivering and managing enterprise systems on the Microsoft platform. Patterns & practices are available for both IT infrastructure and software development topics.

Patterns & practices are based on real-world experiences that go far beyond white papers to help enterprise IT pros and developers quickly deliver sound solutions. This technical guidance is reviewed and approved by Microsoft engineering teams, consultants, Product Support Services, and by partners and customers. Organizations around the world have used patterns & practices to:

**Reduce project cost**

- Exploit Microsoft's engineering efforts to save time and money on projects
- Follow Microsoft's recommendations to lower project risks and achieve predictable outcomes

**Increase confidence in solutions**

- Build solutions on Microsoft's proven recommendations for total confidence and predictable results
- Provide guidance that is thoroughly tested and supported by PSS, not just samples, but production quality recommendations and code

**Deliver strategic IT advantage**

- Gain practical advice for solving business and IT problems today, while preparing companies to take full advantage of future Microsoft technologies.

**To learn more about *patterns & practices* visit: *msdn.microsoft.com/practices***

**To purchase *patterns & practices* guides visit: *shop.microsoft.com/practices***

patterns & practices
*Proven practices for predictable results*

# patterns & practices

*Proven practices for predictable results*

Patterns & practices are available for both IT infrastructure and software development topics. There are four types of patterns & practices available:

### Reference Architectures

Reference Architectures are IT system-level architectures that address the business requirements, operational requirements, and technical constraints for commonly occurring scenarios. Reference Architectures focus on planning the architecture of IT systems and are most useful for architects.

### Reference Building Blocks

References Building Blocks are re-usable sub-systems designs that address common technical challenges across a wide range of scenarios. Many include tested reference implementations to accelerate development.

Reference Building Blocks focus on the design and implementation of sub-systems and are most useful for designers and implementors.

### Operational Practices

Operational Practices provide guidance for deploying and managing solutions in a production environment and are based on the Microsoft Operations Framework. Operational Practices focus on critical tasks and procedures and are most useful for production support personnel.

### Patterns

Patterns are documented proven practices that enable re-use of experience gained from solving similar problems in the past. Patterns are useful to anyone responsible for determining the approach to architecture, design, implementation, or operations problems.

**To learn more about *patterns & practices* visit: *msdn.microsoft.com/practices***

**To purchase *patterns & practices* guides visit: *shop.microsoft.com/practices***

# patterns & practices current titles

**December 2002**

## Reference Architectures

Microsoft Systems Architecture—Enterprise Data Center *2007 pages*
Microsoft Systems Architecture—Internet Data Center *397 pages*
Application Architecture for .NET: Designing Applications and Services *127 pages*
Microsoft SQL Server 2000 High Availability Series: Volume 1: Planning *92 pages*
Microsoft SQL Server 2000 High Availability Series: Volume 2: Deployment *128 pages*
Enterprise Notification Reference Architecture for Exchange 2000 Server *224 pages*
Microsoft Content Integration Pack for Content Management Server 2001 and SharePoint Portal Server 2001 *124 pages*
UNIX Application Migration Guide *694 pages*
Microsoft Active Directory Branch Office Guide: Volume 1: Planning *88 pages*
Microsoft Active Directory Branch Office Series Volume 2: Deployment and Operations *195 pages*
Microsoft Exchange 2000 Server Hosting Series Volume 1: Planning *227 pages*
Microsoft Exchange 2000 Server Hosting Series Volume 2: Deployment *135 pages*
Microsoft Exchange 2000 Server Upgrade Series Volume 1: Planning *306 pages*
Microsoft Exchange 2000 Server Upgrade Series Volume 2: Deployment *166 pages*

## Reference Building Blocks

Data Access Application Block for .NET *279 pages*
.NET Data Access Architecture Guide *60 pages*
Designing Data Tier Components and Passing Data Through Tiers *70 pages*
Exception Management Application Block for .NET *307 pages*
Exception Management in .NET *35 pages*
Monitoring in .NET Distributed Application Design *40 pages*
Microsoft .NET/COM Migration and Interoperability *35 pages*
Production Debugging for .NET-Connected Applications *176 pages*
Authentication in ASP.NET: .NET Security Guidance *58 pages*
Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication *608 pages*

## Operational Practices

Security Operations Guide for Exchange 2000 Server *136 pages*
Security Operations for Microsoft Windows 2000 Server *188 pages*
Microsoft Exchange 2000 Server Operations Guide *113 pages*
Microsoft SQL Server 2000 Operations Guide *170 pages*
Deploying .NET Applications: Lifecycle Guide *142 pages*
Team Development with Visual Studio .NET and Visual SourceSafe *74 pages*
Backup and Restore for Internet Data Center *294 pages*

**For current list of titles visit: *msdn.microsoft.com/practices***

**To purchase *patterns & practices* guides visit: *shop.microsoft.com/practices***

patterns & practices
*Proven practices for predictable results*