

Steven Holzner

Presented by:



Microsoft Visual C# .NET 2003

KICK START

SAMS

800 East 96th Street, Indianapolis, Indiana 46240

Microsoft Visual C# .NET 2003 Kick Start

Copyright © 2004 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32547-0

Library of Congress Catalog Card Number: 2003092630

Printed in the United States of America

First Printing: July 2003

06 05 04 03 4 3 2

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

1-317-428-3341

international@pearsontechgroup.com

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis.

Associate Publisher

Michael Stephens

Acquisitions Editor

Neil Rowe

Development Editor

Songlin Qiu

Managing Editor

Charlotte Clapp

Project Editor

George E. Nedeff

Copy Editor

Kezia Endsley

Indexer

Ken Johnson

Proofreader

Juli Cook

Technical Editor

Doug Holland

Team Coordinator

Cindy Teeters

Multimedia Developer

Dan Scherf

Interior Designer

Gary Adair

Cover Designer

Gary Adair

Page Layout

Michelle Mitchell

Creating Windows Services, Web Services, and Deploying Applications

Creating Windows Services

There are three programmer-to-programmer topics in this chapter: Windows services, Web services, and how to deploy applications using Windows installer (.MSI) files. All these issues are important ones, and we'll use them to round off the GUI-oriented application coverage. We'll start with Windows services.

Windows services are not typically front-line applications that the user runs and interacts with. Instead, they provide support services, often for device drivers, such as printer device drivers, audio devices, data providers, CD creation software, and so on. As such, Windows services don't need a real user interface as you see in standard Windows applications. They often do have a control panel-like interface that the users can open by clicking or right-clicking an icon in the taskbar, however. (You can create taskbar icons in Windows applications using the `NotifyIcon` control from the Windows Forms tab in the toolbox.) Users can customize, and even start or stop, a Windows service using that control panel. Other Windows applications can interact with the service at runtime; for example, SQL Server uses a Windows service to make it accessible to other applications.



12

IN THIS CHAPTER

- ▶ **Creating Windows Services** 433
- ▶ **Creating Web Services** 448
- ▶ **Deploying Your Applications** 454

We'll see how to create a working Windows service in this chapter. In the FCL, Windows services are based on the `ServiceBase` class, and that class gives us most of the support we'll need. When you write a Windows service, you should override the `OnStart` and `OnStop` methods—even though their names imply they are event handlers, they're actually methods. These methods are called when the service starts and stops. You might also want to override the `OnPause` and `OnContinue` event handlers to handle occasions where the service is paused and resumed.

SHOP TALK

ABUSING WINDOWS SERVICES

Currently, there is a great deal of abuse of Windows services, and it's getting so bad that sooner or later there's going to be a user revolt. Too many software manufacturers just decide that the user's computer has nothing better to do than to continuously run their software, and so you find Windows services that do nothing else besides check—once a second—whether the software manufacturer's software is running, or printer drivers that run as a Windows service, also polling once a second and displaying multiple taskbar icons and pop-ups. Some one-time-use applications, like a few tax programs, appear to install Windows services that run continuously as long as the user has the computer. Other manufacturers use Windows services to gather information about the user's work habits, installed software, and/or accessed files to send over the Internet without the user's knowledge.

Because Windows services can be invisible, they've been incredibly abused. If you look in the Service Control Manager tool that we'll discuss in this chapter, you may find a dozen or so Windows services running that you've never heard of before. It's very important to resist this temptation to monopolize the user's machine. If you need to poll your device driver or service, you can start a `Timer` object (see Chapter 7, "Creating C# Windows Applications") when the service's `OnStart` method is called, but don't use this technique to wrest control from the user more than just occasionally, unless you specifically let the user know what's going on. Many users, when told what some formerly unknown Windows services are doing, consider them no better than viruses.

You can configure Windows services to start automatically when the computer starts, or you can start them manually using an administration tool built into Windows, the Service Control Manager (SCM).

We'll see how this works in practice now. You can see an example, `ch12_01`, in the code for this book, and we'll take that application apart here. To follow along, create a new Windows service project. Choose File, New, Project in the IDE, and select the Windows Service icon in the Templates box of the New Project dialog box. Give this new service the name `ch12_01` and click OK. This creates the new Windows service project in Figure 12.1; the default name for this new service is "Service1".

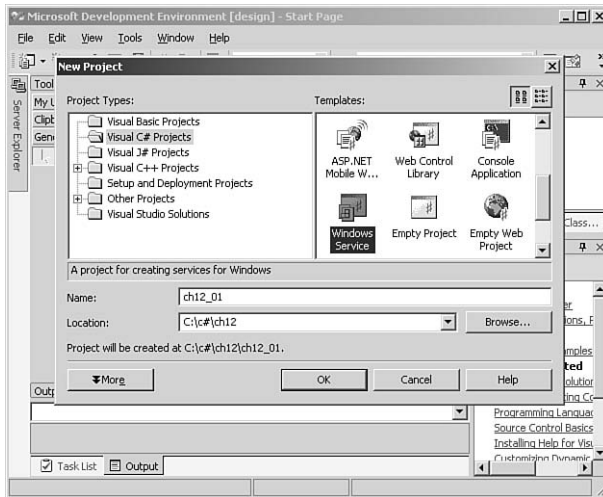


FIGURE 12.1 A new Windows service.

In our Windows service, we are going to write to our Windows service's event log when the `OnStart` and `OnStop` methods are called. To handle an event log, you must specify or create an *event source*. An event source registers your application with the event log as a source of data so the event log can listen for that data. You can give the event source as any string, but the name must be unique among other registered sources.

In this example, we're going to register our event log in the Windows service's constructor, which you can find in the Component Designer generated code region of the Windows service's code, `Service1.cs`. That constructor looks like this now:

```
public Service1()
{
    // This call is required by the Windows.Forms Component Designer.
    InitializeComponent();

    // TODO: Add any initialization after the InitializeComponent call
}
```

In this example, we create an event source named "CSSource1". After the new source is created, we assign its name to the `eventLog1` object's `Source` property like this:

```
public Service1()
{
    // This call is required by the Windows.Forms Component Designer.
    InitializeComponent();
```

```
if (!System.Diagnostics.EventLog.SourceExists("CSSource1"))
{
    System.Diagnostics.EventLog.CreateEventSource("CSSource1",
        "currentLog1");
}

eventLog1.Source = "CSSource1";
}
```

Now we're ready to write to our event log when the service starts and stops. You can do that in the `OnStart` and `OnStop` methods, which look like this in `Service1.cs` currently:

```
protected override void OnStart(string[] args)
{
    // TODO: Add code here to start your service.
}

protected override void OnStop()
{
    // TODO: Add code here to perform any tear-down necessary to
    // stop your service.
}
```

To write text to a Windows service's event log, you can use the log's `WriteEntry` method. In this case, we'll insert a message into the log indicating that the service started or stopped, like this:

```
protected override void OnStart(string[] args)
{
    eventLog1.WriteEntry("Starting ch12_01.");
}

protected override void OnStop()
{
    eventLog1.WriteEntry("Stopping ch12_01.");
}
```

When our Windows service starts, our code will write "Starting ch12_01." to event log `currentLog1`, and when the service stops, our code will write "Stopping ch12_01." to the log.

We've created our Windows service. To install that service, we'll need an installer, so click the `Service1.cs[Design]` tab now to open the designer for `Service1`. Make sure that `eventLog1`

in that designer does *not* have the focus (we want to create an installer for the service itself, not the event log object in the service), and then click the Add Installer link in the description section of the properties window (this link is visible at bottom right in Figure 12.2).

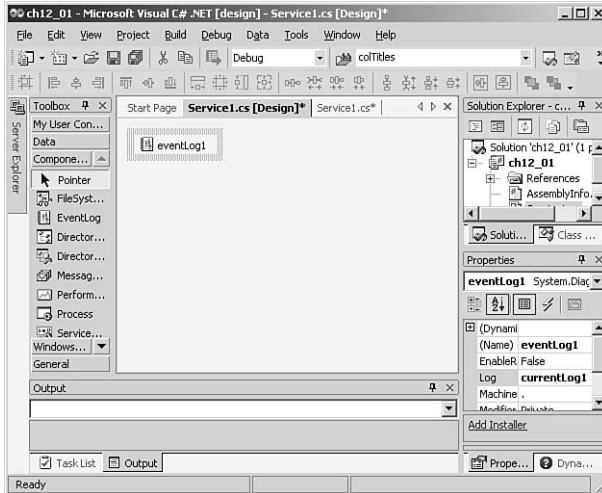


FIGURE 12.2 Adding an event log to a Windows service.

This creates ProjectInstaller.cs with two objects in it, serviceProcessInstaller1 and serviceInstaller1, as you see in Figure 12.3.

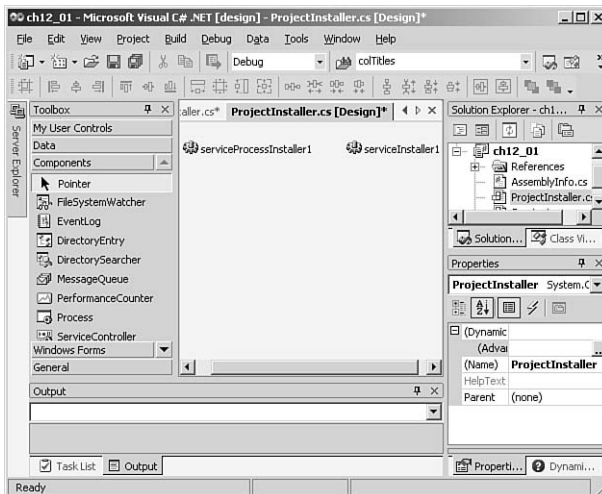


FIGURE 12.3 Creating an installer for a Windows service.

`ServiceInstaller` objects inform Windows about a service by writing Windows Registry values for the service to a Registry subkey under the `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services` Registry key. The service is identified by its `ServiceName` value in this subkey. `ServiceProcessInstaller` objects handle the individual processes started by our service.

When you install a Windows service, you have to indicate which account it should run under. In this example, we'll do that by clicking the `serviceProcessInstaller1` object to give it the focus and setting its `Account` property to `LocalSystem`. Besides `LocalSystem`, you can also set this property to `LocalService`, `NetworkService`, or `User`. When you set `Account` to `User`, you must set the `Username` and `Password` properties of the `serviceProcessInstaller1` object to configure this object for a specific user account.

Now click the `serviceInstaller1` object and make sure its `ServiceName` property is set to the name of this service, `Service1` (it should already be set that way). You use the `ServiceInstaller1` object's `StartType` property to indicate how to start the service. Here are the possible ways of starting the service, using values from the `ServiceStartMode` enumeration:

- `ServiceStartMode.Automatic`—The service should be started automatically when the computer boots.
- `ServiceStartMode.Disabled`—The service is disabled (so it cannot be started).
- `ServiceStartMode.Manual`—The service can only be started manually (by either using the Service Control Manager, or by an application).

USING MANUAL STARTUP WHILE DEVELOPING WINDOWS SERVICES

A Windows service with errors in it that starts automatically on boot can make Windows unstable, so be careful when testing Windows services. Until you're sure a service is working correctly, it's best to keep its start mode `Manual`, making sure it doesn't start again automatically if you need to reboot. (If you get into a loop where a problematic Windows service is starting automatically when you boot and causing Windows to hang, press and hold the F8 key while booting so Windows comes up in safe mode.)

The safest of these while testing a new Windows service is `Manual`, so set the `StartType` property of the `ServiceInstaller1` object to `Manual` now.

Installing a Windows Service

The next step is to build and install our new Windows service, `Service1`. To build the service, select `Build`, `Build ch12_01`, which creates `ch12_01.exe`. To actually install the service in Windows, you can use the `InstallUtil.exe` tool that comes with the .NET Framework. In Windows 2000, for example, you can find `InstallUtil.exe` in the

`C:\WINNT\Microsoft.NET\Framework\xxxxxxx` directory, where `xxxxxxx` is the .NET Framework's version number.

Here's how you install ch12_01.exe using InstallUtil.exe at the DOS command prompt (note that the command line here is too wide for the page, so it's split into two lines):

```
C:\WINNT\Microsoft.NET\Framework\xxxxxxx>installutil  
c:\c#\ch12\ch12_01\bin\Debug\ch12_01.exe
```

```
Microsoft (R) .NET Framework Installation utility Version xxxxxxxx  
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
```

Running a transacted installation.

Beginning the Install phase of the installation.

See the contents of the log file for the

c:\c#\ch12\ch12_01\bin\debug\ch12_01.exe assembly's progress.

The file is located at c:\c#\ch12\ch12_01\bin\debug\ch12_01.InstallLog.

Installing assembly 'c:\c#\ch12\ch12_01\bin\debug\ch12_01.exe'.

Affected parameters are:

assemblypath = c:\c#\ch12\ch12_01\bin\debug\ch12_01.exe

logfile = c:\c#\ch12\ch12_01\bin\debug\ch12_01.InstallLog

Installing service Service1...

Service Service1 has been successfully installed.

Creating EventLog source Service1 in log Application...

The Install phase completed successfully, and the Commit phase is beginning.

See the contents of the log file for the

c:\c#\ch12\ch12_01\bin\debug\ch12_01.exe assembly's progress.

The file is located at c:\c#\ch12\ch12_01\bin\debug\ch12_01.InstallLog.

Committing assembly 'c:\c#\ch12\ch12_01\bin\debug\ch12_01.exe'.

Affected parameters are:

assemblypath = c:\c#\ch12\ch12_01\bin\debug\ch12_01.exe

logfile = c:\c#\ch12\ch12_01\bin\debug\ch12_01.InstallLog

The Commit phase completed successfully.

The transacted install has completed.

```
C:\WINNT\Microsoft.NET\Framework\xxxxxxx>
```

Now our Windows service has been installed. If InstallUtil hadn't been able to install the new service without problems, it would have rolled back the installation and removed the non-working service.

AUTO-INSTALLING A WINDOWS SERVICE

You can also deploy Windows services with setup programs in C#; we'll cover setup programs at the end of this chapter. You create setup programs with setup projects, and to install a Windows service, you add a *custom action* to a setup project. In the Solution Explorer, right-click the setup project, select View, and then select Custom Actions, making the Custom Actions dialog box appear. In the Custom Actions dialog box, right-click the Custom Actions item and select Add Custom Action, making the Select Item in Project dialog box appear. Double-click the Application Folder in the list box, opening that folder. Select Primary Output from *ServiceName* (Active), where *ServiceName* is the name of your service, and click OK. The primary output (that is, the Windows service itself) is added to all four custom action folders—Install, Commit, Rollback, and Uninstall.

Our new Windows service is installed, but not yet started, because we chose manual startup. In this case, we'll use the Service Control Manager to start our service. The SCM is part of Windows; for example, in Windows 2000, you can start the SCM this way:

- In Windows 2000 Server, you select Start, select Programs, click Administrative Tools, and click Services.
- In Windows 2000 Professional, right-click the My Computer icon on the desktop and select the Manage item in the menu that pops up. In the dialog box that appears, expand the Services and Applications node and click the Services item.

You can see the Service Control Manager in Figure 12.4, and you can see our newly installed service, *Service1*, listed in the SCM in the figure.

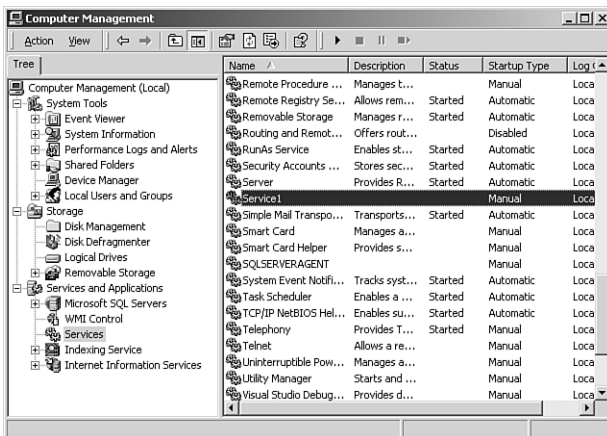


FIGURE 12.4 The Service Control Manager.

To start our new service, right-click `Service1` in the Service Control Manager now and select the Start item in the menu that appears. Doing so starts the service, as you see in Figure 12.5, where `Service1` is listed as Started.

To stop the service, right-click `Service1` in the Service Control Manager and select the Stop item.

RUNNING A WINDOWS SERVICE FROM THE SERVER EXPLORER

You can also run a Windows service from the Server Explorer; just expand the Services node, and then right-click the service you want to start and click Start.

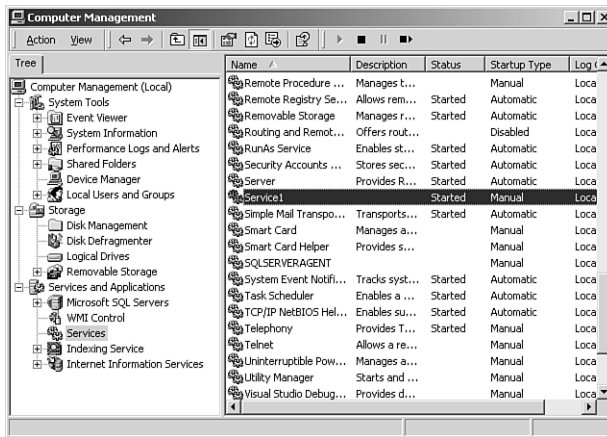


FIGURE 12.5 Starting a Windows service.

We've been able to start and stop our new service, so it should have written to our event log, `currentLog1`. You can check whether it has from inside the IDE; you just open the Server Explorer's Event Logs node as you see in Figure 12.6, and take a look at the entry for `CSSource1` in `currentLog1`. Here, you can see our service's two entries—Starting `ch12_01` and Stopping `ch12_01`—in the event log in the Server Explorer in Figure 12.6. (If you don't see the messages there, or messages don't appear when you start and stop the service a number of times, refresh the event log by right-clicking `currentLog1` in the Server Explorer and selecting the Refresh item.) And that's it—the Windows service is a success.

Our Windows service did exactly what it was supposed to—it wrote to an event log when it was started and stopped. Now that you can run code in a Windows service, you can see this is only the beginning. You can get the service's code started when the `OnStart` method is called, and run it in the background, supplying its service for as long as needed.

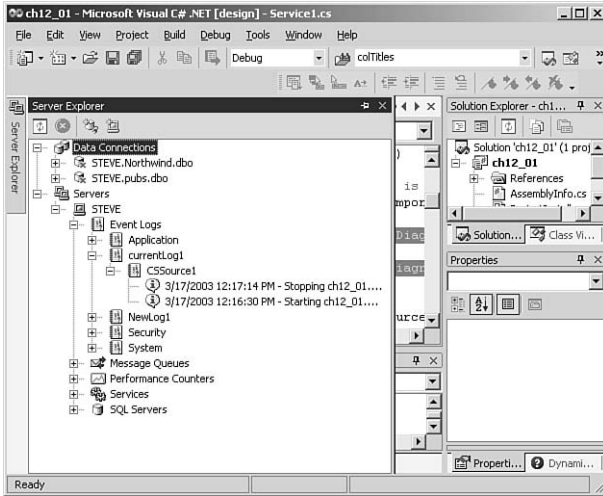


FIGURE 12.6 The Server Explorer.

Uninstalling a Windows Service

You can uninstall a Windows service, removing it from the SCM, with InstallUtil.exe; just use the /u option to uninstall. Here's what you see when you uninstall the Windows service—notice that the command line is the same except for the /u:

```
C:\WINNT\Microsoft.NET\Framework\xxxxxxx>installutil
c:\c#\ch12\ch12_01\bin\Debug\ch12_01.exe /u
```

```
Microsoft (R) .NET Framework Installation utility Version xxxxxxxx
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
```

The uninstall is beginning.

See the contents of the log file for the

```
c:\c#\ch12\ch12_01\bin\debug\ch12_01.exe assembly's progress.
```

The file is located at c:\c#\ch12\ch12_01\bin\debug\ch12_01.InstallLog.

```
Uninstalling assembly 'c:\c#\ch12\ch12_01\bin\debug\ch12_01.exe'.
```

Affected parameters are:

```
assemblypath = c:\c#\ch12\ch12_01\bin\debug\ch12_01.exe
logfile = c:\c#\ch12\ch12_01\bin\debug\ch12_01.InstallLog
```

Removing EventLog source Service1.

Service Service1 is being removed from the system..

Service Service1 was successfully removed from the system.

The uninstall has completed.

```
C:\WINNT\Microsoft.NET\Framework\xxxxxxxxx>
```

Interacting with Windows Services from Other Applications

We've seen how to run a Windows service in the background, but how do you connect to that Windows service from another application? All you need to do is drag a `ServiceController` object from the Components tab of the Toolbox to a form in a Windows application, creating a new object, `serviceController1`. Then you set these properties in the properties window for this object:

- `MachineName`—The name of the computer that hosts the service, or "." for the local computer.
- `ServiceName`—The name of the service you want to work with.

Now using the power of the Windows service becomes easy—you can use the properties and methods exposed by the Windows service as though they were properties and methods of the `serviceController1` object. For example, here's how you might use the `CanStop` and `ServiceName` properties of a Windows service as connected to by `serviceController1`:

```
if (serviceController1.CanStop)
{
    MessageBox.Show(serviceController1.ServiceName + " can be stopped.");
}
```

You can also create a `ServiceController` object in code. To do that, you add a reference to the `System.ServiceProcess` DLL by right-clicking the current project in the Solution Explorer and selecting Add Reference. Then, click the .NET tab in the Add Reference dialog box, select `System.ServiceProcess.dll` and click Select. Finally, click OK to close the Add Reference dialog box. Also, you should include a using statement in your application's code for the `System.ServiceProcess` namespace. Now you can access the properties and methods that are built into a Windows service using the new `ServiceController` object like this:

```
using System.ServiceProcess;
.
.
.
ServiceController controller1 = new ServiceController("Service1");

if (controller1.CanStop)
{
```

```

    MessageBox.Show(controller1.ServiceName + " can be stopped.");
}

```

That's all it takes to access a Windows service from code. (Note that you can even implement events in a Windows service, and, using `ServiceController` objects, handle those events in other applications.)

Next, we'll take a look at the properties, methods, and events of a few of the important Windows services classes to round off this topic.

Working with the ServiceBase Class

The `ServiceBase` class is the base class for Windows services, and you can find the significant public properties of `ServiceBase` objects in Table 12.1, and their significant protected methods in Table 12.2 (note that although the items in Table 12.2 look like events, they're actually methods you can override).

TABLE 12.1

Significant Public Properties of ServiceBase Objects

PROPERTY	PURPOSE
<code>AutoLog</code>	Sets whether to record in the event log automatically.
<code>CanPauseAndContinue</code>	Returns or sets whether the service can be paused and continued.
<code>CanShutdown</code>	Returns or sets whether the service should be informed when the computer shuts down.
<code>CanStop</code>	Returns or sets whether the service can be stopped.
<code>EventLog</code>	Returns the event log.
<code>ServiceName</code>	Returns or sets the name of the service.

TABLE 12.2

Significant Protected Methods of ServiceBase Objects

METHOD	PURPOSE
<code>OnContinue</code>	Called when a service continues (after it was paused).
<code>OnPause</code>	Called when a service is paused.
<code>OnShutdown</code>	Called when the system shuts down.
<code>OnStart</code>	Called when the service starts.
<code>OnStop</code>	Called when a service stops running.

Working with the EventLog Class

The EventLog class supports access to Windows event logs used by Windows services; you can find the significant public static methods of EventLog in Table 12.3, the significant public properties of EventLog objects in Table 12.4, their significant methods in Table 12.5, and their significant events in Table 12.6.

TABLE 12.3
Significant Public Static Methods Properties of the EventLog Class

METHOD	PURPOSE
CreateEventSource	Creates an event source to let you write to a log.
Delete	Deletes a log.
DeleteEventSource	Deletes an event source.
Exists	Returns true if a log exists.
GetEventLogs	Returns an array of event logs.
SourceExists	Checks whether an event source exists.
WriteEntry	Writes an entry to the log.

TABLE 12.4
Significant Public Properties of EventLog Objects

PROPERTY	PURPOSE
Entries	Returns the contents of the log.
Log	Returns or sets the name of the log.
LogDisplayName	Returns the log's display name.
MachineName	Returns or sets the name of the log's computer.
Source	Returns or sets the source name to use when writing to the log.

TABLE 12.5
Significant Public Methods of EventLog Objects

METHOD	PURPOSE
BeginInit	Begins initialization of a log.
Clear	Clears all the entries in a log.
Close	Closes the log.
EndInit	Ends initialization of a log.
WriteEntry	Writes an entry in the log.

TABLE 12.6**Significant Public Events of EventLog Objects**

EVENT	PURPOSE
EntryWritten	Happens when data is written to a log.

Working with the ServiceProcessInstaller Class

As we've already seen, `ServiceProcessInstaller` objects let you install specific processes in a Windows service. You can find the significant public properties of objects of the `ServiceProcessInstaller` class in Table 12.7, their significant methods in Table 12.8, and their significant events in Table 12.9.

TABLE 12.7**Significant Public Properties of ServiceProcessInstaller Objects**

PROPERTY	PURPOSE
Account	Returns or sets the type of account for the service.
HelpText	Returns help text.
Installers	Returns the service's installers.
Parent	Returns or sets the parent installer.
Password	Returns or sets the password for a user account.
Username	Returns or sets a user account.

TABLE 12.8**Significant Public Methods of ServiceProcessInstaller Objects**

METHOD	PURPOSE
Install	Installs a service, writing information to the Registry.
Rollback	Rolls back an installation, removing data written to the Registry.
Uninstall	Uninstalls an installation.

TABLE 12.9**Significant Public Events of ServiceProcessInstaller Objects**

EVENT	PURPOSE
AfterInstall	Happens after an installation.
AfterRollback	Happens after an installation is rolled back.
AfterUninstall	Happens after an uninstallation.
BeforeInstall	Happens before installation.
BeforeRollback	Happens before installers are rolled back.

TABLE 12.9

Continued

EVENT	PURPOSE
BeforeUninstall	Happens before an uninstallation.
Committed	Happens after all installers have committed installations.
Committing	Happens before installers commit installations.

Working with the ServiceInstaller Class

You use `ServiceInstaller` objects to install Windows services, and you can find the significant public properties of objects of this class in Table 12.10, their significant methods in Table 12.11, and their significant events in Table 12.12.

TABLE 12.10

Significant Public Properties of ServiceInstaller Objects

PROPERTY	PURPOSE
DisplayName	The display name for this service.
HelpText	Help text for the installers.
Installers	Returns the installers.
Parent	Returns or sets the parent installer.
ServiceName	The name of this service.
ServicesDependedOn	Specifies services that must be running in order to support this service.
StartType	Specifies when to start this service.

TABLE 12.11

Significant Public Methods of ServiceInstaller Objects

METHOD	PURPOSE
Commit	Commits an installation.
Install	Installs the service by writing data to the Registry.
Rollback	Rolls back a service's data written to the Registry.
Uninstall	Uninstalls the service.

TABLE 12.12**Significant Public Methods of ServiceInstaller Objects**

EVENT	PURPOSE
AfterInstall	Happens after the installers have installed.
AfterRollback	Happens after the installations are rolled back.
AfterUninstall	Happens after all the installers finish their uninstallations.
BeforeInstall	Happens just before the each installer's Install method runs.
BeforeRollback	Happens before the installers are rolled back.
BeforeUninstall	Happens before the installers uninstall.
Committed	Happens after all the installers commit their installations.
Committing	Happens before the installers commit their installations.

That completes our look at Windows services; next up are Web services.

Creating Web Services

We've seen Windows services, and many people are familiar with them on a day-to-day basis. But what about Web services? They're less familiar than Windows services to most people, but the idea is the same—they're code components—on Web servers this time—that your code can call and that can return data, much as you'd work with a Windows service.

For example, Web services can connect to databases, letting you retrieve data from data sources, and you can connect to Web services from either Windows or Web applications. Using Web services, you can implement custom logic such as performing a credit check before approving a loan. Web services are often used as middle-tier business objects, which work with and transfer data in three-tier data Web applications.

Writing the Web Service

As an example, we're going to create a three-tier data application here. This example is going to use its middle tier—our Web service—to connect to the authors example database and fetch or update data from that database. The lowest tier is a Windows application, the middle tier is our Web service that fetches or updates data on demand (although note you can also implement business rules, such as only returning authors for whom you have books in stock, in the middle tier), and the top tier is the authors database in SQL Server.

This Web service example is named `ch02_12`. In Web services, you can implement methods callable from Windows or Web applications (and, in fact, from applications running on other platforms as well); in this example, we're going to write two methods: `GetAuthors` to return a dataset holding the authors table, and `UpdateAuthors` to update that table as needed.

We're going to use a Windows application to call these Web service methods, and we're going to use a data grid to display the authors table. As long as the user's machine is connected to the Internet, our operations will appear as though they're taking place entirely in the Windows application.

To follow along, create a new Web service project named `ch12_02` with File, New, Project, selecting the ASP.NET Web Service icon. Create the new Web service project you see in Figure 12.7.

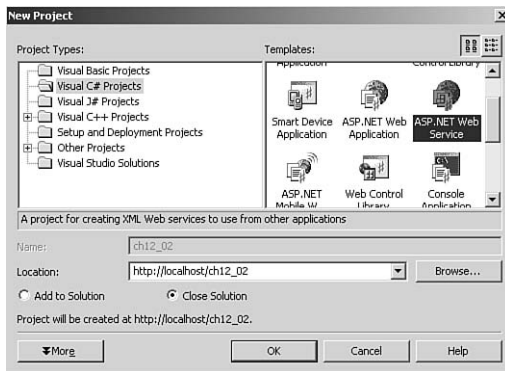


FIGURE 12.7 A new Web service project.

The default name of our new Web service is `Service1`, and we'll keep that name. The C# code for this service is stored in `Service1.asmx.cs`, and in that file you'll see that this new Web service is derived from the `WebService` class:

```
public class Service1 :
    System.Web.Services.WebService
{
    public Service1()
    {
        .
        .
        .
    }
}
```

Because we intend to use the authors database, we'll need to connect to it. As we've done before, drag a `SqlDataAdapter` object from the toolbox; in this case, drag that object onto the Web service designer. Doing so opens the Data Adapter Configuration Wizard; use that tool to connect the data adapter to the authors table. Finally, use Data, Generate Dataset to create a new dataset class, `DataSet1` (no dataset object are created, just the class `DataSet1`). This is the dataset class we'll use to access the authors table in the Web service. (Alternatively, you can connect and create a dataset in code, storing the connection string in the project's `Web.config` file.)

How does the Windows application connect to this new Web service? The Web service will expose methods that we'll write, and the Windows application can call those methods after we create a reference to our Web service. To expose methods in a Web service, you use the `[WebMethod]` attribute. For example, in the `GetAuthors` method, we want to return a dataset filled with the authors table, so we add this code to `Service1.asmx.cs` now:

```
[WebMethod]
public DataSet1 GetAuthors()
{
    DataSet1 authors = new DataSet1();
}
```

```
        sqlDataAdapter1.Fill(authors);  
        return authors;  
    }  
}
```

This code makes the `GetAuthors` method return a dataset object of our new `DataSet1` class that will hold the authors table. As you can see, all we need to do is to create a new object of that class and use the data adapter to fill the object before returning it to the calling code.

The `UpdateAuthors` method, which updates the authors table when the user makes changes in the data grid, is easy to write. We'll pass this method a dataset holding the changed records in the authors table and update the authors table using the data adapter's `Update` method like this:

```
[WebMethod]  
public DataSet1 UpdateAuthors(DataSet1 UpdatedRecords)  
{  
    sqlDataAdapter1.Update(UpdatedRecords);  
    return UpdatedRecords;  
}
```

We've created our two Web methods, `GetAuthors` and `UpdateAuthors`, at this point. To make this Web service available to our Windows application, you can build the Web service now using `Build`, `Build ch12_02` in the IDE. Our Web service is ready for use.

Writing a Windows Application to Connect to Our Web Service

The next step involves writing the Windows application that will call the Web service's `GetAuthors` and `UpdateAuthors` methods. To create this Windows application, add a new Windows Application project to the current solution with `File, Add Project, New Project`. (This application doesn't have to be part of the current solution to connect to the Web service. You can add a reference to the Web service in any Windows application.)

Select the Windows Application icon in the `Add New Project` dialog box, name this new Windows application `ch12_03`, and click `OK` to create the Windows application as you see in [Figure 12.8](#). Also, make this Windows application the startup project by selecting it in the `Solution Explorer` and using `Project, Set as StartUp Project`.

Now that we've created our Windows application, the next step is to connect to the Web service, which you do by adding a Web reference to the Web service. Right-click the `ch12_03` entry in the `Solution Explorer` and select `Add Web Reference`, opening the `Add Web Reference` dialog box. This dialog box lists the available Web services. To add a reference to a Web service, you can enter the URL for the service's `.VSDISCO` (Visual Studio Discovery) file in the `Add Web Reference` dialog box's `Address` box. You can also browse to the service by

clicking the link in the Add Web Reference dialog box for the server you want and then clicking the name of the service, which is `Service1` in this example.

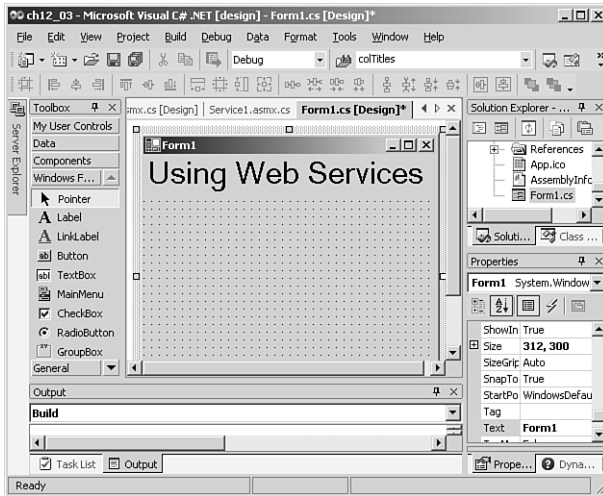


FIGURE 12.8 A new Windows application.

When you select a Web service this way, that Web service appears in the Add Web Reference dialog box, as you see in Figure 12.9, where the Add Web Reference dialog box is displaying the `Service1` Web service. To add a Web reference to this service to the Windows application, click the Add Reference button.

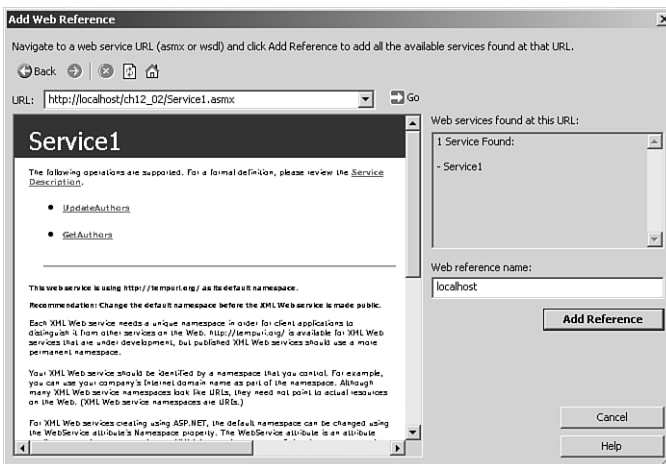


FIGURE 12.9 The Add Web Reference dialog box.

At this point, we have a Web reference to `Service1`, which means our Windows application will know how to find the `GetAuthors` and `UpdateAuthors` methods. To put those methods to work in the Windows application, add a data grid to the application and put two buttons above the data grid with the captions `Get Authors` and `Update Authors`.

To handle data from the Web service, we'll need to let the Windows application know about the `DataSet1` class in the Web service. To do that, drag a new `DataSet` object—not a data adapter—from the `Data` tab of the toolbox to the Windows application. When you do, the `Add DataSet` dialog box opens.

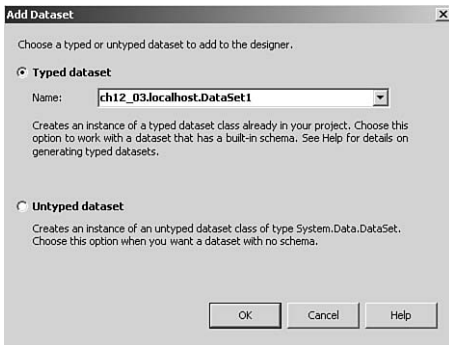


FIGURE 12.10 The `Add DataSet` dialog box.

In the `Add DataSet` dialog box, make sure the `Typed Dataset` radio button is selected, and select the dataset class in our Web service, `DataSet1`, from the drop-down list. Note that the fully qualified name of `DataSet1` is `ch12_03.localhost.DataSet1`, which is the way it appears in the `Add DataSet` dialog box, as shown in [Figure 12.10](#).

To create a new dataset object of the `DataSet1` class, `dataSet11`, click the `OK` button in the `Add DataSet` dialog box. Now we have a dataset that matches the `DataSet1` class in the Web server, which means we can use that dataset as a repository for the data sent to us from the Web service. We bind `dataSet11` to the data grid in the Windows application by setting the data grid's `DataSource` property to `dataSet11`, and its `DataMember` property to `authors`.

We want to fill `dataSet11` with data when the user clicks the `Get Authors` button. When the user clicks that button we'll start by creating an object, `service1`, of our Web service:

```
private void button1_Click(object sender, System.EventArgs e)
{
    ch12_03.localhost.Service1 service1 = new ch12_03.localhost.Service1();
    .
    .
    .
}
```

Using this new object, `service1`, you now have access to the methods built into our Web service. For example, if you wanted to fetch data using the `GetAuthors` method, you just have to call that method like this: `service1.GetAuthors()`, which returns a dataset filled with the authors table. You can store the data from that dataset in `dataSet11` with the `Merge` method like this:

```
private void button1_Click(object sender, System.EventArgs e)
{
    ch12_03.localhost.Service1 service1 = new ch12_03.localhost.Service1();
    dataSet11.Merge(service1.GetAuthors());
}
```

That's how the process works—you create a new object corresponding to the Web service, much as you would with any reference added to your application. Then you can use that object's methods to call the Web service directly.

Besides the `GetAuthors` method, we can also call the `UpdateAuthors` method when the user clicks the Update Authors button. When the user makes changes in the data grid, those changes are sent to our local dataset. We need to send those changes back to the data store when the user clicks the Update Authors button. You can do that by creating a dataset holding just the changed records using the dataset's `GetChanges` method and the `DataSet Merge` method:

```
private void button2_Click(object sender, System.EventArgs e)
{
    ch12_03.localhost.DataSet1 update1 = new ch12_03.localhost.DataSet1();
    update1.Merge(dataSet11.GetChanges());
    .
    .
    .
}
```

We'll call the Web service's `UpdateAuthors` method to update the authors table. That method returns the changed records, and we will merge them back into the Windows application's dataset object, making sure those records will no longer be marked as newly changed:

```
private void button2_Click(object sender, System.EventArgs e)
{
    ch12_03.localhost.Service1 service1 = new ch12_03.localhost.Service1();
    ch12_03.localhost.DataSet1 update1 = new ch12_03.localhost.DataSet1();
    update1.Merge(dataSet11.GetChanges());
    dataSet11.Merge(service1.UpdateAuthors(update1));
}
```

Finally, add a data grid to the Windows application and bind it to `dataSet11` to show our results. That completes the code we need for both the Web service and the Windows application that connects to it. Run this example now and click the Get Authors button. When you do, the authors table is retrieved from the Web service and displayed in the bound data grid, as you see in Figure 12.11.



FIGURE 12.11 Connecting to a Web service from a Windows application.

In addition, when you edit the data in the data grid and click the Update Authors button, those changes will be sent back to the data store. And that's it—we've created a Web service and connected to that Web service in code from a Windows application. As far as the user is concerned, the whole connection process to the Web service was transparent. Web services like this are great for many uses. Employees out in the field using Web services, for example, can connect immediately to the latest version of the home office's business logic as posted to the Internet.

Working with the WebService Class

As we've seen, the `System.Web.Services.WebService` class is the base class for Web Services. You can find the significant public properties of objects in the `Webservice` class in Table 12.13.

TABLE 12.13

Significant Public Properties of Webservice Objects

PROPERTY	PURPOSE
Application	The HTTP Application object for the HTTP request.
Context	The <code>HttpContext</code> object for the HTTP request.
Server	The <code>HttpServerUtility</code> object for the HTTP request.
Session	The <code>HttpSessionState</code> object for the HTTP request.
User	The ASP.NET server User object.

Deploying Your Applications

When you create a .NET application, you can often copy it to another .NET machine simply by copying the .EXE file. On the other hand, installing most real applications isn't that simple; there's usually much more to install than just an .EXE file. You might want to add items to the machine's Start menu, and so on. The best way to install C# applications on .NET machines is to create a Microsoft Installer (.MSI) file, and we'll see how to create such a file here.

To install an application, all you have to do is to copy the .MSI file to the target machine and double-click it. The Microsoft Windows Installer will do the rest, as we'll see. You can also use a `setup.exe` program, which we'll also create here, to run the .MSI file for you.

Creating a Deployable Application

The IDE lets you create deployment files easily. To see how this works, we're going to create an .MSI file for a Windows application, ch12_04. This super-advanced application displays a message box with the text "Thanks for choosing ch12_04!" when the user clicks a Click Me! button. That's all ch12_04 does:

```
private void button1_Click(object sender, System.EventArgs e)
{
    MessageBox.Show("Thanks for choosing ch12_04!");
}
```

This application appears in Figure 12.12 at design time, where you can see the Click Me! button in the main form.

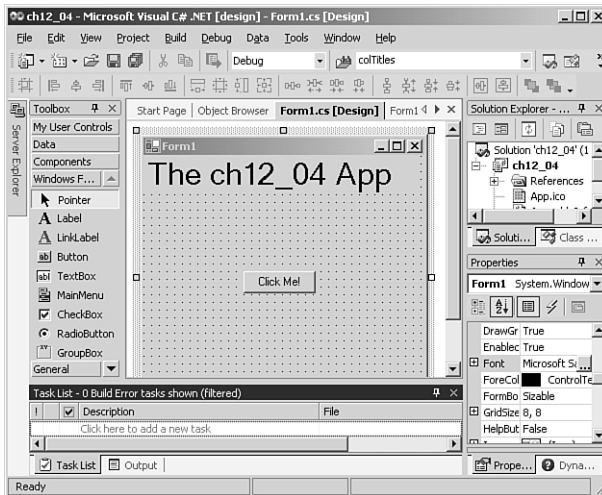


FIGURE 12.12 The ch12_04 application.

To create a deployment package for ch12_04, we'll need an .EXE file for the application, so select Build, Build ch12_04 to create ch12_04.exe. The goal is to create an .MSI file for this application, and to deploy it.

Creating an Installer File

You can create an installer file with a new type of project, a deployment project, which we'll call ch12_05. To create ch12_05 and add it to the current solution (which contains only ch12_04 at the moment), select File, Add Project, New Project, opening the Add New Project dialog box, which you see in Figure 12.13.

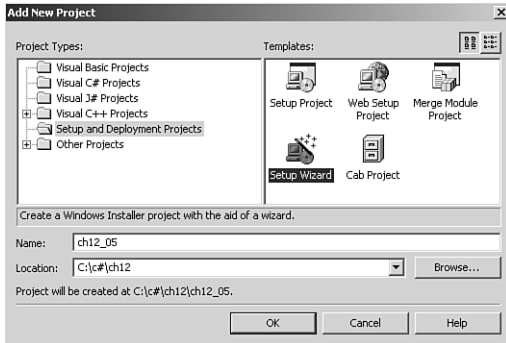


FIGURE 12.13 Creating a deployment project with the Add New Project dialog box.



FIGURE 12.14 The Setup Wizard, first pane.

The `ch12_05` project is going to be a deployment project, not a standard C# project, so select the Setup and Deployment Projects folder in the Project Types pane of the Add New Project dialog box, and the Setup Wizard icon in the Templates pane. There are various options for deployment projects, but the Setup Wizard lets you create them in the quickest way. We'll give this new deployment project the name `ch12_05`, as you see in Figure 12.13. Click OK to open the first pane of the Setup Wizard, as shown in Figure 12.14.

The first pane of the Setup Wizard introduces the wizard; click Next to move to the pane you see in Figure 12.15. As mentioned, the Setup Wizard supports various types of deployment projects, including those for Windows and Web applications. We're going to deploy a Windows application, so select the Create a Setup for a Windows Application radio button.

Click Next to open the third pane in the Setup Wizard, which appears in Figure 12.16.

The third pane lets you specify what files you want to deploy. You can deploy just the application (the check box labeled Primary Output from `ch12_04`), or the application and its documentation, or just the source code, and so forth. In this example, we will deploy all items for the `ch12_04` application, so select all the items, as shown in Figure 12.16.

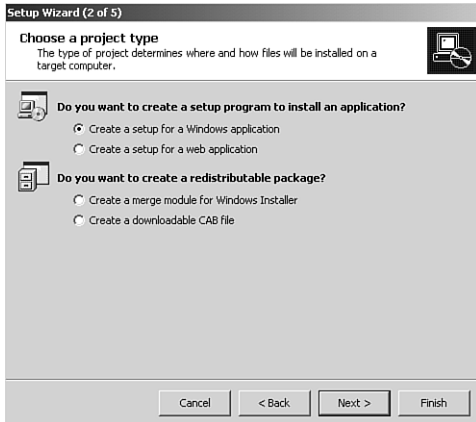


FIGURE 12.15 The Setup Wizard, second pane.

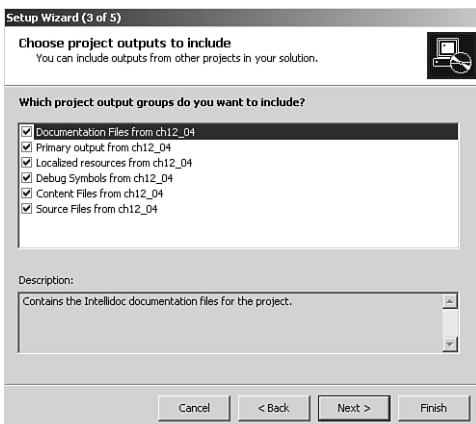


FIGURE 12.16 The Setup Wizard, third pane.

Clicking Next again moves you to the fourth pane of the Setup Wizard, shown in Figure 12.17. In this pane, the Setup Wizard lets you add other files to be deployed, such as additional documentation, license information, contact information, and so on.

In this example we're not going to deploy any additional files, so just click the Next button to move to the fifth and final pane of the Setup Wizard, shown in Figure 12.18. This pane of the Setup Wizard summarizes what the wizard will do; click Finish to create the installer file.

Clicking Finish closes the Setup Wizard, and you'll see the file structure of the setup project, as shown in Figure 12.19. It's easy to move files around to different target locations in the user's machine just by dragging them. You can specify the name of the application that the Windows installer displays by setting the setup project's `ProductName` property, and you can also set the `Manufacturer` property to the name of your company.

We've now created our deployment project. To create the .MSI file, select Build, Build ch12_05 in the IDE, creating ch12_05.msi. This is our deployment file; all you need to do is to copy it to the target machine. Double-clicking that file on the target machine opens the Windows installer, as you see in Figure 12.20.

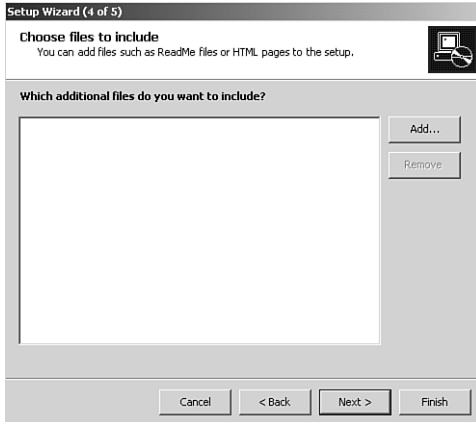


FIGURE 12.17 The Setup Wizard, fourth pane.

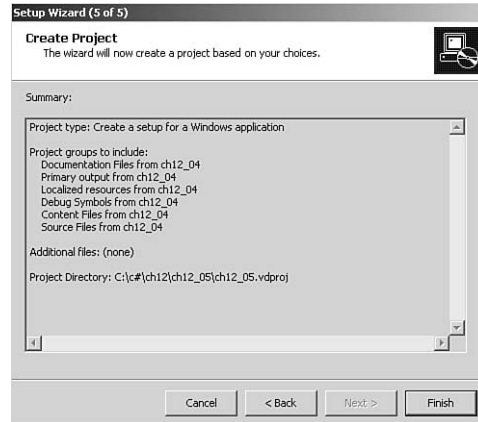


FIGURE 12.18 The Setup Wizard, fifth pane.

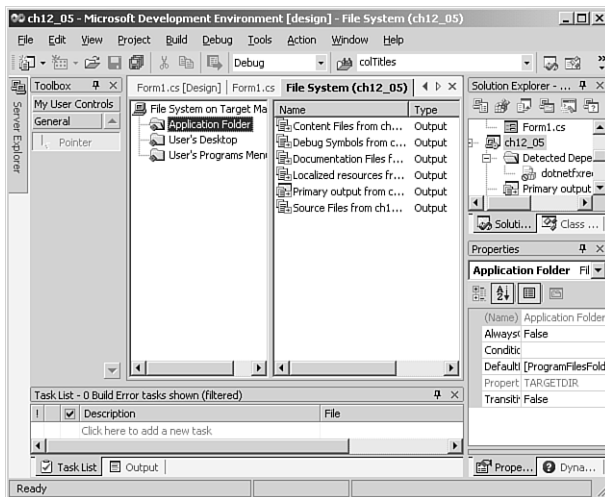


FIGURE 12.19 The setup project in the IDE.

Besides creating the .MSI file, building the deployment project also creates the files setup.exe and setup.ini in the same directory as the .MSI file. If you copy all three files to the target machine and place them in the same directory, the user only has to run setup.exe, which will launch the .MSI file automatically.

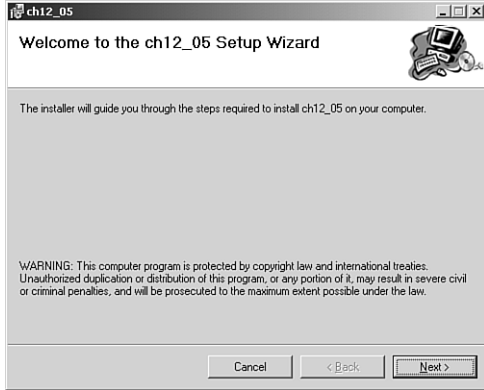


FIGURE 12.20 The Windows installer, first pane.

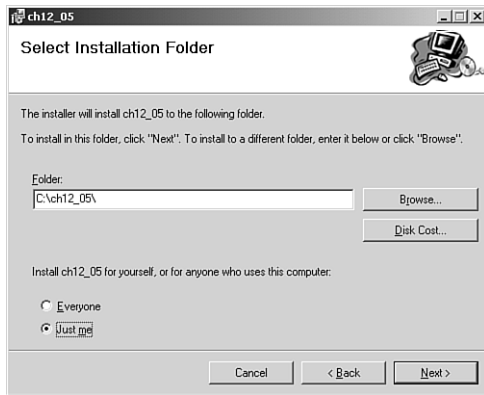


FIGURE 12.21 The Windows installer, second pane.

Click Next in the Windows installer, making its second pane appear, as shown in Figure 12.21. This pane lets you indicate where to install the application.

Clicking Next twice more installs the application, as you see in Figure 12.22. That's it; you've now installed the application, ch12_04, using the installer file ch12_05.msi.

To complete the test, double-click the newly installed ch12_04.exe file, running that application as you see in Figure 12.23.

At this point, we've created a Windows application, an installer for that application, and used the installer to install the application on a target machine. In this case, we installed a Windows application, but the whole process is much the same for Web applications. To create an installer for Web applications, select the Create a Setup for a Web Application option in the second pane of the Setup Wizard.

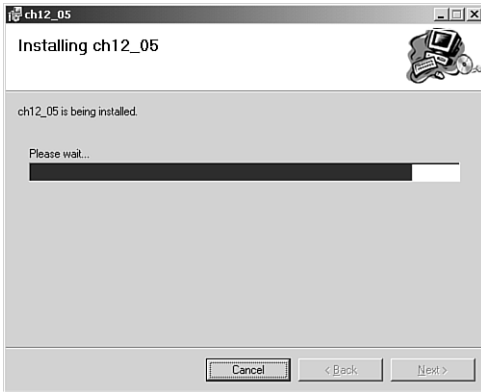


FIGURE 12.22 Installing ch12_04 using the ch12_05 installer project.



FIGURE 12.23 Running ch12_04.

In Brief

In this chapter, we looked at three important topics—Windows services, Web services, and how to deploy applications. Here's an overview of this chapter's topics:

- Windows services run in the background, and you have the option of starting them automatically when the computer starts. These services are typically used to configure device drivers such as those that handle printers, CD creation software, audio devices, and data providers like SQL Server. You can give a Windows service a user interface, which often resembles a control panel that opens when the user clicks or right-clicks a taskbar icon. You can create a taskbar icon with the `NotifyIcon` control in the IDE's toolbox.
- You can create Windows services in the IDE and the IDE will write much of the C# code you need. Windows services are based on the `ServiceBase` class. To implement your Windows service, you override various methods of the `ServiceBase` class, including `OnStart` and `OnStop`, to handle Windows service actions.
- To install Windows services, you need an installer for every service you want to install. That involves using both the `ServiceProcessInstaller` and `ServiceInstaller` classes in C#.
- You can use a Windows service's `StartType` property to indicate when the service should start. `ServiceStartMode.Automatic` means that the service should be started automatically when the computer is booted, `ServiceStartMode.Disabled` means that it cannot be started, and `ServiceStartMode.Manual` means that the service can only be started manually (by using the Service Control Manager or by an application).

- You can interact with a Windows service in an application with an object of the `ServiceController` class. You can call the methods of the service using an object of this class.
- Web services expose methods that can be called by other code across the Internet. In C#, Web services are based on the `WebService` class. To expose methods from a Web service, you declare them with the `[WebMethod]` attribute.
- You can call the Web methods of Web services if you first add a Web reference to that service. You can do that by right-clicking a project in the Solution Explorer, selecting Add Web Reference, and browsing to the Web service you want in the Add Web Reference dialog box. When you have a Web reference to a Web service, you create a new object corresponding to that service and call the Web methods of that object.
- To deploy your application, you can create .MSI (Microsoft Installer) files using a setup and deployment project. To create a deployment package for a project, you add a setup and deployment project to the current solution.
- After your setup project has been created, you build it to create the .MSI file you can deploy to target machines (if they're running the .NET Framework). On the target machine, double-click the .MSI file to open it in the Windows installer. Alternatively, you can also use the `setup.exe` and `setup.ini` files created by building the deployment project to install the application.