Chapter 1

Creating and Managing Windows Services

MICROSOFT EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- Create and manipulate a Windows service.
  - Write code that is executed when a Windows service is started or stopped.
- Implement security for a Windows service.
- Instrument and debug a Windows service.
  - Configure the debugging environment.
- Configure client computers and servers to use a Windows service.
Windows services provide a means for application logic to run continuously on your computer, usually providing device driver or other operating system services. Windows services are useful for server applications that should always be available for clients’ requests. If you are familiar with Microsoft SQL Server 2000, you will notice that it runs as a Windows service. An easy-to-understand example of a Windows service application is the Windows time service, which updates the clock you see on your computer’s taskbar. Until now, it was very difficult to develop this type of application by using Visual Basic. The .NET Framework contains a set of classes that provide the basic functionality for Windows service applications. Now it is easy to make use of these Framework classes and use Visual Basic .NET to implement customized Windows service applications.

In this chapter, you will learn how to use Visual Studio .NET to create a simple Windows service application using the `System.ServiceProcess.ServiceBase` class. Then you will look at another .NET Framework class, the `System.ServiceProcess.ServiceController` class, to learn how to create Visual Basic .NET applications that can programmatically control and send custom commands to a Windows service. You will also review some considerations for setting security options and debugging that are specific to Windows services.

**Introduction to Windows Services**

A *Windows service* is an application that runs on a server or workstation computer and provides ongoing functionality without direct user interaction. Windows services are often used to perform system monitoring.

A Windows service will run in its own process, independently of users or other programs running on the same computer. Windows services are frequently configured to start automatically when the computer boots up. Unlike most applications, Windows services run under their own security identity, rather than under the identity of the currently logged-in user. They can start running, even if there is no user logged onto the computer. This behavior is exactly what is needed for applications that run unattended on a server or that need to be available all the time on a desktop computer.

---

**NOTE**

The Visual Studio project templates and associated functionality that enable you to create Windows service applications are not available in the Visual Basic .NET Standard Edition. Neither is the Server Explorer feature. These features are included with Visual Studio .NET Professional, Enterprise Developer, and Enterprise Architect Editions.
When you create an application that will run as a Windows service, you must be careful not to include any user interface elements such as message boxes or other dialog boxes. A Windows service is not meant to provide a visual interface for users.

A Windows service will typically report its results and error messages to an event log.

### Real World Scenario

**Using Windows Services to Monitor a Directory**

You are a software developer for a medium-sized organization. You are hoping that some new features of the .NET Framework will help solve a problem that your department has been facing for some time. Your department is in charge of managing documents that are submitted for posting on your company’s website. Documents are submitted by many departments throughout the company. End users simply copy the files to a designated network-shared directory.

Your department needs to know when new files are added to the directory. A consultant who left long ago wrote a system to periodically check the directory, but no one currently on the staff knows how to make changes or maintain the program.

A Windows service application is the perfect solution for this type of requirement. The service will always be running on the server, so administrators do not have to remember to check the directory or manually run a program. The .NET Framework even provides other useful classes, such as the FileSystemWatcher, which handles the actual task of firing events when files are added, deleted, or changed in the target directory. The Windows service can write to an event log, so there is an audit history.

By using the security features in the .NET Framework, your application can check the user’s identity and permissions to make sure they are authorized to make changes. Other Framework classes provide the means to send an e-mail message, if necessary to notify administrators when new documents have been added.

It’s clear that the .NET Framework provides a wealth of features to quickly and easily design solutions for this kind of common business requirement.

An administrator can interactively manage Windows service applications by using the Service Control Manager (see Figure 1.1). You can find this tool under different menus, depending on the operating system you are using:

- **Start ➤ Programs ➤ Administrative Tools ➤ Services in Windows 2000 Server**
- **Start Settings ➤ Control Panel ➤ Administrative Tools ➤ Services in Windows 2000 Professional**
- **Start ➤ Control Panel ➤ Administrative Tools ➤ Services in Windows XP Professional**
Chapter 1 • Creating and Managing Windows Services

FIGURE 1.1 The Service Control Manager console

The Service Control Manager shows you a list of all services that are installed on the computer. For each service, you can see the name, description, current status (Started,Paused, or Stopped), startup type (Automatic—starts automatically on boot, or Manual) and the identity that the service logs on as. By using the menus and toolbar buttons, you can issue commands to start, stop, pause, continue, or restart the selected service. You can also view a Properties dialog box that enables you to change configuration options for a service.

Alternatively, you can view the Windows services running on your computer directly from within Visual Studio .NET by using the Server Explorer. To open the Server Explorer, choose View ➤ Server Explorer. Expand the node Servers, expand the node with your computer name, and then expand the Services node. You will see all the services that are running. When you right-click on a service, the pop-up menu provides options to start or stop the service and view the properties. You see a bit less detail here than in the Service Control Manager, but it is convenient to be able to start and stop the service from within Visual Studio .NET.

In Exercise 1.1, you will use the Windows Service Control Manager utility to view the existing Windows services that are currently running on your computer.

EXERCISE 1.1

Using the Service Control Manager

1. Start the Service Control Manager. For Windows 2000 Server, choose Start ➤ Programs ➤ Administrative Tools ➤ Services. (If you are using a different operating system, see the instructions provided earlier, immediately before Figure 1.1.).

2. Review the list of services that are running on your computer. For instance, select the Clipbook entry.
Creating a Windows Service by Using Visual Studio .NET

The .NET Framework classes include a set of base classes, in the System.ServiceProcess namespace, that provide the underlying functionality of a Windows service application. Visual Studio .NET offers a project template that automatically sets a reference to System.ServiceProcess and also provides you some boilerplate code. This section describes the default setup in detail. When you create a project by using the template, you need to concentrate only on the unique features that your application will implement.

When you create a new project in Visual Studio .NET and choose Windows Service as your project template, the project will initially look like Figure 1.2. The default project contains one

---

**EXERCISE 1.1 (continued)**

3. Right-click on the service and choose Start or Stop from the pop-up menu.

4. The status of the service changes in the Service Control Manager window. It is important to note that you should right-click the service name again and return the service to its original state. You don’t want to inadvertently cause another application that depends on this service to fail or, conversely, to leave an unnecessary service running.

5. Right-click one more time and choose Properties. Review the choices that are available in the Properties dialog box.

---

Creating a Windows Service by Using Visual Studio .NET

The .NET Framework classes include a set of base classes, in the System.ServiceProcess namespace, that provide the underlying functionality of a Windows service application. Visual Studio .NET offers a project template that automatically sets a reference to System.ServiceProcess and also provides you some boilerplate code. This section describes the default setup in detail. When you create a project by using the template, you need to concentrate only on the unique features that your application will implement.

When you create a new project in Visual Studio .NET and choose Windows Service as your project template, the project will initially look like Figure 1.2. The default project contains one
component class module (with the default name Service1.vb). If you view the code inside Service1.vb (see Listing 1.1), you will notice that a class has been created (also using the default class name Service1). This class inherits from the System.ServiceProcess.ServiceBase namespace. The template also adds an Imports statement for the System.ServiceProcess namespace.

Listing 1.1: Default Code for a Windows Service Application

Imports System.ServiceProcess

Public Class Service1
    Inherits System.ServiceProcess.ServiceBase

    'Component Designer generated code appears here

    Protected Overrides Sub OnStart(ByVal args() As String)
        ' Add code here to start your service. This method
        ' should set things in motion so your service can
        ' do its work.
        End Sub
    
    Protected Overrides Sub OnStop()
        ' Add code here to perform any tear-down necessary to
        ' stop your service.
        End Sub

    End Class

END OF LISTING 1.1
Protected Overrides Sub OnStop()
    ' Add code here to perform any teardown necessary
    ' to stop your service.
End Sub

End Class

If you expand the References node in the Solution Explorer window, you can see that a reference has been added for System.ServiceProcess.dll. Note that the .dll suffix is not present.

The default code also contains two procedure definitions for important methods of the ServiceBase class, OnStart and OnStop. You will add your custom code to these, and other methods, to implement the specific behavior of your Windows service application.

If you expand the region titled Component Designer Generated Code, you will see implementations for the New and Dispose methods, with code specific to how these standard Framework methods should be coded for a Windows service. There is also a Sub Main() procedure with some code needed for a Windows service to be started correctly (see Listing 1.2). The code in this procedure calls the Run method of the ServiceBase class and passes a reference to a new instance of your service. This is the code that enables your service to start when the operating system or a user invokes it.

Listing 1.2: Component Designer Generated Code
' The main entry point for the process
<MTAThread()> _
Shared Sub Main()
    Dim ServicesToRun() As _
        System.ServiceProcess.ServiceBase

    ' More than one NT Service may run in the same
    ' process. To add another service to this process,
    ' change the following line to create a second
    ' service object. For example,
    ' ServicesToRun = New _
    '    System.ServiceProcess.ServiceBase() _
    '    {New Service1, New MySecondUserService}
    ' ServicesToRun = New System.ServiceProcess.ServiceBase() _
    '    {New Service1}
    System.ServiceProcess.ServiceBase.Run(ServicesToRun)
End Sub
Creating and Managing Windows Services

Methods and Properties of the **ServiceBase Class**

Now that you have seen the basics required to create a Windows service application, you can concentrate on creating a service with custom functionality. To do this, you will provide custom implementations for methods of the parent `ServiceBase` class (see Table 1.1). The `ServiceBase` class also defines properties that you can set to affect the behavior of your service (see Table 1.2).

**TABLE 1.1** Methods of the `ServiceBase` Class

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OnContinue</code></td>
<td>Implement this method to run custom code when a service is resumed after being paused.</td>
</tr>
<tr>
<td><code>OnCustomCommand</code></td>
<td>Implement this method when you need custom actions that can be called programmatically by a <code>ServiceController</code> object.</td>
</tr>
<tr>
<td><code>OnPause</code></td>
<td>Implement this method to run custom code when a service is paused.</td>
</tr>
<tr>
<td><code>OnPowerEvent</code></td>
<td>Implement this method to run custom code when the computer’s power status has changed—for example, a laptop computer going into suspended mode.</td>
</tr>
<tr>
<td><code>OnShutdown</code></td>
<td>Implement this method to run custom code before the computer shuts down.</td>
</tr>
<tr>
<td><code>OnStart</code></td>
<td>Implement this method to run custom code when a service starts. It is preferred to put initialization code in this procedure rather than in the constructor (Sub <code>New</code> method).</td>
</tr>
<tr>
<td><code>OnStop</code></td>
<td>Implement this method to run custom code when a service is stopped.</td>
</tr>
</tbody>
</table>

---

**Tip**

You will see how to implement the `OnCustomCommand` method later in this chapter, in the section titled “Executing Custom Commands for a Service.”

**Note**

It is preferred to use the `OnStart` method for any code that must run when your service is started. Code in the constructor method, `Sub New`, runs when the service is instantiated, before it is completely started and running in the context of the Service Control Manager. Also, the Visual Studio .NET documentation states that “there is no guarantee the objects will be reinitialized when you restart a service after it has been stopped.”
The Project Installers are “helper” classes that you add to your Windows service project. They provide important information that is used during the installation of your service application, such as the name that will be displayed in the Service Control Manager console, whether the service is started automatically or manually, and the security account. The security account is a Windows user login or system account that provides the identity and permissions that the Windows service will run with. Each Windows service project will have one instance of the ServiceInstaller class and one instance of the ServiceProcessInstaller class for each service that is included in the project.

When you are working in the Visual Studio .NET Integrated Development Environment (IDE), you can add ServiceInstaller components directly to your project from the Toolbox.

If you prefer, you can also create these objects in code. You will see an example of that in Chapter 10, “Deploying, Securing, and Configuring Windows-Based Applications.”

### Table 1.2 Properties of the ServiceBase Class

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoLog</td>
<td>If this property is set to True, every time the service is started, stopped, paused, or continued, an entry will be written to the Windows Application event log. Set this property to False if you want to code custom log messages.</td>
</tr>
<tr>
<td>CanHandlePowerEvent</td>
<td>Set this to True if you have written custom code for the OnPowerEvent method. This will enable you to take special action if the computer that your service is running on experiences a change in power status—for example, a laptop computer going into suspended mode.</td>
</tr>
<tr>
<td>CanPauseAndContinue</td>
<td>Set this value to True if you want to allow your service to be paused.</td>
</tr>
<tr>
<td>CanShutdown</td>
<td>Set this to True if you have written custom code for the OnShutdown method. This will enable you to take special action before the computer shuts down.</td>
</tr>
<tr>
<td>CanStop</td>
<td>This value is usually set to True. It is set to False for some important operating system services, which should not be stopped by a user.</td>
</tr>
<tr>
<td>EventLog</td>
<td>If the AutoLog property is set to True, messages will be written to the Windows Application event log. If you set AutoLog to False, then you can specify a different event log for messages.</td>
</tr>
<tr>
<td>ServiceName</td>
<td>Gets or sets the service name.</td>
</tr>
</tbody>
</table>

---

### Project Installer Classes

The Project Installers are “helper” classes that you add to your Windows service project. They provide important information that is used during the installation of your service application, such as the name that will be displayed in the Service Control Manager console, whether the service is started automatically or manually, and the security account. The security account is a Windows user login or system account that provides the identity and permissions that the Windows service will run with. Each Windows service project will have one instance of the ServiceInstaller class and one instance of the ServiceProcessInstaller class for each service that is included in the project.

When you are working in the Visual Studio .NET Integrated Development Environment (IDE), you can add ServiceInstaller components directly to your project from the Toolbox.

If you prefer, you can also create these objects in code. You will see an example of that in Chapter 10, “Deploying, Securing, and Configuring Windows-Based Applications.”
Setting Security Account Context for Windows Services

A Windows service runs independently of any user who might be logged onto the computer; therefore, the service must have a security identity of its own. When you create a Windows service application, you can select from one of four options for the security identity:

**User**  You create a specific username and password (using the standard Windows tools for doing so) for your application. Provide this username and password during installation. You must also provide this user with the appropriate permissions to complete the work of the Windows service application.

**LocalSystem**  *LocalSystem* is a built-in Windows account. It is the most commonly used setting for Windows services. It is a highly privileged account and is seen by other servers as an anonymous account.

**LocalService**  This is a built-in Windows account. It provides limited privileges on the local computer and is seen by other computers on the network as an anonymous user, so it is unlikely that code running under this identity will be allowed access to resources on other computers on the network. This account is available only on Windows XP and later operating systems.

**NetworkService**  This is a built-in Windows account. It runs with limited privileges on the local computer and can communicate with other servers as an authenticated domain account. This account is available only on Windows XP and later operating systems.

Again, the most commonly used security identity is LocalSystem. This built-in Windows account has a high level of privileges on the computer system. However, it is considered good security practice for applications to run with the least privileges required to perform their work. For example, do not allow the privilege to write to the system Registry if that is not needed to perform the core function of the service. To provide stronger security options, Windows XP and later operating systems have two new built-in accounts: LocalService and NetworkService. These two accounts have fewer privileges assigned to them by default. When installing a Windows service application, you should determine the level of privilege required and choose the best account.

*NOTE*  These security accounts and other security considerations are discussed more thoroughly in Chapter 10.

Running a Windows Service

Unlike most .NET projects, you cannot run a Windows service application directly from the Visual Studio .NET IDE by choosing Debug ➔ Start from the main menu (or its equivalent toolbar or keystroke shortcuts). If you try to do this, you will see a message box that reads:

“Cannot start service from the command line or a debugger. A Windows service must first be installed (using **Installutil.exe**) and then started with the Server Explorer, Windows Services Administrative tool or the **NET START** command.”
What this means is that you cannot interactively run your application for testing from within the Visual Studio .NET IDE. That is the way most Visual Basic .NET developers are used to working, and it’s very convenient. Working with Windows service applications is a bit more structured.

You must first build and install your Windows service before you can test and debug it to see whether it is working correctly. Although this seems like a big drawback to developing this type of application, keep in mind that a Windows service application runs in a different context than regular user applications. It runs in the context of the Service Control Manager and under a different security context than the user identity that you are logged in as during development. To debug a Windows service application, you must complete the application, install it, and then attach a debugger to the running process.

We cover the steps to attaching a debugger to the process later in this chapter, in the section “Debugging a Windows Service.” In Exercise 1.2, you will create a simple Windows service application. The steps for creating a setup project that will perform the installation of the service are included in the exercise. For a full explanation of creating setup and deployment projects, see Chapter 10.

For practical purposes, in real-world Windows service applications, you will probably want to create a Console or Windows Forms application to interactively test specific program logic before you add the code to your Windows service. After you are satisfied that your test code is working correctly, you can add it to the methods of your Windows service project.

For your first Windows service, you are going to design a simple service that uses a custom event log to record information about when the service is started and stopped.

You will create a new Windows service application project called CustomLogService. Next you will change some properties of the component. You will also add EventLog and Installer components from the Toolbox to the project. Finally, you are going to add code to the OnStart and OnStop events and also to the constructor method, Sub New.

**EXERCISE 1.2**

Creating a Windows Service by Using Visual Studio .NET

Setting Up the Project:

1. Start Visual Studio .NET and create a new project by using the Windows Service project template. Name the project CustomLogService and select an appropriate directory on your computer.

2. Using the Solution Explorer, rename the component Service1.vb to CustomLogService.vb.

3. Click on the design surface of CustomLogService.vb and display the Properties window. Change the Name property and the Service name property to CustomLogService. Change the AutoLog property to False. Verify that the CanStop property is set to True.
4. Display the Visual Studio .NET Toolbox and click the Components tab. Drag an EventLog component onto the design surface.

5. Click the EventLog component and display the Properties window. Change the name to CustomEventLog.

Adding Code:

6. Open the code editor for CustomLogService.vb. Verify that the class is named CustomLogService and that it inherits from System.ServiceProcess.ServiceBase:

   Public Class CustomLogService
   Inherits System.ServiceProcess.ServiceBase

7. Expand the region titled Component Designer Generated Code. Add code to the New procedure. Code to initialize the custom event log is placed in the New procedure, instead of OnStart, because you want this code to run only when the Windows service is first installed, rather than each time it is restarted. Your completed code should look like this:

   Public Sub New()
   MyBase.New()
   ' This call is required by the Component Designer.
   InitializeComponent()
   ' Add any initialization after InitializeComponent()
   If Not EventLog.SourceExists("CustomSource") Then
Add code to the OnStart and OnStop event procedures. Here you will write an entry to the custom event log to keep track of when the service is stopped and started. Your code should look like this:

```vbnet
Protected Overrides Sub OnStart(ByVal args() As String)
    CustomEventLog.WriteEntry("The service has been started.")
End Sub

Protected Overrides Sub OnStop()
    CustomEventLog.WriteEntry("The service has been stopped.")
End Sub
```

### Adding Installer Components:

9. Click the design surface of CustomLogService.vb and display the Properties window. Near the bottom of the Properties window is a link titled Add Installer. Click this link, and a new **component class module** called ProjectInstaller.vb will be added to your project. You will see that the design surface for this component has two other component icons on it: ServiceProcessInstaller1 and ServiceInstaller1.
10. Click ServiceProcessInstaller1 and display the Properties window. Select the Account property. Choose LocalSystem from the drop-down list. (If you decide to have your service running under a user account, you would also fill in the necessary information in the Password and Username properties here.)

11. Click ServiceInstaller1 and display the Properties window. Select the StartType property. Choose Automatic from the drop-down list.

**Building the Service:**

Before you can build the service, you need to clean up some details.

12. Display the Task List window by choosing View ➤ Other Windows ➤ Task List from the menu. You will most likely see two errors; the first one says “Type Service1 is not defined.” There is a remaining reference to the default name Service1.

13. Double-click the first entry in the Task List window, the code editor window will display the section of code where the error is located and the line of code that is in error will be highlighted. Change Service1 to CustomLogService.

14. The next item in the Task List says “‘Sub Main’ was not found in ‘CustomLogService.Service1’”. This refers to the project Startup Object. Double-click this entry in the Task List window, and a dialog box pops up showing the new correct reference to CustomLogService.CustomLogService. Select this item and click OK.

15. Now you can build the CustomLogService. Right-click the project name in the Solution Explorer and choose Build, or choose Build ➤ Build CustomLogService from the menu.

16. Save the CustomLogService project. You will be using it for future exercises.

**Creating a Setup Project to Install the Service:**

Many details are involved in creating a setup project and deploying Windows service applications. This topic is covered in more detail in Chapter 10. The following instructions are designed to get your new application up and running quickly so you can test it.

17. In the Solution Explorer, click on the solution. Choose File ➤ Add Project ➤ New Project from the Visual Studio menu.

18. In the Add New Project dialog box, select Setup and Deployment Projects and select the Setup Project template. Name the new project CustomLogSetup. Click OK.

19. In the Solution Explorer, right-click CustomLogSetup. Choose Add ➤ Project Output from the menu. The Add Project Output Group dialog box displays. Select Primary Output and click OK.
20. In the Solution Explorer, right-click CustomLogSetup again. Choose View ➤ Custom Actions from the menu.

21. In the upper-left corner of the work area, right-click Custom Actions. Choose Add Custom Action. The Select Item in Project dialog box displays. Double-click Application Folder, select Primary Output from CustomLogService (Active), and click OK. Your screen should look like the following one.
22. Build the setup project. Right-click the CustomLogSetup project name in the Solution Explorer and choose Build, or choose the menu command Build ➤ Build CustomLogSetup.

23. Save the CustomLogSetup project, because you will be using it again later in this chapter.

Installing and Testing the Service:

24. In the Debug subdirectory of the CustomLogSetup project directory, you will find a Windows Installer file named CustomLogSetup.msi. Double-click this file to start the installation.

25. This will start a Setup Wizard. Accept all the defaults and complete the installation.

26. Run the Service Control Manager to verify that your service is installed. To do this, click Start ➤ Programs ➤ Administrative Tools ➤ Services (or the appropriate sequence for your operating system version). You should see CustomLogService in the list.

27. Right-click on your service and choose Properties. Start your service.

28. Click Start ➤ Programs ➤ Administrative Tools ➤ Event Viewer (or the appropriate sequence for your operating system version) to view your custom event log in the Event Viewer.

29. Click the log named CustomLog. Then right-click any one of the log entries and choose Properties (or just double-click the entry). You will see your custom message in the Properties dialog box.
Debugging a Windows Service

Now that your service is installed and running, you can use the Visual Studio .NET debugger to attach to the service and use the standard debugging tools, such as breakpoints, stepping through code and others, to make sure your service is running correctly.

In Exercise 1.3, you will be attaching the debugger to a Windows service. You will be using a special capability of the Visual Studio .NET debugger that enables you to attach the debugger to an external process running on the computer. Because you have access to the source code for your service, you can set breakpoints. While the service is running, when a breakpoint is hit, you will go into break mode and can step through the code to examine variable values and perform other debugging actions.

EXERCISE 1.3

**EXERCISE 1.3**

**Debugging a Windows Service**

1. In Visual Studio .NET, open the CustomLogService project. Right-click CustomLogService.vb in the Solution Explorer and choose View Code.
2. Set a breakpoint on the line of code in the OnStop procedure that writes the log entry:

   CustomEventLog.WriteEntry("The service has been stopped.")

3. From the Visual Studio .NET menus, choose Debug ➤ Processes. You will see a list of running processes on your computer. Make sure that the check boxes labeled Show System Processes and Show Processes in all Sessions are both selected.

   ![Processes Dialog Box](image)

4. Select CustomLogService and click the Attach button.

5. The Attach to Process dialog box displays. Make sure that the Common Language Runtime option is checked and click OK. Close the Processes dialog box.

6. Back in Control Panel, start the Service Control Manager. Select CustomLogService and stop the service. A yellow highlight in Visual Studio .NET indicates that the breakpoint has been hit. The Service Control Manager will not be able to finish stopping the service until you release the debugger. Choose Debug ➤ Stop Debugging from the menu to do so.

---

**EXERCISE 1.3 (continued)**

---

**WARNING**

Be careful when using the debugger to attach to a process. Use this technique only when you are working with processes that you can control. Attaching a debugger to one of the operating system processes, for example, could cause your computer to hang up.
Configuring Client Computers and Servers to Use a Windows Service

Until now, you have been using the built-in Windows tools to view and manage Windows services. The .NET Framework also provides a set of classes that enable you to work with Windows services directly from your Visual Basic .NET application code. This can be very useful if you have created a Windows service that monitors and logs some system performance data, but you want it to run only while your application is running. You can start the service when your application starts up and stop it when your application closes. You can even add custom commands to your service and call them from application code.

In this section, you are going to learn how to use the `ServiceController` class. This is a .NET Framework class that has methods to programmatically control a Windows service. You will create a Windows Forms application that can start and stop services. A sample application called `ServiceControllerProject` is included on the book’s CD and incorporates all the features covered in this section (see Figure 1.3). You might want to load the application code so that you can review it while you are reading this section.

**FIGURE 1.3** The `ServiceControllerProject` demo

Exercise 1.4 at the end of the chapter is designed to take you step-by-step through the features of the `ServiceControllerProject` demo application. Exercise 1.5 provides some examples that modify the `CustomLogService` that you created earlier in the chapter to support custom commands and for building a service controller application of your own to test them.
Instantiating the **ServiceController** Object

When you instantiate a `ServiceController` object, you must supply two important pieces of information:

- The service name that you want to control
- The machine name that the service is running on

If you do not specify a machine name, the default is to look for the service on the local machine. Your project must include a reference to `System.ServiceProcess.dll`, and you should add an `Imports` statement for `System.ServiceProcess` as well. The `ServiceController` object can be instantiated as follows:

```vbnet
Dim servController = New ServiceController("CustomLogService")
```

In the preceding example, the service name is passed to the overloaded constructor method as a single string parameter. The `ServiceName` property can also be set independently, as shown here:

```vbnet
Dim servController as New ServiceController()
servController.ServiceName = "CustomLogService"
```

Properties and Methods of the **ServiceController** Class

There are several important properties of the service that you might be interested in testing. Table 1.3 shows some of the properties of the `ServiceController` class. The listed properties map to the properties of the `ServiceBase` class discussed in the first part of this chapter.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanPauseAndContinue</td>
<td>True if the service can be paused and continued. (Read-only.)</td>
</tr>
<tr>
<td>CanShutdown</td>
<td>True if the service should be notified when the system is shutting down. (Read-only.)</td>
</tr>
<tr>
<td>CanStop</td>
<td>True if the service can be stopped after it has started. (Read-only.)</td>
</tr>
<tr>
<td>DependentServices</td>
<td>Gets the set of services that depends on the service associated with this <code>ServiceController</code> instance.</td>
</tr>
<tr>
<td>DisplayName</td>
<td>A friendly name for the service.</td>
</tr>
<tr>
<td>MachineName</td>
<td>The name of the computer on which the service is running.</td>
</tr>
<tr>
<td>ServiceName</td>
<td>Identifies the service that this instance of the <code>ServiceController</code> references.</td>
</tr>
<tr>
<td>ServicesDependedOn</td>
<td>The set of services that this service depends on.</td>
</tr>
</tbody>
</table>
Remember, you use the properties of the `ServiceBase` class when you are creating a Windows service. The `CanStop`, `CanPauseAndContinue`, and `CanShutdown` properties of the `ServiceBase` class enable you to set the behavior for your service. In the `ServiceController` class, these properties are read-only. The `ServiceController` instance can only test the property to see what was set when the service was created.

When you are working programmatically with a service, it is good practice to always test the service’s state before you try an operation. For example, before you try to issue a `Pause` command to a service, test the `CanPauseAndContinue` property to see whether `Pause` is a valid action for that particular service:

```vbnet
If servController.CanPauseAndContinue = True Then
    servController.Pause()
End If
```

You also might want to test the current value of the `Status` property before issuing a command to change the status:

```vbnet
If servController.Status = _
    ServiceControllerStatus.Paused Then
    servController.Continue()
End If
```

The only valid settings for the `Status` property are defined by the `ServiceControllerStatus enumeration`, as Intellisense in Visual Studio .NET will show you (see Figure 1.4).

**FIGURE 1.4** The `ServiceControllerStatus enumeration`
Table 1.4 lists the methods of the `ServiceController` class. These methods enable you to write code in a Visual Basic .NET application that can cause a Windows service application to start, stop, pause, or continue. Your code can also call custom commands and get other information about the service.

**Table 1.4** Methods of the `ServiceController` Class

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close</td>
<td>Disconnects the <code>ServiceController</code> object from the service and releases any resources that were in use.</td>
</tr>
<tr>
<td>Continue</td>
<td>Resumes a service after a paused command.</td>
</tr>
<tr>
<td>ExecuteCommand</td>
<td>Executes a custom command on the service.</td>
</tr>
<tr>
<td>GetDevices</td>
<td>Gets a list of device driver services on a computer.</td>
</tr>
<tr>
<td>GetServices</td>
<td>Gets a list of services on a computer.</td>
</tr>
<tr>
<td>Pause</td>
<td>Pauses the service.</td>
</tr>
<tr>
<td>Refresh</td>
<td>Gets current property values.</td>
</tr>
<tr>
<td>Start</td>
<td>Starts the service.</td>
</tr>
<tr>
<td>Stop</td>
<td>Stops this service and any services that are dependent on this service.</td>
</tr>
<tr>
<td>WaitForStatus</td>
<td>Waits for the service to reach the specified status or for the request to time out.</td>
</tr>
</tbody>
</table>

The `Start`, `Stop`, `Pause`, and `Continue` methods are easy to understand. They work the same way in code that they work when you are issuing these commands through the Service Control Manager interface.

The `Refresh` method gets the current settings for the properties of the service that you are monitoring, without affecting the state of the service.

The `GetServices` and `GetDevices` methods populate an array of `ServiceController` objects, which in turn can access information about all the services installed on a computer (as shown in Listing 1.3). The `GetDevices` method gets those services that are of type `KernelDriver` or `FileSystemDriver`.

Listing 1.3 shows the procedure from the `ServiceControllerProject` demo that loads a `ListBox` control with the names of all services on the computer.
Listing 1.3: A Procedure to List All Services Running on the Local Computer

Private Sub btnGetServices_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnGetServices.Click
    Dim servArray() As ServiceController
    Dim i As Integer
    servArray = ServiceController.GetServices()
    lstDisplay.Items.Clear()
    For i = 0 To servArray.Length - 1
        lstDisplay.Items.Add(servArray(i).ServiceName)
    Next
    servArray = Nothing
End Sub

The `WaitForStatus` method takes into consideration that sometimes a particular service might take a long time to start or not start at all. Also, `StartPending`, `StopPending`, `PausePending`, and `ContinuePending` will appear as the service's status briefly, before they have completely reached a final state. You can test this with the `ServiceControllerProject` demo. After a service is stopped, click the Start button. The display in the list box will show the status as `StartPending`. If you click the Get Properties button again a moment later, the display updates to show that the service now has a status of `Running`. If you select the check box labeled Wait Until Running, the code in the `ServiceControllerProject` demo will call the `ServiceController` object's `WaitForStatus` method and the code will block until the target service achieves the specified status. The display does not update until the service's status is `Running`. This is shown in the following code snippet:

If chkWait.Checked Then
    servController.WaitForStatus( ServiceControllerStatus.Running)
End If

You can also call the `WaitForStatus` method by specifying two parameters: the status to wait for and a `TimeSpan` value, which indicates how long your code should wait before it times out and reports an error condition.

Executing Custom Commands for a Service

The `ServiceController` class offers a method that enables you to define truly customized functionality for your Windows service application. You have seen how to add code to standard methods that will fire in the normal cycle of events, as a Windows service application is
started and stopped. The ServiceController class provides a means to call custom methods that you have designed for your Windows service application.

Let's return to the source code for your Windows service application named CustomLogService. You will add another event procedure to the service and then recompile and reinstall it.

When you create custom functionality for a Windows service, calls to any of your procedures are handled inside the single Windows service event procedure named OnCustomCommand. Inside this procedure, you can use a conditional test or Case statement to break out one or more groups of code that will be executed as part of a given command. The OnCustomCommand method accepts an integer parameter that indicates which section of code should be executed for any specific call to the method. The integer parameter must be within the range of 128 and 256. Values below 128 are reserved for system commands. If the AutoLog property of the service is True, calls to OnCustomCommand will be noted in the Windows Application event log.

The procedure inside your Windows Service application will look like Listing 1.4.

Listing 1.4: The OnCustomCommand Procedure
Protected Overrides Sub OnCustomCommand( _
    ByVal command As Integer)
    Select Case command
        Case 130
            CustomEventLog.WriteEntry( _
                "Command 130 successfully completed.")
        Case 140
            CustomEventLog.WriteEntry( _
                "Command 140 successfully completed.")
        Case 150
            CustomEventLog.WriteEntry( _
                "Command 150 successfully completed.")
        Case Else
            CustomEventLog.WriteEntry( _
                "ERROR: Unrecognized command parameter!")
    End Select
End Sub

For simplicity, your custom command does nothing more than write a log entry to verify that the command successfully completed. But that's enough to test your code in the ServiceControllerProject demo.

After you have the code in the Windows service application, you can write a method in your ServiceController application that calls the ServiceController.ExecuteCommand method. The ServiceControllerProject demo has a simple user interface that calls the method and passes a user-selected integer parameter (see Figure 1.5). As you can see from Listing 1.4, the OnCustomCommand method will recognize three valid parameter values: 130, 140, and 150. If any other value is passed, an error message will be written to the custom event log.

Listing 1.5 shows the code from the ServiceControllerProject demo that calls the Execute command method.
Configuring Client Computers and Servers to Use a Windows Service

Figure 1.5 Executing a custom command from the ServiceControllerProject demo

Listing 1.5: Executing a Custom Command
Private Sub btnCommand_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCommand.Click
    Dim commandNumber As Integer
    servController = New ServiceController("CustomLogService")
    Try
        commandNumber = CType(txtCommand.Text, Integer)
        servController.ExecuteCommand(commandNumber)
        MessageBox.Show("Command completed. Check the Custom event log.")
    Catch ex As Exception
        MessageBox.Show("Invalid command number.")
    End Try
End Sub

One important thing to remember about calling a custom command on a Windows service is that all error handling must be done within the Windows service application itself. In this simple example, our “error handling” consisted of writing an error message to the event log. In a real-world application, you will need to consider your error handling carefully. The error handling implemented in the ServiceController client application guards only against sending a nonnumeric value as a parameter to the call to ExecuteCommand.

In Exercise 1.4, you will load the ServiceControllerProject demo and try some of its features that were discussed in this section.
Chapter 1 • Creating and Managing Windows Services

EXERCISE 1.4

Trying the ServiceController Demo Project

On the CD included with this book, you will find a Visual Basic .NET project titled ServiceControllerProject. Open this project in Visual Studio .NET.

1. If you have already created the CustomLogService (see Exercise 1.2), the ServiceControllerProject will immediately display information about the service when you first run it. If you do not have a service named CustomLogService installed, you will get an error message. If you get this message, type in the name of a valid service, such as ClipBook. Then click the Get Properties button.

2. Experiment with the Stop and Start buttons. You might also want to open the Service Control Manager and watch the service status changing there as well. You will need to refresh the display in the Service Control Manager each time you change the status by using the Visual Basic .NET application.

Because the CanPauseAndContinue property of CustomLogService is set to False, the Pause and Continue buttons are disabled.

3. Notice that when you stop and then start the CustomLogService, the status that is displayed is StartPending. If you click the Get Properties button again a few seconds later, you will see the status is now Running.

4. Select the Wait Until Running check box; then stop and start the service again. This time the ListBox display will not be updated until the service has been fully started and the status has reached Running.

5. Type in the name of a different service, such as EventLog, and view its properties. Remember, do not stop the system services or services you didn’t create (especially if you’re not sure what the service does); doing so can cause problems with your computer.

6. The Service Lists menu displays another form, where you can see a list of all the services installed on your computer. The GetDevices method shows all installed services that are device drivers.

7. Finally, the Execute Commands menu displays one more form. This form contains code to execute custom commands against CustomLogService. You can’t test this feature yet. In the next exercise, Exercise 1.5, you will modify CustomLogService to accept custom commands.

In Exercise 1.5, you will uninstall and modify the CustomLogService you created in Exercise 1.2. To uninstall the CustomLogService, you will be using the Windows Control Panel application Add/Remove Programs. While looking at the list of installed applications on your computer, you will see only the entry for the setup program that installs the service. You will not see an entry for the service itself.
EXERCISE 1.5

Uninstalling and Modifying **CustomLogService**


2. Verify that CustomLogService is no longer installed by checking the Service Control Manager. Use Control Panel to open the Service Control Manager.

3. Open the CustomLogService solution in Visual Studio .NET (it should contain both the service and setup projects).

4. Add the following method to CustomLogService.vb. Add this code directly after the OnStart and OnStop methods (refer to the following screen capture):

   Protected Overrides Sub OnCustomCommand(ByVal command As Integer)
       Select Case command
           Case 130
               CustomEventLog.WriteEntry(_
                   "Command 130 successfully completed.")
           Case 140
               CustomEventLog.WriteEntry(_
                   "Command 140 successfully completed.")
           Case 150
               CustomEventLog.WriteEntry(_
                   "Command 150 successfully completed.")
           Case Else
               CustomEventLog.WriteEntry(_
                   "Unrecognized command parameter!")
       End Select
   End Sub
Chapter 1 • Creating and Managing Windows Services

EXERCISE 1.5 (continued)

5. Save the solution. Right-click the CustomLogService project in the Solution Explorer and chose Build. Then right-click the CustomLogSetup project and choose Build.

6. Go to the Debug subdirectory under the CustomLogSetup project directory. Double-click the CustomLogSetup.msi file to install the revised version of the service.

7. Use the ServiceControllerProject demo that you used in Exercise 1.4 to verify that the CustomLogService is once again installed on your computer.

8. Start the CustomLogService.

9. Go to the Execute Commands form in the ServiceControllerProject demo and test execution of the custom commands. Try the valid parameter numbers 130, 140, and 150 and then try an invalid number, such as 155.

10. Open the Windows Event Viewer and look at the entries. Double-click an entry to display the Properties dialog box and view the message.

Summary

In this chapter, you learned about creating and managing Windows service applications. We covered the following topics:

- An introduction to how Windows services work
- How to view existing Windows services by using system tools such as the Service Control Manager and the Event Viewer
- How Visual Studio .NET helps you to create the foundations of a Windows service application
- The properties and methods of the .NET Framework ServiceBase class
- How to use Visual Studio .NET to add .NET Framework components, such as the Project Installers and an EventLog, to your project directly from the Toolbox
- How to add custom code to the OnStart and OnStop methods of a Windows service
- How to attach the Visual Studio .NET debugger to a running Windows service
- How to use the .NET Framework ServiceController class to control a Windows service from application code
- How to code custom commands for a Windows service and how to call them from a ServiceController object
Exam Essentials

**Know how to create a Windows service.** Visual Studio .NET offers you a built-in template that makes setting up a Windows service easy. Windows service applications inherit from the `System.ServiceProcess.ServiceBase` namespace.

**Be familiar with the properties and methods of the** `System.ServiceProcess.ServiceBase` **class.** Know how the code in the `Sub Main` method of a Windows service calls the `Run` method to instantiate the service.

**Understand the security accounts that can be used with Windows services.** LocalSystem is currently the most commonly used security setting but it is a highly privileged account, which could lead to security breaches. Windows XP (and later) offers the opportunity to use accounts with lesser privileges: LocalService and NetworkService.

**Understand that you cannot directly run a Windows service from Visual Studio .NET.** You must attach the debugger to the running process.

**Know how to manipulate a Windows service application.** Know how to use the Windows utility Service Control Manager to manipulate a Windows service. Be familiar with the properties and methods of the `System.ServiceProcess.ServiceController` class. Know how to use the `ServiceController` to stop and start Windows services programmatically.

Key Terms

Before you take the exam, be certain you are familiar with the following terms:

- **AutoLog**
- **event log**
- **LocalSystem**
- **OnStart**
- **OnStop**
- **security account**
- **Server Explorer**
- **Service Control Manager**

- **ServiceBase**
- **ServiceController**
- **ServiceControllerStatus**
- **ServiceInstaller**
- **ServiceProcessInstaller**
- **System.ServiceProcess**
- **Windows service**
Review Questions

1. How can you best describe a Windows service application?
   A. It impersonates the identity of the user who is logged in.
   B. It runs in its own process with its own security account.
   C. It runs in the same process space as the web server with the identity of IUSR_Machine.
   D. It runs in the same process space as the operating system and must have Administrator privileges.

2. Windows services begin running:
   A. When the computer is booted, if the Startup type is set to Automatic
   B. When a user logs in
   C. Only when an Administrator starts them
   D. Only when called by a ServiceController object

3. How can you view information about the services running on a specific computer?
   A. By using the Server Explorer in Visual Studio .NET
   B. By using a method of the ServiceController class
   C. By using the Windows Service Control Manager console
   D. All of the above

4. Your Windows service needs to read some default values from a disk file every time it is started. How can you accomplish this?
   A. Write code in the OnStart method of your service application.
   B. Write code in the OnCustomCommand method of your service application.
   C. Write code in the Sub Main method of your service application.
   D. Write code in the Sub New method of your service application.

5. All Windows service applications support the same basic interface, because:
   A. The operating system will not load them if they do not implement all standard methods.
   B. They will not compile if they do not implement all standard methods.
   C. They all inherit from the System.ServiceProcess.ServiceInstaller class.
   D. They all inherit from the System.ServiceProcess.ServiceBase class.
Review Questions

6. If you leave the AutoLog property set to the default value of True in your Windows service, what behavior will you see when the service is running?
   A. Stop, Start, Pause, and Continue events will be written to a custom event log with the same name as your service.
   B. Stop, Start, Pause, and Continue events will be written to the Windows Application event log.
   C. No logging will take place unless you set the EventLog property to the name of a custom event log.
   D. No logging will take place unless you write code in the OnStart method to write entries to a custom event log.

7. You create a Windows service project that includes two Windows services. When you add installer components to your project, how many objects will be added?
   A. One ServiceInstaller object
   B. One ServiceInstaller object and one ServiceProcessInstaller object
   C. One ServiceInstaller object and two ServiceProcessInstaller objects
   D. Two ServiceInstaller objects and two ServiceProcessInstaller objects

8. You need to specify the security account that your Windows service will run under. How can you specify this while creating the project in Visual Studio .NET?
   A. Set the Account property of the ServiceBase class.
   B. Set the Account property of the ServiceProcessInstaller object.
   C. Set the Account property of the ServiceInstaller object.
   D. Change the Account setting in the Project Properties dialog box.

9. What is the most commonly used security account for running Windows services?
   A. Interactive User
   B. LocalSystem
   C. Administrator
   D. NetworkService

10. You have created a Windows service application and you would like to use the debugging tools in Visual Studio .NET to troubleshoot a problem with the application. You load the application in Visual Studio .NET. What should you do next?
    A. Select Debug ➤ Start from the Visual Studio .NET menu.
    B. Select Debug ➤ Step Into from the Visual Studio .NET menu.
    C. Select Debug ➤ Processes from the Visual Studio .NET menu.
    D. Select Debug ➤ Exceptions from the Visual Studio .NET menu.
11. You need to create an application that is able programmatically to start and stop a Windows service. Which .NET Framework class should you use?
   A. System.ServiceProcess.ServiceBase
   B. System.ServiceProcess.ServiceController
   C. System.ServiceProcess.ServiceInstaller
   D. System.ServiceProcess.Status

12. You are creating an application that controls a Windows service programmatically. You would like to be able to call the Pause method to temporarily disable the service while your application is running, but this does not seem to be working. What can you do to overcome this problem?
   A. You must set the CanPauseAndContinue property of the ServiceController to True before you can call the Pause method.
   B. You must set the CanStop property of the ServiceController to True before you can call the Pause method.
   C. Nothing. You cannot use the Pause method if the original designer of the Windows service did not set the CanShutdown property to True.
   D. Nothing. You cannot use the Pause method if the original designer of the Windows service did not set the CanPauseAndContinue property of the ServiceBase class to True.

13. You have created an application that is able to programmatically start and stop a Windows service. However, after using the Start method, your application always reports back that the service’s status is StartPending rather than the Running status that you are looking for. How can you be sure that the service has successfully been started and is running, before your application takes any further action?
   A. Use the GetService method and see whether your service is included in the array of services that is returned.
   B. Set a Timer control to call the Refresh method until a status of Running is returned.
   C. Use the WaitforStatus method with ServiceControllerStatus.Running as the parameter.
   D. Use the ExecuteCommand method to run custom code when the service starts.

14. You need to create an application that is able to programmatically execute custom commands of a Windows service. How do you call custom commands from your application?
   A. Use the ServiceBase class OnCustomCommand method and pass an integer parameter.
   B. Use the ServiceController class OnCustomCommand method and pass a string parameter.
   C. Use the ServiceBase class ExecuteCommand method and pass a string parameter.
   D. Use the ServiceController class ExecuteCommand method and pass an integer parameter.
15. What does the `OnPowerEvent` method of the `ServiceBase` class do?

   A. Enables the designer of the Windows service to write code that will run in the event of a power outage
   B. Enables the designer of the Windows service to write code that will run when the computer shuts down
   C. Enables the designer of the Windows service to write code that will run when a laptop computer goes into suspended mode
   D. Enables the designer of the Windows service to write code that will run when a custom command is executed
Answers to Review Questions

1. B. A Windows service runs in its own memory process space and has its own security account, most commonly LocalSystem. A Windows service does not interfere with other users or programs running on the computer.

2. A. If the service’s StartUpType property is set to Automatic, the service will be started when the computer is started or rebooted. If the StartUpType property is set to Manual, then the service must be started by using either the Service Control Manager console or by application code that uses a ServiceController object.

3. D. You can view information about Windows services by using either the Windows Service Control Manager console or the Visual Studio .NET Server Explorer. The GetServices and GetDevices methods of the ServiceController class also provide information about the services that are running on a specific computer.

4. A. The OnStart method is the recommended place to put code that should run when a service is started. Code in the constructor, Sub New, might not run when a service is stopped and restarted.

5. D. To create an application that will run as a Windows service in Visual Studio .NET, you must inherit base class functionality from the System.ServiceProcess.ServiceBase class.

6. B. If the AutoLog property of a Windows service application is set to True, Stop, Start, Pause, and Continue events will be written to the Windows Application event log without any further coding necessary.

7. C. A Windows service project in Visual Studio .NET can contain more than one Windows service component class module. When you add installers to the project, one ServiceInstaller object will be added, and one ServiceProcessInstaller object will be added for each Windows service module contained in the project.

8. B. Use the Account property of the ServiceProcessInstaller object to specify which security account the service should run under.

9. B. LocalSystem is currently the most commonly used security account for running Windows service applications. It is a highly privileged account, which can pose a security risk. The new Windows XP accounts, NetworkService and LocalService, might be better choices from a security standpoint.

10. C. To debug a Windows service application, you must install and run it. After it is running, you can attach the Visual Studio .NET debugger to this external process. Use the Debug ➤ Processes menu choice to display the Processes dialog box to choose from all running processes on the computer.

11. B. The System.ServiceProcess.ServiceController .NET Framework class has properties and methods that enable you to get information about a Windows service and to control the service through application code.
12. D. The original creator of the Windows service application sets the CanStop and CanPauseAndContinue properties of the service. The original designer might not want the service to be stopped or paused by a user, as is often the case with operating system services.

13. C. The WaitForStatus method of the ServiceController class will cause application code to block until the desired status is reached.

14. D. To call a custom command from an application that can control Windows services programmatically, use the ServiceController.ExecuteCommand method. This method takes a single integer parameter, which indicates the command that the user would like to run. Valid parameter values are defined by the designer of the Windows service application (within the range of 128 to 256).

15. C. The OnPowerEvent method is intended to be used if your service must run on laptop computers. You might want to save data, for example, before the computer goes into suspended mode.