

Developing Windows Service Applications _____	1
Introduction to Windows Service Applications _____	3
Walkthrough Creating a Windows Service Application in the Component Desig	6
Service Application Programming Architecture _____	18
How to Create Windows Services _____	21
How to Write Services Programmatically _____	24
How to Add Installers to Your Service _____	26
How to Specify the Security Context for Services _____	28
How to Install and Uninstall Services _____	30
How to Start Services _____	32
How to Pause a Windows Service _____	35
How to Continue a Windows Service _____	37
How to Debug Windows Service Applications _____	39
How to Log Information About Services _____	43
Troubleshooting Debugging Windows Services _____	46
Troubleshooting Service Application Won't Install _____	47

# Developing Windows Service Applications

## .NET Framework (current version)

Using Microsoft Visual Studio or the Microsoft .NET Framework SDK, you can easily create services by creating an application that is installed as a service. This type of application is called a Windows service. With framework features, you can create services, install them, and start, stop, and otherwise control their behavior.

### Warning

The Windows service template for C++ was not included in Visual Studio 2010. To create a Windows service, you can either create a service in managed code in Visual C# or Visual Basic, which could interoperate with existing C++ code if required, or you can create a Windows service in native C++ by using the [ATL Project Wizard](#).

## In This Section

### [Introduction to Windows Service Applications](#)

Provides an overview of Windows service applications, the lifetime of a service, and how service applications differ from other common project types.

### [Walkthrough: Creating a Windows Service Application in the Component Designer](#)

Provides an example of creating a service in Visual Basic and Visual C#.

### [Service Application Programming Architecture](#)

Explains the language elements used in service programming.

### [How to: Create Windows Services](#)

Describes the process of creating and configuring Windows services using the Windows service project template.

## Related Sections

### [ServiceBase](#)

Describes the major features of the [ServiceBase](#) class, which is used to create services.

### [ServiceProcessInstaller](#)

Describes the features of the [ServiceProcessInstaller](#) class, which is used along with the [ServiceInstaller](#) class to install and uninstall your services.

### [ServiceInstaller](#)

Describes the features of the [ServiceInstaller](#) class, which is used along with the [ServiceProcessInstaller](#) class to install and uninstall your service.

### [NIB Creating Projects from Templates](#)

Describes the projects types used in this chapter and how to choose between them.

© 2016 Microsoft

# Introduction to Windows Service Applications

## .NET Framework (current version)

Microsoft Windows services, formerly known as NT services, enable you to create long-running executable applications that run in their own Windows sessions. These services can be automatically started when the computer boots, can be paused and restarted, and do not show any user interface. These features make services ideal for use on a server or whenever you need long-running functionality that does not interfere with other users who are working on the same computer. You can also run services in the security context of a specific user account that is different from the logged-on user or the default computer account. For more information about services and Windows sessions, see the Windows SDK documentation in the MSDN Library.

You can easily create services by creating an application that is installed as a service. For example, suppose you want to monitor performance counter data and react to threshold values. You could write a Windows Service application that listens to the performance counter data, deploy the application, and begin collecting and analyzing data.

You create your service as a Microsoft Visual Studio project, defining code within it that controls what commands can be sent to the service and what actions should be taken when those commands are received. Commands that can be sent to a service include starting, pausing, resuming, and stopping the service; you can also execute custom commands.

After you create and build the application, you can install it by running the command-line utility `InstallUtil.exe` and passing the path to the service's executable file. You can then use the **Services Control Manager** to start, stop, pause, resume, and configure your service. You can also accomplish many of these same tasks in the **Services** node in **Server Explorer** or by using the [ServiceController](#) class.

## Service Applications vs. Other Visual Studio Applications

Service applications function differently from many other project types in several ways:

- The compiled executable file that a service application project creates must be installed on the server before the project can function in a meaningful way. You cannot debug or run a service application by pressing F5 or F11; you cannot immediately run a service or step into its code. Instead, you must install and start your service, and then attach a debugger to the service's process. For more information, see [How to: Debug Windows Service Applications](#).
- Unlike some types of projects, you must create installation components for service applications. The installation components install and register the service on the server and create an entry for your service with the Windows **Services Control Manager**. For more information, see [How to: Add Installers to Your Service Application](#).
- The `Main` method for your service application must issue the `Run` command for the services your project contains. The `Run` method loads the services into the **Services Control Manager** on the appropriate server. If you use the **Windows Services** project template, this method is written for you automatically. Note that loading a service is not the same thing as starting the service. See "Service Lifetime" below for more information.
- Windows Service applications run in a different window station than the interactive station of the logged-on user.

A window station is a secure object that contains a Clipboard, a set of global atoms, and a group of desktop objects. Because the station of the Windows service is not an interactive station, dialog boxes raised from within a Windows service application will not be seen and may cause your program to stop responding. Similarly, error messages should be logged in the Windows event log rather than raised in the user interface.

The Windows service classes supported by the .NET Framework do not support interaction with interactive stations, that is, the logged-on user. The .NET Framework also does not include classes that represent stations and desktops. If your Windows service must interact with other stations, you will need to access the unmanaged Windows API. For more information, see the Windows SDK documentation.

The interaction of the Windows service with the user or other stations must be carefully designed to include scenarios such as there being no logged on user, or the user having an unexpected set of desktop objects. In some cases, it may be more appropriate to write a Windows application that runs under the control of the user.

- Windows service applications run in their own security context and are started before the user logs into the Windows computer on which they are installed. You should plan carefully what user account to run the service within; a service running under the system account has more permissions and privileges than a user account.

## Service Lifetime

A service goes through several internal states in its lifetime. First, the service is installed onto the system on which it will run. This process executes the installers for the service project and loads the service into the **Services Control Manager** for that computer. The **Services Control Manager** is the central utility provided by Windows to administer services.

After the service has been loaded, it must be started. Starting the service allows it to begin functioning. You can start a service from the **Services Control Manager**, from **Server Explorer**, or from code by calling the [Start](#) method. The [Start](#) method passes processing to the application's [OnStart](#) method and processes any code you have defined there.

A running service can exist in this state indefinitely until it is either stopped or paused or until the computer shuts down. A service can exist in one of three basic states: [Running](#), [Paused](#), or [Stopped](#). The service can also report the state of a pending command: [ContinuePending](#), [PausePending](#), [StartPending](#), or [StopPending](#). These statuses indicate that a command has been issued, such as a command to pause a running service, but has not been carried out yet. You can query the [Status](#) to determine what state a service is in, or use the [WaitForStatus](#) to carry out an action when any of these states occurs.

You can pause, stop, or resume a service from the **Services Control Manager**, from **Server Explorer**, or by calling methods in code. Each of these actions can call an associated procedure in the service ([OnStop](#), [OnPause](#), or [OnContinue](#)), in which you can define additional processing to be performed when the service changes state.

## Types of Services

There are two types of services you can create in Visual Studio using the .NET Framework. Services that are the only service in a process are assigned the type [Win32OwnProcess](#). Services that share a process with another service are assigned the type [Win32ShareProcess](#). You can retrieve the service type by querying the [ServiceType](#) property.

You might occasionally see other service types if you query existing services that were not created in Visual Studio. For more information on these, see the [ServiceType](#).

## Services and the ServiceController Component

The [ServiceController](#) component is used to connect to an installed service and manipulate its state; using a [ServiceController](#) component, you can start and stop a service, pause and continue its functioning, and send custom commands to a service. However, you do not need to use a [ServiceController](#) component when you create a service application. In fact, in most cases your [ServiceController](#) component should exist in a separate application from the Windows service application that defines your service.

For more information, see [ServiceController](#).

## Requirements

- Services must be created in a **Windows Service** application project or another .NET Framework–enabled project that creates an .exe file when built and inherits from the [ServiceBase](#) class.
- Projects containing Windows services must have installation components for the project and its services. This can be easily accomplished from the **Properties** window. For more information, see [How to: Add Installers to Your Service Application](#).

## See Also

[Developing Windows Service Applications](#)

[Service Application Programming Architecture](#)

[How to: Create Windows Services](#)

[How to: Install and Uninstall Services](#)

[How to: Start Services](#)

[How to: Debug Windows Service Applications](#)

[Walkthrough: Creating a Windows Service Application in the Component Designer](#)

[How to: Add Installers to Your Service Application](#)

# Walkthrough: Creating a Windows Service Application in the Component Designer

## .NET Framework (current version)

This article demonstrates how to create a simple Windows Service application in Visual Studio that writes messages to an event log. Here are the basic steps that you perform to create and use your service:

1. [Creating a Service](#) by using the **Windows Service** project template, and configure it. This template creates a class for you that inherits from [System.ServiceProcess.ServiceBase](#) and writes much of the basic service code, such as the code to start the service.
2. [Adding Features to the Service](#) for the [OnStart](#) and [OnStop](#) procedures, and override any other methods that you want to redefine.
3. [Setting Service Status](#). By default, services created with [System.ServiceProcess.ServiceBase](#) implement only a subset of the available status flags. If your service takes a long time to start up, pause, or stop, you can implement status values such as Start Pending or Stop Pending to indicate that it's working on an operation.
4. [Adding Installers to the Service](#) for your service application.
5. (Optional) [Set Startup Parameters](#), specify default startup arguments, and enable users to override default settings when they start your service manually.
6. [Building the Service](#).
7. [Installing the Service](#) on the local machine.
8. Access the Windows Service Control Manager and [Starting and Running the Service](#).
9. [Uninstalling a Windows Service](#).

### Warning

The Windows Services project template that is required for this walkthrough is not available in the Express edition of Visual Studio.

### Note

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the Visual Studio IDE](#).

## Creating a Service

To begin, you create the project and set values that are required for the service to function correctly.

### To create and configure your service

1. In Visual Studio, on the menu bar, choose **File, New, Project**.

The **New Project** dialog box opens.

2. In the list of Visual Basic or Visual C# project templates, choose **Windows Service**, and name the project **MyNewService**. Choose **OK**.

The project template automatically adds a component class named `Service1` that inherits from `System.ServiceProcess.ServiceBase`.

3. On the **Edit** menu, choose **Find and Replace, Find in Files** (Keyboard: Ctrl+Shift+F). Change all occurrences of `Service1` to `MyNewService`. You'll find instances in `Service1.cs`, `Program.cs`, and `Service1.Designer.cs` (or their `.vb` equivalents).
4. In the **Properties** window for **Service1.cs [Design]** or **Service1.vb [Design]**, set the `ServiceName` and the **(Name)** property for `Service1` to `MyNewService`, if it's not already set.
5. In Solution Explorer, rename **Service1.cs** to **MyNewService.cs**, or **Service1.vb** to **MyNewService.vb**.

## Adding Features to the Service

In this section, you add a custom event log to the Windows service. Event logs are not associated in any way with Windows services. Here the `EventLog` component is used as an example of the type of component you could add to a Windows service.

### To add custom event log functionality to your service

1. In **Solution Explorer**, open the context menu for **MyNewService.cs** or **MyNewService.vb**, and then choose **View Designer**.
2. From the **Components** section of the **Toolbox**, drag an `EventLog` component to the designer.
3. In **Solution Explorer**, open the context menu for **MyNewService.cs** or **MyNewService.vb**, and then choose **View Code**.
4. Add a declaration for the `eventLog` object in the `MyNewService` class, right after the line that declares the `components` variable:

**VB**

```
Private components As System.ComponentModel.IContainer  
Private EventLog1 As System.Diagnostics.EventLog
```



5. Add or edit the constructor to define a custom event log:

**VB**

```
' To access the constructor in Visual Basic, select New from the
' method name drop-down list.
Public Sub New()
    MyBase.New()
    InitializeComponent()
    Me.EventLog1 = New System.Diagnostics.EventLog
    If Not System.Diagnostics.EventLog.SourceExists("MySource") Then
        System.Diagnostics.EventLog.CreateEventSource("MySource",
            "MyNewLog")
    End If
    EventLog1.Source = "MySource"
    EventLog1.Log = "MyNewLog"
End Sub
```

## To define what occurs when the service starts

- In the Code Editor, locate the [OnStart](#) method that was automatically overridden when you created the project, and replace the code with the following. This adds an entry to the event log when the service starts running:

**VB**

```
' To access the OnStart in Visual Basic, select OnStart from the
' method name drop-down list.
Protected Overrides Sub OnStart(ByVal args() As String)
    EventLog1.WriteEntry("In OnStart")
End Sub
```

A service application is designed to be long-running, so it usually polls or monitors something in the system. The monitoring is set up in the [OnStart](#) method. However, [OnStart](#) doesn't actually do the monitoring. The [OnStart](#) method must return to the operating system after the service's operation has begun. It must not loop forever or block. To set up a simple polling mechanism, you can use the [System.Timers.Timer](#) component as follows: In the [OnStart](#) method, set parameters on the component, and then set the [Enabled](#) property to **true**. The timer raises events in your code periodically, at which time your service could do its monitoring. You can use the following code to do this:

**VB**

```
' Set up a timer to trigger every minute.
Dim timer As System.Timers.Timer = New System.Timers.Timer()
timer.Interval = 60000 ' 60 seconds
AddHandler timer.Elapsed, AddressOf Me.OnTimer
timer.Start()
```

Add code to handle the timer event:

**VB**

```

Private Sub OnTimer(sender As Object, e As Timers.ElapsedEventArgs)
    ' TODO: Insert monitoring activities here.
    eventLog1.WriteEntry("Monitoring the System", EventLogEntryType.Information,
eventId)
    eventId = eventId + 1
End Sub

```

You might want to perform tasks by using background worker threads instead of running all your work on the main thread. For an example of this, see the [System.ServiceProcess.ServiceBase](#) reference page.

## To define what occurs when the service is stopped

- Replace the code for the [OnStop](#) method with the following. This adds an entry to the event log when the service is stopped:

**VB**

```

Protected Overrides Sub OnStop()
    EventLog1.WriteEntry("In OnStop.")
End Sub

```

In the next section, you can override the [OnPause](#), [OnContinue](#), and [OnShutdown](#) methods to define additional processing for your component.

## To define other actions for the service

- Locate the method that you want to handle, and override it to define what you want to occur.

The following code shows how you can override the [OnContinue](#) method:

**VB**

```

Protected Overrides Sub OnContinue()
    EventLog1.WriteEntry("In OnContinue.")
End Sub

```

Some custom actions have to occur when a Windows service is installed by the [Installer](#) class. Visual Studio can create these installers specifically for a Windows service and add them to your project.

## Setting Service Status

Services report their status to the Service Control Manager, so that users can tell whether a service is functioning correctly. By default, services that inherit from [ServiceBase](#) report a limited set of status settings, including Stopped, Paused, and Running. If a service takes a little while to start up, it might be helpful to report a Start Pending status. You can also implement the Start Pending and Stop Pending status settings by adding code that calls into the Windows [SetServiceStatus](#) function.

## To implement service pending status

1. Add a **using** statement or **Imports** declaration to the [System.Runtime.InteropServices](#) namespace in the MyNewService.cs or MyNewService.vb file:

**VB**

```
Imports System.Runtime.InteropServices
```

2. Add the following code to MyNewService.cs to declare the [ServiceState](#) values and to add a structure for the status, which you'll use in a platform invoke call:

**VB**

```
Public Enum ServiceState
    SERVICE_STOPPED = 1
    SERVICE_START_PENDING = 2
    SERVICE_STOP_PENDING = 3
    SERVICE_RUNNING = 4
    SERVICE_CONTINUE_PENDING = 5
    SERVICE_PAUSE_PENDING = 6
    SERVICE_PAUSED = 7
End Enum

<StructLayout(LayoutKind.Sequential)>
Public Structure ServiceStatus
    Public dwServiceType As Long
    Public dwCurrentState As ServiceState
    Public dwControlsAccepted As Long
    Public dwWin32ExitCode As Long
    Public dwServiceSpecificExitCode As Long
    Public dwCheckpoint As Long
    Public dwWaitHint As Long
End Structure
```

3. Now, in the [MyNewService](#) class, declare the [SetServiceStatus](#) function by using platform invoke:

**VB**

```
Declare Auto Function SetServiceStatus Lib "advapi32.dll" (ByVal handle As IntPtr,
    ByRef serviceStatus As ServiceStatus) As Boolean
```

4. To implement the Start Pending status, add the following code to the beginning of the [OnStart](#) method:

**VB**

```
' Update the service state to Start Pending.
Dim serviceStatus As ServiceStatus = New ServiceStatus()
serviceStatus.dwCurrentState = ServiceState.SERVICE_START_PENDING
serviceStatus.dwWaitHint = 100000
SetServiceStatus(Me.ServiceHandle, serviceStatus)
```

5. Add code to set the status to Running at the end of the [OnStart](#) method.

```
// Update the service state to Running.
serviceStatus.dwCurrentState = ServiceState.SERVICE_RUNNING;
SetServiceStatus(this.ServiceHandle, ref serviceStatus);
```

**VB**

```
' Update the service state to Running.
serviceStatus.dwCurrentState = ServiceState.SERVICE_RUNNING
SetServiceStatus(Me.ServiceHandle, serviceStatus)
```

6. (Optional) Repeat this procedure for the [OnStop](#) method.

### Caution

The [Service Control Manager](#) uses the **dwWaitHint** and **dwCheckpoint** members of the [SERVICE\\_STATUS structure](#) to determine how much time to wait for a Windows Service to start or shut down. If your [OnStart](#) and [OnStop](#) methods run long, your service can request more time by calling [SetServiceStatus](#) again with an incremented **dwCheckPoint** value.

## Adding Installers to the Service

Before you can run a Windows Service, you need to install it, which registers it with the Service Control Manager. You can add installers to your project that handle the registration details.

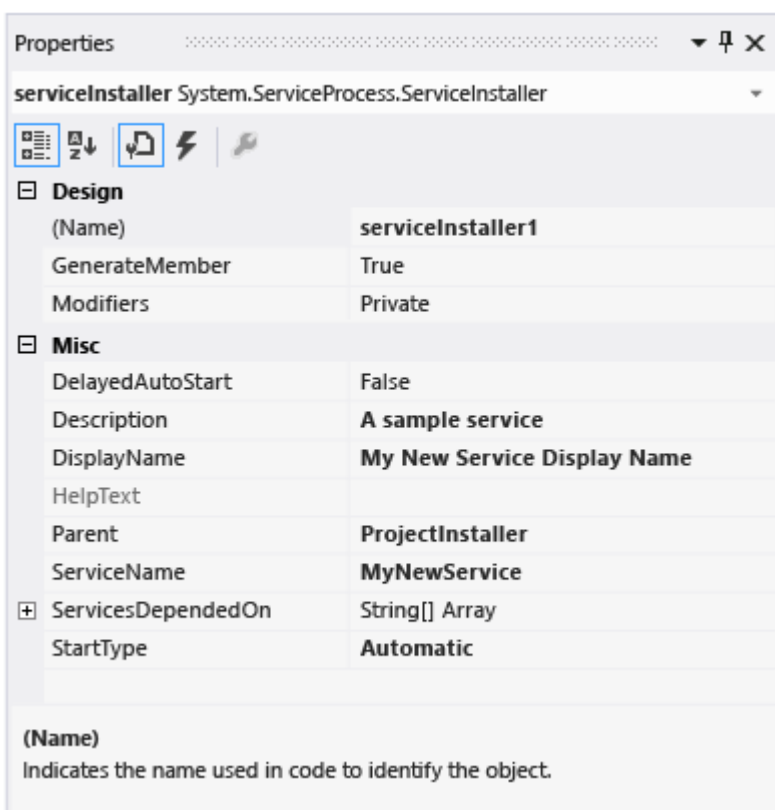
### To create the installers for your service

1. In **Solution Explorer**, open the context menu for **MyNewService.cs** or **MyNewService.vb**, and then choose **View Designer**.
2. Click the background of the designer to select the service itself, instead of any of its contents.
3. Open the context menu for the designer window (if you're using a pointing device, right-click inside the window), and then choose **Add Installer**.

By default, a component class that contains two installers is added to your project. The component is named

**ProjectInstaller**, and the installers it contains are the installer for your service and the installer for the service's associated process.

- In **Design** view for **ProjectInstaller**, choose **serviceInstaller1** for a Visual C# project, or **ServiceInstaller1** for a Visual Basic project.
- In the **Properties** window, make sure the [ServiceName](#) property is set to **MyNewService**.
- Set the **Description** property to some text, such as "A sample service". This text appears in the Services window and helps the user identify the service and understand what it's used for.
- Set the [DisplayName](#) property to the text that you want to appear in the Services window in the **Name** column. For example, you can enter "MyNewService Display Name". This name can be different from the [ServiceName](#) property, which is the name used by the system (for example, when you use the `net start` command to start your service).
- Set the [StartType](#) property to [Automatic](#).



- In the designer, choose **serviceProcessInstaller1** for a Visual C# project, or **ServiceProcessInstaller1** for a Visual Basic project. Set the [Account](#) property to [LocalSystem](#). This will cause the service to be installed and to run on a local service account.



### Security Note

The [LocalSystem](#) account has broad permissions, including the ability to write to the event log. Use this account with caution, because it might increase your risk of attacks from malicious software. For other tasks, consider using the [LocalService](#) account, which acts as a non-privileged user on the local computer and presents anonymous credentials to any remote server. This example fails if you try to use the [LocalService](#) account, because it needs permission to write to the event log.

For more information about installers, see [How to: Add Installers to Your Service Application](#).

## Set Startup Parameters

A Windows Service, like any other executable, can accept command-line arguments, or startup parameters. When you add code to process startup parameters, users can start your service with their own custom startup parameters by using the Services window in the Windows Control Panel. However, these startup parameters are not persisted the next time the service starts. To set startup parameters permanently, you can set them in the registry, as shown in this procedure.

### Note

Before you decide to add startup parameters, consider whether that is the best way to pass information to your service. Although startup parameters are easy to use and to parse, and users can easily override them, they might be harder for users to discover and use without documentation. Generally, if your service requires more than just a few startup parameters, you should consider using the registry or a configuration file instead. Every Windows Service has an entry in the registry under HKLM\System\CurrentControlSet\services. Under the service's key, you can use the **Parameters** subkey to store information that your service can access. You can use application configuration files for a Windows Service the same way you do for other types of programs. For example code, see [AppSettings](#).

## Adding startup parameters

1. In the **Main** method in Program.cs or in MyNewService.Designer.vb, add an argument for the command line:

VB

```
Shared Sub Main(ByVal cmdArgs() As String)           Dim ServicesToRun() As
System.ServiceProcess.ServiceBase                 ServicesToRun = New
System.ServiceProcess.ServiceBase() {New MyNewServiceVB(cmdArgs)}
System.ServiceProcess.ServiceBase.Run(ServicesToRun) End Sub
```

2. Change the **MyNewService** constructor as follows:

VB

```
Public Sub New(ByVal cmdArgs() As String)           InitializeComponent()           Dim
eventSourceName As String = "MySource"           Dim logName As String =
"MyNewLog"           If (cmdArgs.Count() > 0) Then           eventSourceName =
cmdArgs(0)           End If           If (cmdArgs.Count() > 1) Then           logName =
cmdArgs(1)           End If           eventLog1 = New
System.Diagnostics.EventLog()           If (Not
System.Diagnostics.EventLog.SourceExists(eventSourceName)) Then
System.Diagnostics.EventLog.CreateEventSource(           eventSourceName,
logName)           End If           eventLog1.Source = eventSourceName
eventLog1.Log = logName           End Sub
```

This code sets the event source and log name according to the supplied startup parameters, or uses default values

if no arguments are supplied.

- To specify the command-line arguments, add the following code to the `ProjectInstaller` class in `ProjectInstaller.cs` or `ProjectInstaller.vb`:

**VB**

```
Protected Overrides Sub OnBeforeInstall(ByVal savedState As IDictionary)
    Dim parameter As String = "MySource1" & "MyLogFile1"
    Context.Parameters("assemblypath") = "" + Context.Parameters("assemblypath") +
    "" & "" + parameter + ""
    MyBase.OnBeforeInstall(savedState)
End Sub
```

This code modifies the **ImagePath** registry key, which typically contains the full path to the executable for the Windows Service, by adding the default parameter values. The quotation marks around the path (and around each individual parameter) are required for the service to start up correctly. To change the startup parameters for this Windows Service, users can change the parameters given in the **ImagePath** registry key, although the better way is to change it programmatically and expose the functionality to users in a friendly way (for example, in a management or configuration utility).

## Building the Service

### To build your service project

- In **Solution Explorer**, open the context menu for your project, and then choose **Properties**. The property pages for your project appear.
- On the Application tab, in the **Startup object** list, choose **MyNewService.Program**.
- In **Solution Explorer**, open the context menu for your project, and then choose **Build** to build the project (Keyboard: Ctrl+Shift+B).

## Installing the Service

Now that you've built the Windows service, you can install it. To install a Windows service, you must have administrative credentials on the computer on which you're installing it.

### To install a Windows Service

- In Windows 7 and Windows Server, open the **Developer Command Prompt** under **Visual Studio Tools** in the **Start** menu. In Windows 8 or Windows 8.1, choose the **Visual Studio Tools** tile on the **Start** screen, and then run Developer Command Prompt with administrative credentials. (If you're using a mouse, right-click on **Developer Command Prompt**, and then choose **Run as Administrator**.)
- In the Command Prompt window, navigate to the folder that contains your project's output. For example, under your My Documents folder, navigate to Visual Studio 2013\Projects\MyNewService\bin\Debug.

- Enter the following command:

```
installutil.exe MyNewService.exe
```

If the service installs successfully, `installutil.exe` will report success. If the system could not find `InstallUtil.exe`, make sure that it exists on your computer. This tool is installed with the .NET Framework to the folder `%WINDIR%\Microsoft.NET\Framework[64]\framework_version`. For example, the default path for the 32-bit version of the .NET Framework 4, 4.5, 4.5.1, and 4.5.2 is `C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe`.

If the `installutil.exe` process reports failure, check the install log to find out why. By default the log is in the same folder as the service executable. The installation can fail if the `RunInstallerAttribute` class is not present on the `ProjectInstaller` class, or else the attribute is not set to `true`, or else the `ProjectInstaller` class is not `public`.

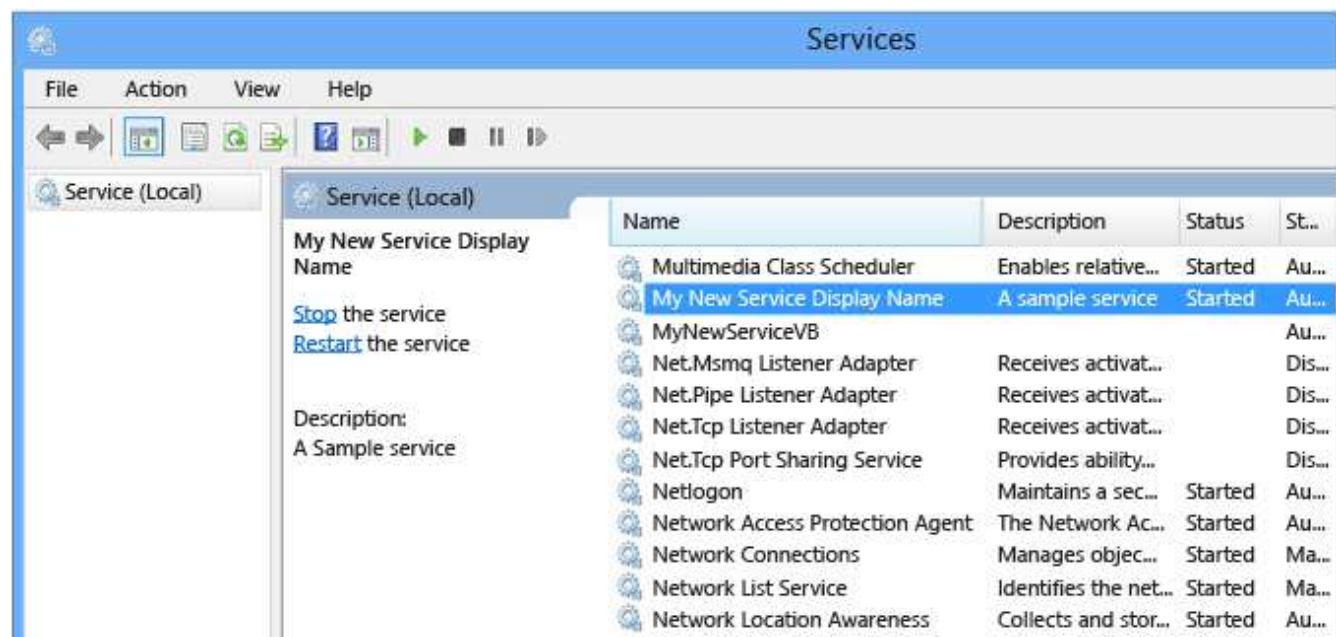
For more information, see [How to: Install and Uninstall Services](#).

## Starting and Running the Service

### To start and stop your service

- In Windows, open the **Start** screen or **Start** menu, and type `services.msc`.

You should now see **MyNewService** listed in the **Services** window.



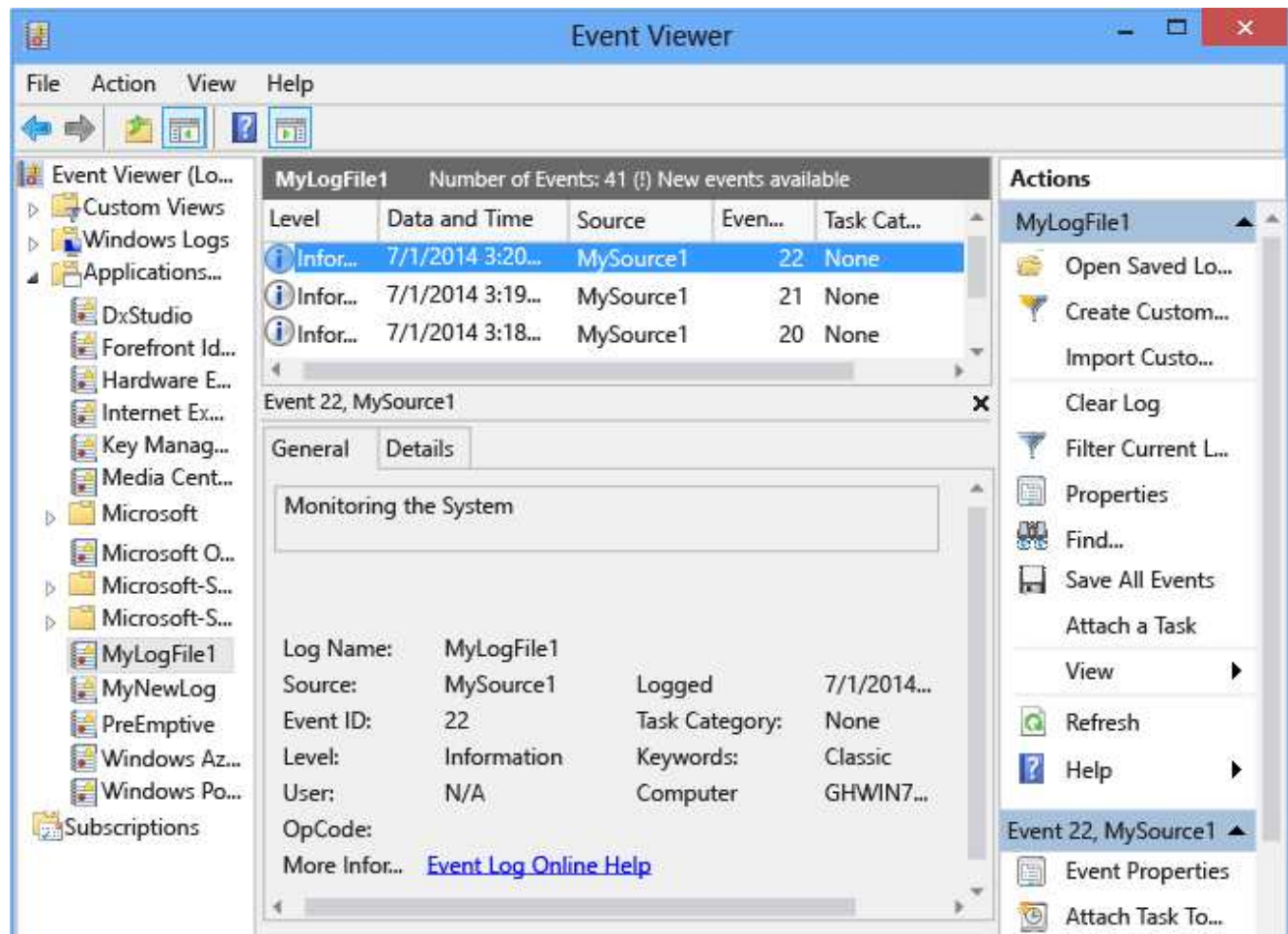
- In the **Services** window, open the shortcut menu for your service, and then choose **Start**.
- Open the shortcut menu for the service, and then choose **Stop**.
- (Optional) From the command line, you can use the commands `net start ServiceName` and `net`



stop **ServiceName** to start and stop your service.

## To verify the event log output of your service

1. In Visual Studio, open **Server Explorer** (Keyboard: Ctrl+Alt+S), and access the **Event Logs** node for the local computer.
2. Locate the listing for **MyNewLog** (or **MyLogFile1**, if you used the optional procedure to add command-line arguments) and expand it. You should see entries for the two actions (start and stop) your service has performed.



## Uninstalling a Windows Service

### To uninstall your service

1. Open a developer command prompt with administrative credentials.
2. In the Command Prompt window, navigate to the folder that contains your project's output. For example, under your My Documents folder, navigate to Visual Studio 2013\Projects\MyNewService\bin\Debug.
3. Enter the following command:

```
installutil.exe /u MyNewService.exe
```

If the service uninstalls successfully, `installutil.exe` will report that your service was successfully removed. For more information, see [How to: Install and Uninstall Services](#).

## Next Steps

You can create a standalone setup program that others can use to install your Windows service, but it requires additional steps. ClickOnce doesn't support Windows services, so you can't use the Publish Wizard. You can use a full edition of InstallShield, which Microsoft doesn't provide. For more information about InstallShield, see [InstallShield Limited Edition](#). You can also use the [Windows Installer XML Toolset](#) to create an installer for a Windows service.

You might explore the use of a [ServiceController](#) component, which enables you to send commands to the service you have installed.

You can use an installer to create an event log when the application is installed instead of creating the event log when the application runs. Additionally, the event log will be deleted by the installer when the application is uninstalled. For more information, see the [EventLogInstaller](#) reference page.

## See Also

- [Developing Windows Service Applications](#)
- [Introduction to Windows Service Applications](#)
- [How to: Debug Windows Service Applications](#)
- [How to: Access and Initialize Server Explorer/Database Explorer Services \(Windows\)](#)

# Service Application Programming Architecture

## .NET Framework (current version)

Windows Service applications are based on a class that inherits from the [System.ServiceProcess.ServiceBase](#) class. You override methods from this class and define functionality for them to determine how your service behaves.

The main classes involved in service creation are:

- [System.ServiceProcess.ServiceBase](#) — You override methods from the [ServiceBase](#) class when creating a service and define the code to determine how your service functions in this inherited class.
- [System.ServiceProcess.ServiceProcessInstaller](#) and [System.ServiceProcess.ServiceInstaller](#) —You use these classes to install and uninstall your service.

In addition, a class named [ServiceController](#) can be used to manipulate the service itself. This class is not involved in the creation of a service, but can be used to start and stop the service, pass commands to it, and return a series of enumerations.

## Defining Your Service's Behavior

In your service class, you override base class functions that determine what happens when the state of your service is changed in the Services Control Manager. The [ServiceBase](#) class exposes the following methods, which you can override to add custom behavior.

Method	Override to
<a href="#">OnStart</a>	Indicate what actions should be taken when your service starts running. You must write code in this procedure for your service to perform useful work.
<a href="#">OnPause</a>	Indicate what should happen when your service is paused.
<a href="#">OnStop</a>	Indicate what should happen when your service stops running.
<a href="#">OnContinue</a>	Indicate what should happen when your service resumes normal functioning after being paused.
<a href="#">OnShutdown</a>	Indicate what should happen just prior to your system shutting down, if your service is running at that time.
<a href="#">OnCustomCommand</a>	Indicate what should happen when your service receives a custom command. For more information on custom commands, see MSDN online.

**OnPowerEvent**

Indicate how the service should respond when a power management event is received, such as a low battery or suspended operation.

 **Note**

These methods represent states that the service moves through in its lifetime; the service transitions from one state to the next. For example, you will never get the service to respond to an [OnContinue](#) command before [OnStart](#) has been called.

There are several other properties and methods that are of interest. These include:

- The [Run](#) method on the [ServiceBase](#) class. This is the main entry point for the service. When you create a service using the Windows Service template, code is inserted in your application's [Main](#) method to run the service. This code looks like this:

**VB**

```
Dim ServicesToRun() As System.ServiceProcess.ServiceBase
ServicesToRun =
    New System.ServiceProcess.ServiceBase() {New Service1()}
System.ServiceProcess.ServiceBase.Run(ServicesToRun)
```

 **Note**

These examples use an array of type [ServiceBase](#), into which each service your application contains can be added, and then all of the services can be run together. If you are only creating a single service, however, you might choose not to use the array and simply declare a new object inheriting from [ServiceBase](#) and then run it. For an example, see [How to: Write Services Programmatically](#).

- A series of properties on the [ServiceBase](#) class. These determine what methods can be called on your service. For example, when the [CanStop](#) property is set to **true**, the [OnStop](#) method on your service can be called. When the [CanPauseAndContinue](#) property is set to **true**, the [OnPause](#) and [OnContinue](#) methods can be called. When you set one of these properties to **true**, you should then override and define processing for the associated methods.

 **Note**

Your service must override at least [OnStart](#) and [OnStop](#) to be useful.

You can also use a component called the [ServiceController](#) to communicate with and control the behavior of an existing service.

## See Also

[Introduction to Windows Service Applications](#)  
[How to: Create Windows Services](#)

© 2016 Microsoft

# How to: Create Windows Services

## .NET Framework (current version)

When you create a service, you can use a Visual Studio project template called **Windows Service**. This template automatically does much of the work for you by referencing the appropriate classes and namespaces, setting up the inheritance from the base class for services, and overriding several of the methods you're likely to want to override.

### Warning

The Windows Services project template is not available in the Express edition of Visual Studio.

At a minimum, to create a functional service you must:

- Set the [ServiceName](#) property.
- Create the necessary installers for your service application.
- Override and specify code for the [OnStart](#) and [OnStop](#) methods to customize the ways in which your service behaves.

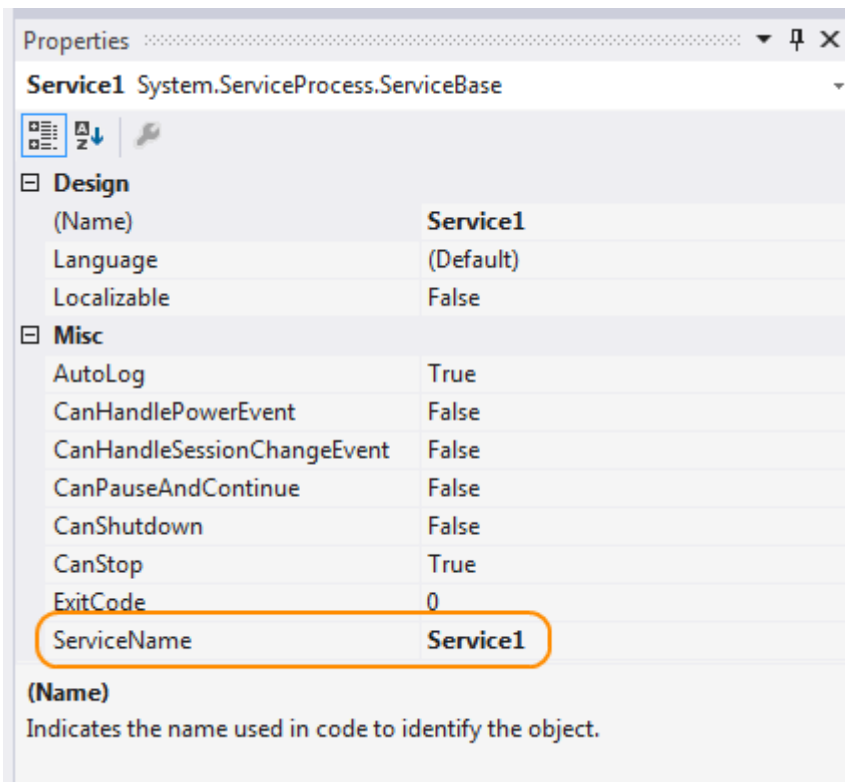
## To create a Windows Service application

1. Create a **Windows Service** project.

### Note

For instructions on writing a service without using the template, see [How to: Write Services Programmatically](#).

2. In the **Properties** window, set the [ServiceName](#) property for your service.



#### Note

The value of the [ServiceName](#) property must always match the name recorded in the installer classes. If you change this property, you must update the [ServiceName](#) property of installer classes as well.

- Set any of the following properties to determine how your service will function.

Property	Setting
<a href="#">CanStop</a>	<b>True</b> to indicate that the service will accept requests to stop running; <b>false</b> to prevent the service from being stopped.
<a href="#">CanShutdown</a>	<b>True</b> to indicate that the service wants to receive notification when the computer on which it lives shuts down, enabling it to call the <a href="#">OnShutdown</a> procedure.
<a href="#">CanPauseAndContinue</a>	<b>True</b> to indicate that the service will accept requests to pause or to resume running; <b>false</b> to prevent the service from being paused and resumed.
<a href="#">CanHandlePowerEvent</a>	<b>True</b> to indicate that the service can handle notification of changes to the computer's power status; <b>false</b> to prevent the service from being notified of these changes.
<a href="#">AutoLog</a>	<b>True</b> to write informational entries to the Application event log when your service performs an action; <b>false</b> to disable this functionality. For more information, see <a href="#">How to: Log Information About Services</a> .

 **Note**

By default, [AutoLog](#) is set to **true**.

 **Note**

When [CanStop](#) or [CanPauseAndContinue](#) are set to **false**, the **Service Control Manager** will disable the corresponding menu options to stop, pause, or continue the service.

4. Access the Code Editor and fill in the processing you want for the [OnStart](#) and [OnStop](#) procedures.
5. Override any other methods for which you want to define functionality.
6. Add the necessary installers for your service application. For more information, see [How to: Add Installers to Your Service Application](#).
7. Build your project by selecting **Build Solution** from the **Build** menu.

 **Note**

Do not press F5 to run your project — you cannot run a service project in this way.

8. Install the service. For more information, see [How to: Install and Uninstall Services](#).

## See Also

- [Introduction to Windows Service Applications](#)
- [How to: Write Services Programmatically](#)
- [How to: Add Installers to Your Service Application](#)
- [How to: Log Information About Services](#)
- [How to: Start Services](#)
- [How to: Specify the Security Context for Services](#)
- [How to: Install and Uninstall Services](#)
- [Walkthrough: Creating a Windows Service Application in the Component Designer](#)



# How to: Write Services Programmatically

## .NET Framework (current version)

If you choose not to use the Windows Service project template, you can write your own services by setting up the inheritance and other infrastructure elements yourself. When you create a service programmatically, you must perform several steps that the template would otherwise handle for you:

- You must set up your service class to inherit from the [ServiceBase](#) class.
- You must create a **Main** method for your service project that defines the services to run and calls the [Run](#) method on them.
- You must override the [OnStart](#) and [OnStop](#) procedures and fill in any code you want them to run.

## To write a service programmatically

1. Create an empty project and create a reference to the necessary namespaces by following these steps:
  - a. In **Solution Explorer**, right-click the **References** node and click **Add Reference**.
  - b. On the **.NET Framework** tab, scroll to **System.dll** and click **Select**.
  - c. Scroll to **System.ServiceProcess.dll** and click **Select**.
  - d. Click **OK**.
2. Add a class and configure it to inherit from [ServiceBase](#):

**VB**

```
Public Class UserService1
    Inherits System.ServiceProcess.ServiceBase
End Class
```

3. Add the following code to configure your service class:

**VB**

```
Public Sub New()
    Me.ServiceName = "MyService2"
    Me.CanStop = True
    Me.CanPauseAndContinue = True
    Me.AutoLog = True
End Sub
```

4. Create a **Main** method for your class, and use it to define the service your class will contain; `userService1` is the

name of the class:

**VB**

```
Shared Sub Main()  
    System.ServiceProcess.ServiceBase.Run(New UserService1)  
End Sub
```

5. Override the [OnStart](#) method, and define any processing you want to occur when your service is started.

**VB**

```
Protected Overrides Sub OnStart(ByVal args() As String)  
    ' Insert code here to define processing.  
End Sub
```

6. Override any other methods you want to define custom processing for, and write code to determine the actions the service should take in each case.
7. Add the necessary installers for your service application. For more information, see [How to: Add Installers to Your Service Application](#).
8. Build your project by selecting **Build Solution** from the **Build** menu.

 **Note**

Do not press F5 to run your project — you cannot run a service project in this way.

9. Create a setup project and the custom actions to install your service. For an example, see [Walkthrough: Creating a Windows Service Application in the Component Designer](#).
10. Install the service. For more information, see [How to: Install and Uninstall Services](#).

## See Also

- [Introduction to Windows Service Applications](#)
- [How to: Create Windows Services](#)
- [How to: Add Installers to Your Service Application](#)
- [How to: Log Information About Services](#)
- [Walkthrough: Creating a Windows Service Application in the Component Designer](#)

# How to: Add Installers to Your Service Application

## .NET Framework (current version)

Visual Studio ships installation components that can install resources associated with your service applications. Installation components register an individual service on the system to which it is being installed and let the Services Control Manager know that the service exists. When you work with a service application, you can select a link in the Properties window to automatically add the appropriate installers to your project.

### Note

Property values for your service are copied from the service class to the installer class. If you update the property values on the service class, they are not automatically updated in the installer.

When you add an installer to your project, a new class (which, by default, is named `ProjectInstaller`) is created in the project, and instances of the appropriate installation components are created within it. This class acts as a central point for all of the installation components your project needs. For example, if you add a second service to your application and click the Add Installer link, a second installer class is not created; instead, the necessary additional installation component for the second service is added to the existing class.

You do not need to do any special coding within the installers to make your services install correctly. However, you may occasionally need to modify the contents of the installers if you need to add special functionality to the installation process.

### Note

The dialog boxes and menu commands you see might differ from those described in Help depending on your active settings or edition. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see [Customizing Development Settings in Visual Studio](#).

## To add installers to your service application

1. In **Solution Explorer**, access **Design** view for the service for which you want to add an installation component.
2. Click the background of the designer to select the service itself, rather than any of its contents.
3. With the designer in focus, right-click, and then click **Add Installer**.

A new class, `ProjectInstaller`, and two installation components, `ServiceProcessInstaller` and `ServiceInstaller`, are added to your project, and property values for the service are copied to the components.

4. Click the `ServiceInstaller` component and verify that the value of the `ServiceName` property is set to the same value as

the [ServiceName](#) property on the service itself.

- To determine how your service will be started, click the [ServiceInstaller](#) component and set the [StartType](#) property to the appropriate value.

Value	Result
<a href="#">Manual</a>	The service must be manually started after installation. For more information, see <a href="#">How to: Start Services</a> .
<a href="#">Automatic</a>	The service will start by itself whenever the computer reboots.
<a href="#">Disabled</a>	The service cannot be started.

- To determine the security context in which your service will run, click the [ServiceProcessInstaller](#) component and set the appropriate property values. For more information, see [How to: Specify the Security Context for Services](#).
- Override any methods for which you need to perform custom processing.
- Perform steps 1 through 7 for each additional service in your project.

#### Note

For each additional service in your project, you must add an additional [ServiceInstaller](#) component to the project's **ProjectInstaller** class. The [ServiceProcessInstaller](#) component added in step three works with all of the individual service installers in the project.

## See Also

- [Introduction to Windows Service Applications](#)
- [How to: Install and Uninstall Services](#)
- [How to: Start Services](#)
- [How to: Specify the Security Context for Services](#)

# How to: Specify the Security Context for Services

## .NET Framework (current version)

By default, services run in a different security context than that of the logged-in user. Services run in the context of the default system account, called **LocalSystem**, which gives them different access privileges to system resources than the user. You can change this behavior to specify a different user account under which your service should run.

You set the security context by manipulating the **Account** property for the process within which the service runs. This property allows you to set the service to one of four account types:

- **User**, which causes the system to prompt for a valid user name and password when the service is installed and runs in the context of an account specified by a single user on the network;
- **LocalService**, which runs in the context of an account that acts as a non-privileged user on the local computer, and presents anonymous credentials to any remote server;
- **LocalSystem**, which runs in the context of an account that provides extensive local privileges, and presents the computer's credentials to any remote server;
- **NetworkService**, which runs in the context of an account that acts as a non-privileged user on the local computer, and presents the computer's credentials to any remote server.

For more information, see the [ServiceAccount](#) enumeration.

## To specify the security context for a service

1. After creating your service, add the necessary installers for it. For more information, see [How to: Add Installers to Your Service Application](#).
2. In the designer, access the **ProjectInstaller** class and click the service process installer for the service you are working with.

### Note

For every service application, there are at least two installation components in the **ProjectInstaller** class — one that installs the processes for all services in the project, and one installer for each service the application contains. In this instance, you want to select **ServiceProcessInstaller**.

3. In the **Properties** window, set the **Account** to the appropriate value.

## See Also

- [Introduction to Windows Service Applications](#)
- [How to: Add Installers to Your Service Application](#)
- [How to: Create Windows Services](#)

© 2016 Microsoft

# How to: Install and Uninstall Services

## .NET Framework (current version)

If you're developing a Windows Service by using the .NET Framework, you can quickly install your service application by using a command-line utility called InstallUtil.exe. If you're a developer who wants to release a Windows Service that users can install and uninstall you should use InstallShield. See [121be21b-b916-43e2-8f10-8b080516d2a0](#).

### Warning

If you want to uninstall a service from your computer, don't follow the steps in this article. Instead, find out which program or software package installed the service, and then choose **Add/Remove Programs** in Control Panel to uninstall that program. Note that many services are integral parts of Windows; if you remove them, you might cause system instability.

In order to use the steps in this article, you first need to add a service installer to your Windows Service. See [Walkthrough: Creating a Windows Service Application in the Component Designer](#).

Windows Service projects cannot be run directly from the Visual Studio development environment by pressing F5. This is because the service in the project must be installed before you can run the project.

### Tip

You can launch **Server Explorer** and verify that your service has been installed or uninstalled. For more information, see [How to: Access and Initialize Server Explorer/Database Explorer](#).

## To install your service manually

1. On the Windows **Start** menu or **Start** screen, choose **Visual Studio** , **Visual Studio Tools**, **Developer Command Prompt**.

A Visual Studio command prompt appears.

2. Access the directory where your project's compiled executable file is located.
3. Run InstallUtil.exe from the command prompt with your project's executable as a parameter:

```
installutil <yourproject>.exe
```

If you're using the Visual Studio command prompt, InstallUtil.exe should be on the system path. If not, you can add it to the path, or use the fully qualified path to invoke it. This tool is installed with the .NET Framework, and its path is

`%WINDIR%\Microsoft.NET\Framework[64]\<framework_version>`. For example, for the 32-bit version of the .NET Framework 4 or 4.5.\*, if your Windows installation directory is `C:\Windows`, the path is `C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe`. For the 64-bit version of the .NET Framework 4 or 4.5.\*, the default path is `C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe`.

## To uninstall your service manually

1. On the Windows **Start** menu or **Start** screen, choose **Visual Studio, Visual Studio Tools, Developer Command Prompt**.

A Visual Studio command prompt appears.

2. Run `InstallUtil.exe` from the command prompt with your project's output as a parameter:

```
installutil /u <yourproject>.exe
```

3. Sometimes, after the executable for a service is deleted, the service might still be present in the registry. In that case, use the command `sc delete` to remove the entry for the service from the registry.

## See Also

[Introduction to Windows Service Applications](#)

[How to: Create Windows Services](#)

[How to: Add Installers to Your Service Application](#)

[Installutil.exe \(Installer Tool\)](#)



# How to: Start Services

## .NET Framework (current version)

After a service is installed, it must be started. Starting calls the [OnStart](#) method on the service class. Usually, the [OnStart](#) method defines the useful work the service will perform. After a service starts, it remains active until it is manually paused or stopped.

Services can be set up to start automatically or manually. A service that starts automatically will be started when the computer on which it is installed is rebooted or first turned on. A user must start a service that starts manually.

### Note

By default, services created with Visual Studio are set to start manually.

There are several ways you can manually start a service — from **Server Explorer**, from the **Services Control Manager**, or from code using a component called the [ServiceController](#).

You set the [StartType](#) property on the [ServiceInstaller](#) class to determine whether a service should be started manually or automatically.

## To specify how a service should start

1. After creating your service, add the necessary installers for it. For more information, see [How to: Add Installers to Your Service Application](#).
2. In the designer, click the service installer for the service you are working with.
3. In the **Properties** window, set the [StartType](#) property to one of the following:

To have your service install	Set this value
When the computer is restarted	<b>Automatic</b>
When an explicit user action starts the service	<b>Manual</b>

### Tip

To prevent your service from being started at all, you can set the [StartType](#) property to **Disabled**. You might do this if you are going to reboot a server several times and want to save time by preventing the services that would normally start from starting up.

 **Note**

These and other properties can be changed after your service is installed.

There are several ways you can start a service that has its [StartType](#) process set to **Manual** — from **Server Explorer**, from the **Windows Services Control Manager**, or from code. It is important to note that not all of these methods actually start the service in the context of the **Services Control Manager**; **Server Explorer** and programmatic methods of starting the service actually manipulate the controller.

## To manually start a service from Server Explorer

1. In **Server Explorer**, add the server you want if it is not already listed. For more information, see [How to: Access and Initialize Server Explorer/Database Explorer](#).
2. Expand the **Services** node, and then locate the service you want to start.
3. Right-click the name of the service, and click **Start**.

## To manually start a service from Services Control Manager

1. Open the **Services Control Manager** by doing one of the following:
  - In Windows XP and 2000 Professional, right-click **My Computer** on the desktop, and then click **Manage**. In the dialog box that appears, expand the **Services and Applications** node.
  - or -
  - In Windows Server 2003 and Windows 2000 Server, click **Start**, point to **Programs**, click **Administrative Tools**, and then click **Services**.

 **Note**

In Windows NT version 4.0, you can open this dialog box from **Control Panel**.

You should now see your service listed in the **Services** section of the window.

2. Select your service in the list, right-click it, and then click **Start**.

## To manually start a service from code

1. Create an instance of the [ServiceController](#) class, and configure it to interact with the service you want to administer.
2. Call the [Start](#) method to start the service.

## See Also

[Introduction to Windows Service Applications](#)

[How to: Create Windows Services](#)

[How to: Add Installers to Your Service Application](#)

[How to: Access and Initialize Server Explorer/Database Explorer](#)

© 2016 Microsoft

# How to: Pause a Windows Service (Visual Basic)

## .NET Framework (current version)

This example uses the [ServiceController](#) component to pause the IIS Admin service on the local computer.

## Example

**VB**

```
Dim theController As System.ServiceProcess.ServiceController
theController = New System.ServiceProcess.ServiceController("IISAdmin")
```

**VB**

```
' Pauses the service.
theController.Pause()
```

This code example is also available as an IntelliSense code snippet. In the code snippet picker, it is located in **Windows Operating System > Windows Services**. For more information, see [Code Snippets](#).

## Compiling the Code

This example requires:

- A project reference to System.serviceprocess.dll.
- Access to the members of the [System.ServiceProcess](#) namespace. Add an **Imports** statement if you are not fully qualifying member names in your code. For more information, see [Imports Statement \(.NET Namespace and Type\)](#).

## Robust Programming

The [MachineName](#) property of the [ServiceController](#) class is the local computer by default. To reference Windows services on another computer, change the [MachineName](#) property to the name of that computer.

The following conditions may cause an exception:

- The service cannot be paused. ([InvalidOperationException Class](#))
- An error occurred when accessing a system API. ([Win32Exception Class](#))

## .NET Framework Security

Control of services on the computer may be restricted by using the [ServiceControllerPermissionAccess Enumeration](#) to set permissions in the [ServiceControllerPermission Class](#).

Access to service information may be restricted by using the [PermissionState Enumeration](#) to set permissions in the [SecurityPermission Class](#).

## See Also

[ServiceController](#)

[ServiceControllerStatus](#)

[WaitForStatus](#)

[How to: Continue a Windows Service \(Visual Basic\)](#)

© 2016 Microsoft

# How to: Continue a Windows Service (Visual Basic)

## .NET Framework (current version)

This example uses the [ServiceController](#) component to continue the IIS Admin service on the local computer.

## Example

**VB**

```
Dim theController As System.ServiceProcess.ServiceController
theController = New System.ServiceProcess.ServiceController("IISAdmin")
```

**VB**

```
' Checks that the service is paused.
If theController.Status =
    System.ServiceProcess.ServiceControllerStatus.Paused Then

    ' Continues the service.
    theController.Continue()
End If
```

This code example is also available as an IntelliSense code snippet. In the code snippet picker, it is located in **Windows Operating System > Windows Services**. For more information, see [Code Snippets](#).

## Compiling the Code

This example requires:

- A project reference to System.serviceprocess.dll.
- Access to the members of the [System.ServiceProcess](#) namespace. Add an **Imports** statement if you are not fully qualifying member names in your code. For more information, see [Imports Statement \(.NET Namespace and Type\)](#).

## Robust Programming

The [MachineName](#) property of the [ServiceController](#) class is the local computer by default. To reference Windows services on another computer, change the [MachineName](#) property to the name of that computer.

You cannot call the [Continue](#) method on a service until the service controller status is [Paused](#).

The following conditions may cause an exception:

- The service cannot be resumed. ([InvalidOperationException](#))
- An error occurred when accessing a system API. ([Win32Exception](#))

## .NET Framework Security

Control of services on the computer may be restricted by using the [ServiceControllerPermissionAccess](#) enumeration to set permissions in the [ServiceControllerPermission](#) class.

Access to service information may be restricted by using the [PermissionState](#) enumeration to set permissions in the [SecurityPermission](#) class.

## See Also

[ServiceController](#)

[ServiceControllerStatus](#)

[How to: Pause a Windows Service \(Visual Basic\)](#)

© 2016 Microsoft

# How to: Debug Windows Service Applications

## .NET Framework (current version)

A service must be run from within the context of the Services Control Manager rather than from within Visual Studio. For this reason, debugging a service is not as straightforward as debugging other Visual Studio application types. To debug a service, you must start the service and then attach a debugger to the process in which it is running. You can then debug your application by using all of the standard debugging functionality of Visual Studio.

### Caution

You should not attach to a process unless you know what the process is and understand the consequences of attaching to and possibly killing that process. For example, if you attach to the WinLogon process and then stop debugging, the system will halt because it can't operate without WinLogon.

You can attach the debugger only to a running service. The attachment process interrupts the current functioning of your service; it doesn't actually stop or pause the service's processing. That is, if your service is running when you begin debugging, it is still technically in the Started state as you debug it, but its processing has been suspended.

After attaching to the process, you can set breakpoints and use these to debug your code. Once you exit the dialog box you use to attach to the process, you are effectively in debug mode. You can use the Services Control Manager to start, stop, pause and continue your service, thus hitting the breakpoints you've set. You can later remove this dummy service after debugging is successful.

This article covers debugging a service that's running on the local computer, but you can also debug Windows Services that are running on a remote computer. See [Remote Debugging](#).

### Note

Debugging the [OnStart](#) method can be difficult because the Services Control Manager imposes a 30-second limit on all attempts to start a service. For more information, see [Troubleshooting: Debugging Windows Services](#).

### Warning

To get meaningful information for debugging, the Visual Studio debugger needs to find symbol files for the binaries that are being debugged. If you are debugging a service that you built in Visual Studio, the symbol files (.pdb files) are in the same folder as the executable or library, and the debugger loads them automatically. If you are debugging a service that you didn't build, you should first find symbols for the service and make sure they can be found by the debugger. See [Specify Symbol \(.pdb\) and Source Files in the Visual Studio Debugger](#). If you're debugging a system process or want to have symbols for system calls in your services, you should add the Microsoft Symbol Servers. See [Debugging Symbols](#).



## To debug a service

1. Build your service in the Debug configuration.
2. Install your service. For more information, see [How to: Install and Uninstall Services](#).
3. Start your service, either from **Services Control Manager**, **Server Explorer**, or from code. For more information, see [How to: Start Services](#).
4. Start Visual Studio with administrative credentials so you can attach to system processes.
5. (Optional) On the Visual Studio menu bar, choose **Tools, Options**. In the **Options** dialog box, choose **Debugging, Symbols**, select the **Microsoft Symbol Servers** check box, and then choose the **OK** button.
6. On the menu bar, choose **Attach to Process** from the **Debug** or **Tools** menu. (Keyboard: Ctrl+Alt+P)

The **Processes** dialog box appears.

7. Select the **Show processes from all users** check box.
8. In the **Available Processes** section, choose the process for your service, and then choose **Attach**.

### Tip

The process will have the same name as the executable file for your service.

The **Attach to Process** dialog box appears.

9. Choose the appropriate options, and then choose **OK** to close the dialog box.

### Note

You are now in debug mode.

10. Set any breakpoints you want to use in your code.
11. Access the Services Control Manager and manipulate your service, sending stop, pause, and continue commands to hit your breakpoints. For more information about running the Services Control Manager, see [How to: Start Services](#). Also, see [Troubleshooting: Debugging Windows Services](#).

## Debugging Tips for Windows Services

Attaching to the service's process allows you to debug most, but not all, the code for that service. For example, because the service has already been started, you cannot debug the code in the service's **OnStart** method or the code in the **Main** method that is used to load the service this way. One way to work around this limitation is to create a temporary second service in your service application that exists only to aid in debugging. You can install both services, and then start this

dummy service to load the service process. Once the temporary service has started the process, you can use the **Debug** menu in Visual Studio to attach to the service process.

Try adding calls to the [Sleep](#) method to delay action until you're able to attach to the process.

Try changing the program to a regular console application. To do this, rewrite the `Main` method as follows so it can run both as a Windows Service and as a console application, depending on how it's started.

## How to: Run a Windows Service as a console application

1. Add a method to your service that runs the [OnStart](#) and [OnStop](#) methods:

```
internal void TestStartupAndStop(string[] args)
{
    this.OnStart(args);
    Console.ReadLine();
    this.OnStop();
}
```

2. Rewrite the `Main` method as follows:

```
static void Main(string[] args)
{
    if (Environment.UserInteractive)
    {
        MyNewService service1 = new MyNewService(args);
        service1.TestStartupAndStop(args);
    }
    else
    {
        // Put the body of your old Main method here.
    }
}
```

3. In the **Application** tab of the project's properties, set the **Output type** to **Console Application**.
4. Choose **Start Debugging** (F5).
5. To run the program as a Windows Service again, install it and start it as usual for a Windows Service. It's not necessary to reverse these changes.

In some cases, such as when you want to debug an issue that occurs only on system startup, you have to use the Windows debugger. Install [Debugging Tools for Windows](#) and see [How to debug Windows Services](#).

## See Also

[Introduction to Windows Service Applications](#)

[How to: Install and Uninstall Services](#)

[How to: Start Services](#)

[Debugging a Service](#)

© 2016 Microsoft

# How to: Log Information About Services

## .NET Framework (current version)

By default, all Windows Service projects have the ability to interact with the Application event log and write information and exceptions to it. You use the [AutoLog](#) property to indicate whether you want this functionality in your application. By default, logging is turned on for any service you create with the Windows Service project template. You can use a static form of the [EventLog](#) class to write service information to a log without having to create an instance of an [EventLog](#) component or manually register a source.

The installer for your service automatically registers each service in your project as a valid source of events with the Application log on the computer where the service is installed, when logging is turned on. The service logs information each time the service is started, stopped, paused, resumed, installed, or uninstalled. It also logs any failures that occur. You do not need to write any code to write entries to the log when using the default behavior; the service handles this for you automatically.

If you want to write to an event log other than the Application log, you must set the [AutoLog](#) property to **false**, create your own custom event log within your services code, and register your service as a valid source of entries for that log. You must then write code to record entries to the log whenever an action you're interested in occurs.

### Note

If you use a custom event log and configure your service application to write to it, you must not attempt to access the event log before setting the service's [ServiceName](#) property in your code. The event log needs this property's value to register your service as a valid source of events.

## To enable default event logging for your service

- Set the [AutoLog](#) property for your component to **true**.

### Note

By default, this property is set to **true**. You do not need to set this explicitly unless you are building more complex processing, such as evaluating a condition and then setting the [AutoLog](#) property based on the result of that condition.

## To disable event logging for your service

- Set the [AutoLog](#) property for your component to **false**.

**VB**

```
Me.AutoLog = False
```

## To set up logging to a custom log

1. Set the [AutoLog](#) property to **false**.

### Note

You must set [AutoLog](#) to false in order to use a custom log.

2. Set up an instance of an [EventLog](#) component in your Windows Service application.
3. Create a custom log by calling the [CreateEventSource](#) method and specifying the source string and the name of the log file you want to create.
4. Set the [Source](#) property on the [EventLog](#) component instance to the source string you created in step 3.
5. Write your entries by accessing the [WriteEntry](#) method on the [EventLog](#) component instance.

The following code shows how to set up logging to a custom log.

### Note

In this code example, an instance of an [EventLog](#) component is named `eventLog1` (`EventLog1` in Visual Basic). If you created an instance with another name in step 2, change the code accordingly.

### VB

```
Public Sub New()  
    ' Turn off autologging  
    Me.AutoLog = False  
    ' Create a new event source and specify a log name that  
    ' does not exist to create a custom log  
    If Not System.Diagnostics.EventLog.SourceExists("MySource") Then  
        System.Diagnostics.EventLog.CreateEventSource("MySource",  
            "MyLog")  
    End If  
    ' Configure the event log instance to use this source name  
    EventLog1.Source = "MySource"  
End Sub
```

### VB

```
Protected Overrides Sub OnStart(ByVal args() As String)  
    ' Write an entry to the log you've created.
```

```
EventLog1.WriteEntry("In Onstart.")  
End Sub
```

## See Also

[Introduction to Windows Service Applications](#)

© 2016 Microsoft

# Troubleshooting: Debugging Windows Services

## .NET Framework (current version)

When you debug a Windows service application, your service and the **Windows Service Manager** interact. The **Service Manager** starts your service by calling the [OnStart](#) method, and then waits 30 seconds for the [OnStart](#) method to return. If the method does not return in this time, the manager shows an error that the service cannot be started.

When you debug the [OnStart](#) method as described in [How to: Debug Windows Service Applications](#), you must be aware of this 30-second period. If you place a breakpoint in the [OnStart](#) method and do not step through it in 30 seconds, the manager will not start the service.

## See Also

[How to: Debug Windows Service Applications](#)  
[Introduction to Windows Service Applications](#)

© 2016 Microsoft

# Troubleshooting: Service Application Won't Install

## .NET Framework (current version)

If your service application will not install correctly, check to make sure that the [ServiceName](#) property for the service class is set to the same value as is shown in the installer for that service. The value must be the same in both instances in order for your service to install correctly.

### Note

You can also look at the installation logs to get feedback on the installation process.

You should also check to determine whether you have another service with the same name already installed. Service names must be unique for installation to succeed.

## See Also

[Introduction to Windows Service Applications](#)