Contents

Develop Windows service apps

2/20/2019 • 2 minutes to read • Edit Online

Using Visual Studio or the .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.

> **NOTE**
>
> In Visual Studio you can create a service in managed code in Visual C# or Visual Basic, which can 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.

Create Projects from Templates - Describes the projects types used in this chapter and how to choose between them.

Introduction to Windows Service Applications

4/8/2019 • 6 minutes to read • Edit Online

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.

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

- 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

Tutorial: Create a Windows service app

4/24/2019 • 15 minutes to read • Edit Online

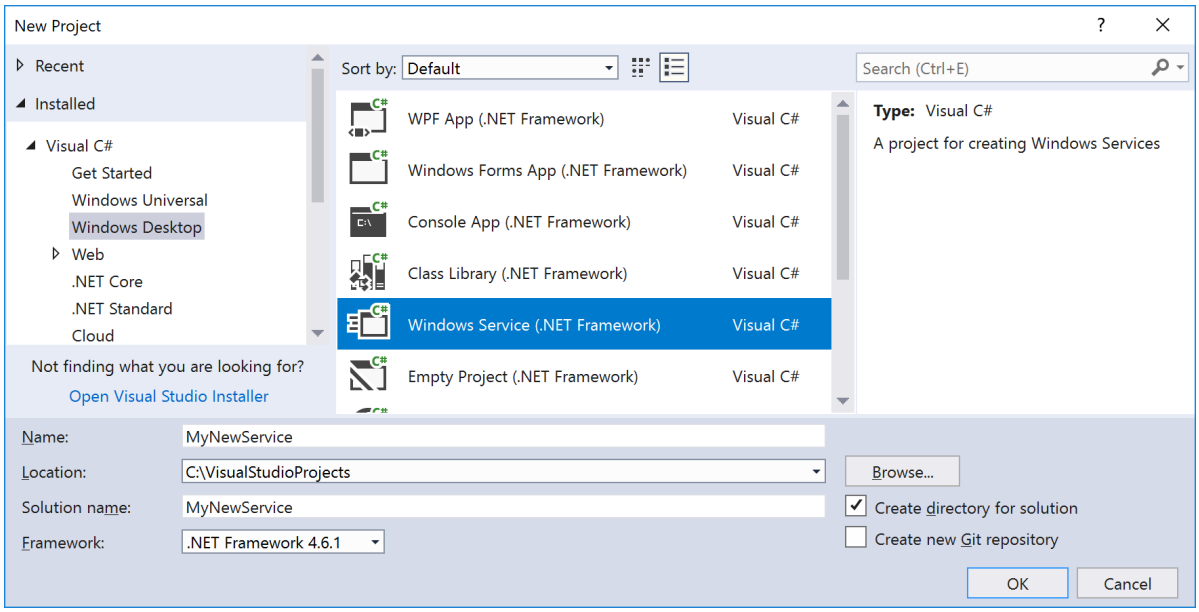
This article demonstrates how to create a Windows service app in Visual Studio that writes messages to an event log.

Create a service

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

1. From the Visual Studio **File** menu, select **New** > **Project** (or press **Ctrl**+**Shift**+**N**) to open the **New Project** window.

2. Navigate to and select the **Windows Service (.NET Framework)** project template. To find it, expand **Installed** and **Visual C#** or **Visual Basic**, then select **Windows Desktop**. Or, enter *Windows Service* in the search box on the upper right and press **Enter**.



> **NOTE**
>
> If you don't see the **Windows Service** template, you may need to install the **.NET desktop development** workload:
>
> In the **New Project** dialog, select **Open Visual Studio Installer** on the lower left. Select the **.NET desktop development** workload, and then select **Modify**.

3. For **Name**, enter *MyNewService*, and then select **OK**.

   The **Design** tab appears (**Service1.cs [Design]** or **Service1.vb [Design]**).

   The project template includes a component class named `Service1` that inherits from System.ServiceProcess.ServiceBase. It includes much of the basic service code, such as the code to start the service.

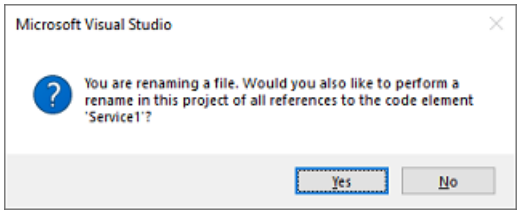
Rename the service

Rename the service from **Service1** to **MyNewService**.

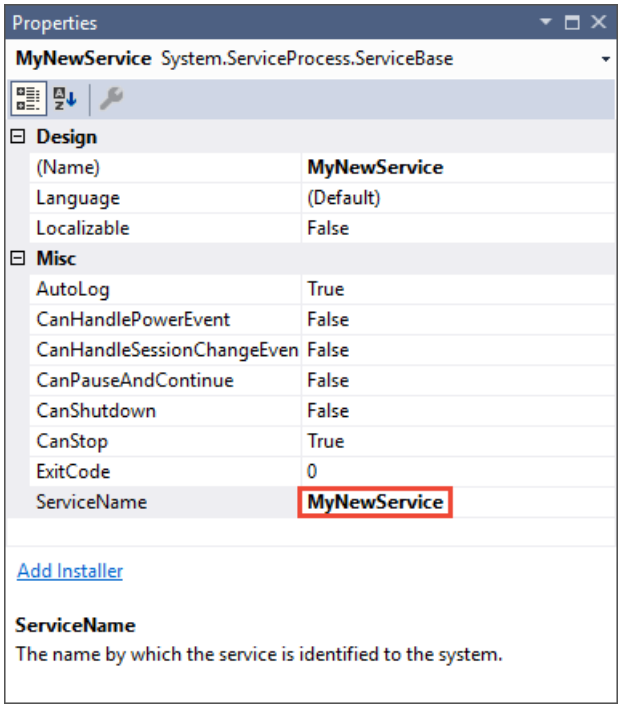
1. In **Solution Explorer**, select **Service1.cs**, or **Service1.vb**, and choose **Rename** from the shortcut menu. Rename the file to **MyNewService.cs**, or **MyNewService.vb**, and then press **Enter**

   A pop-up window appears asking whether you would like to rename all references to the code element *Service1*.

2. In the pop-up window, select **Yes**.



3. In the **Design** tab, select **Properties** from the shortcut menu. From the **Properties** window, change the **ServiceName** value to *MyNewService*.



4. Select **Save All** from the **File** menu.

Add features to the service

In this section, you add a custom event log to the Windows service. The EventLog component is an example of the type of component you can add to a Windows service.

Add custom event log functionality

1. In **Solution Explorer**, from the shortcut menu for **MyNewService.cs**, or **MyNewService.vb**, choose **View Designer**.

2. In **Toolbox**, expand **Components**, and then drag the **EventLog** component to the **Service1.cs [Design]**, or **Service1.vb [Design]** tab.

3. In **Solution Explorer**, from the shortcut menu for **MyNewService.cs**, or **MyNewService.vb**, choose **View Code**.

4. Define a custom event log. For C#, edit the existing `MyNewService()` constructor; for Visual Basic, add the `New()` constructor:

```
public MyNewService()
{
    InitializeComponent();
    eventLog1 = new System.Diagnostics.EventLog();
    if (!System.Diagnostics.EventLog.SourceExists("MySource"))
    {
        System.Diagnostics.EventLog.CreateEventSource(
            "MySource","MyNewLog");
    }
    eventLog1.Source = "MySource";
    eventLog1.Log = "MyNewLog";
}
```

```
' 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
```

5. Add a `using` statement to **MyNewService.cs** (if it doesn't already exist), or an `Imports` statement **MyNewService.vb**, for the System.Diagnostics namespace:

```
using System.Diagnostics;
```

```
Imports System.Diagnostics
```

6. Select **Save All** from the **File** menu.

Define what occurs when the service starts

In the code editor for **MyNewService.cs** or **MyNewService.vb**, locate the OnStart method; Visual Studio automatically created an empty method definition when you created the project. Add code that writes an entry to the event log when the service starts:

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

```
' 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
```

Polling

Because a service application is designed to be long-running, it usually polls or monitors the system, which you set up in the OnStart method. The `OnStart` method must return to the operating system after the service's

operation has begun so that the system isn't blocked.

To set up a simple polling mechanism, use the System.Timers.Timer component. The timer raises an Elapsed event at regular intervals, at which time your service can do its monitoring. You use the Timer component as follows:

- Set the properties of the Timer component in the `MyNewService.OnStart` method.
- Start the timer by calling the Start method.

Set up the polling mechanism.

1. Add the following code in the `MyNewService.OnStart` event to set up the polling mechanism:

```
// Set up a timer that triggers every minute.
Timer timer = new Timer();
timer.Interval = 60000; // 60 seconds
timer.Elapsed += new ElapsedEventHandler(this.OnTimer);
timer.Start();
```

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

2. Add a `using` statement to **MyNewService.cs**, or an `Imports` statement to **MyNewService.vb**, for the System.Timers namespace:

```
using System.Timers;
```

```
Imports System.Timers
```

3. In the `MyNewService` class, add the `OnTimer` method to handle the Timer.Elapsed event:

```
public void OnTimer(object sender, ElapsedEventArgs args)
{
    // TODO: Insert monitoring activities here.
    eventLog1.WriteEntry("Monitoring the System", EventLogEntryType.Information, eventId++);
}
```

```
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
```

4. In the `MyNewService` class, add a member variable. It contains the identifier of the next event to write into the event log:

```
private int eventId = 1;
```

```
Private eventId As Integer = 1
```

Instead of running all your work on the main thread, you can run tasks by using background worker threads. For

more information, see System.ComponentModel.BackgroundWorker.

Define what occurs when the service is stopped

Insert a line of code in the OnStop method that adds an entry to the event log when the service is stopped:

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

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

Define other actions for the service

You can override the OnPause, OnContinue, and OnShutdown methods to define additional processing for your component.

The following code shows how you to override the OnContinue method in the `MyNewService` class:

```
protected override void OnContinue()
{
    eventLog1.WriteEntry("In OnContinue.");
}
```

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

Set service status

Services report their status to the Service Control Manager so that a user can tell whether a service is functioning correctly. By default, a service that inherits from ServiceBase reports a limited set of status settings, which include SERVICE_STOPPED, SERVICE_PAUSED, and SERVICE_RUNNING. If a service takes a while to start up, it's useful to report a SERVICE_START_PENDING status.

You can implement the SERVICE_START_PENDING and SERVICE_STOP_PENDING status settings by adding code that calls the Windows SetServiceStatus function.

Implement service pending status

1. Add a `using` statement to **MyNewService.cs**, or an `Imports` statement to **MyNewService.vb**, for the System.Runtime.InteropServices namespace:

   ```
   using System.Runtime.InteropServices;
   ```

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

2. Add the following code to **MyNewService.cs**, or **MyNewService.vb**, to declare the `ServiceState` values and to add a structure for the status, which you'll use in a platform invoke call:

```
public enum ServiceState
{
    SERVICE_STOPPED = 0x00000001,
    SERVICE_START_PENDING = 0x00000002,
    SERVICE_STOP_PENDING = 0x00000003,
    SERVICE_RUNNING = 0x00000004,
    SERVICE_CONTINUE_PENDING = 0x00000005,
    SERVICE_PAUSE_PENDING = 0x00000006,
    SERVICE_PAUSED = 0x00000007,
}

[StructLayout(LayoutKind.Sequential)]
public struct ServiceStatus
{
    public int dwServiceType;
    public ServiceState dwCurrentState;
    public int dwControlsAccepted;
    public int dwWin32ExitCode;
    public int dwServiceSpecificExitCode;
    public int dwCheckPoint;
    public int dwWaitHint;
};
```

```
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
```

NOTE

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.

3. In the `MyNewService` class, declare the SetServiceStatus function by using platform invoke:

```
[DllImport("advapi32.dll", SetLastError = true)]
private static extern bool SetServiceStatus(System.IntPtr handle, ref ServiceStatus serviceStatus);
```

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

4. To implement the SERVICE_START_PENDING status, add the following code to the beginning of the OnStart method:

```
// Update the service state to Start Pending.
ServiceStatus serviceStatus = new ServiceStatus();
serviceStatus.dwCurrentState = ServiceState.SERVICE_START_PENDING;
serviceStatus.dwWaitHint = 100000;
SetServiceStatus(this.ServiceHandle, ref serviceStatus);
```

```
' 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 the end of the `OnStart` method to set the status to SERVICE_RUNNING:

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

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

6. (Optional) If OnStop is a long-running method, repeat this procedure in the `OnStop` method. Implement the SERVICE_STOP_PENDING status and return the SERVICE_STOPPED status before the `OnStop` method exits.

   For example:

```
// Update the service state to Stop Pending.
ServiceStatus serviceStatus = new ServiceStatus();
serviceStatus.dwCurrentState = ServiceState.SERVICE_STOP_PENDING;
serviceStatus.dwWaitHint = 100000;
SetServiceStatus(this.ServiceHandle, ref serviceStatus);

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

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

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

Add installers to the service

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

1. In **Solution Explorer**, from the shortcut menu for **MyNewService.cs**, or **MyNewService.vb**, choose **View Designer**.

2. In the **Design** view, select the background area, then choose **Add Installer** from the shortcut menu.

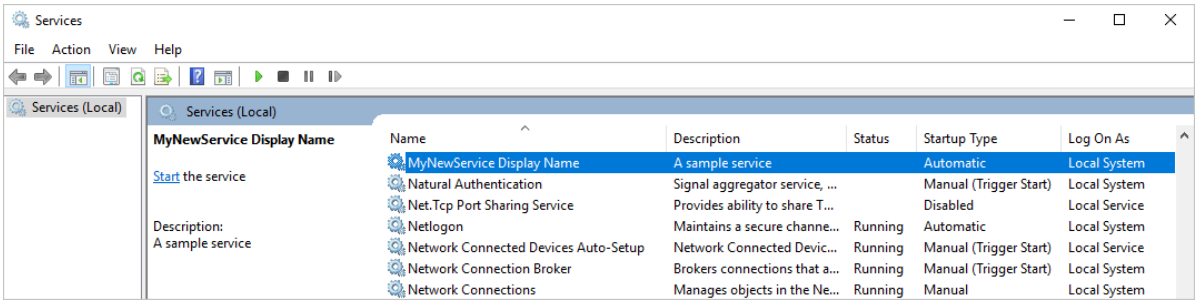
   By default, Visual Studio adds a component class named `ProjectInstaller`, which contains two installers, to your project. These installers are for your service and for the service's associated process.

3. In the **Design** view for **ProjectInstaller**, select **serviceInstaller1** for a Visual C# project, or **ServiceInstaller1** for a Visual Basic project, then choose **Properties** from the shortcut menu.

4. In the **Properties** window, verify the ServiceName property is set to **MyNewService**.

5. Add text to the Description property, such as *A sample service*.

   This text appears in the **Description** column of the **Services** window and describes the service to the user.

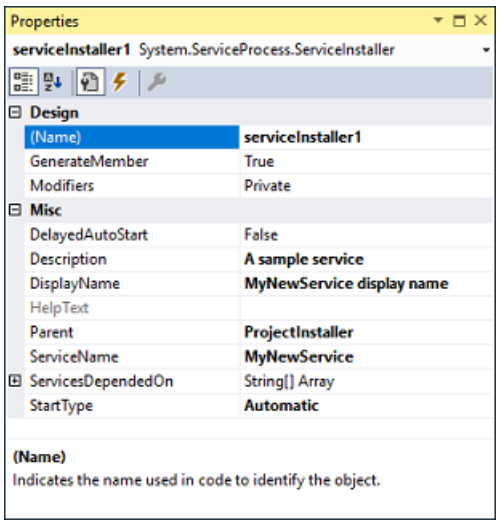


6. Add text to the DisplayName property. For example, *MyNewService Display Name*.

   This text appears in the **Display Name** column of the **Services** window. This name can be different from the ServiceName property, which is the name the system uses (for example, the name you use for the `net start` command to start your service).

7. Set the StartType property to Automatic from the drop-down list.

8. When you're finished, the **Properties** windows should look like the following figure:



9. In the **Design** view for **ProjectInstaller**, choose **serviceProcessInstaller1** for a Visual C# project, or **ServiceProcessInstaller1** for a Visual Basic project, then choose **Properties** from the shortcut menu. Set the Account property to LocalSystem from the drop-down list.

   This setting installs the service and runs it by using the local system account.

> **IMPORTANT**
>
> 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.

(Optional) Set startup parameters

> **NOTE**
>
> Before you decide to add startup parameters, consider whether it's the best way to pass information to your service. Although they're easy to use and parse, and a user can easily override them, they might be harder for a user to discover and use without documentation. Generally, if your service requires more than just a few startup parameters, you should use the registry or a configuration file instead.

A Windows service can accept command-line arguments, or startup parameters. When you add code to process startup parameters, a user can start your service with their own custom startup parameters in the service properties window. However, these startup parameters aren't persisted the next time the service starts. To set startup parameters permanently, set them in the registry.

Each Windows service has a registry entry under the **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services** subkey. Under each service's subkey, 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 sample code, see ConfigurationManager.AppSettings.

To add startup parameters

1. Select **Program.cs**, or **MyNewService.Designer.vb**, then choose **View Code** from the shortcut menu. In the `Main` method, change the code to add an input parameter and pass it to the service constructor:

```
static void Main(string[] args)
{
    ServiceBase[] ServicesToRun;
    ServicesToRun = new ServiceBase[]
    {
        new MyNewService(args)
    };
    ServiceBase.Run(ServicesToRun);
}
```

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

2. In **MyNewService.cs**, or **MyNewService.vb**, change the `MyNewService` constructor to process the input parameter as follows:

```
using System.Diagnostics;

public MyNewService(string[] args)
{
    InitializeComponent();

    string eventSourceName = "MySource";
    string logName = "MyNewLog";

    if (args.Length > 0)
    {
       eventSourceName = args[0];
    }

    if (args.Length > 1)
    {
        logName = args[1];
    }

    eventLog1 = new EventLog();

    if (!EventLog.SourceExists(eventSourceName))
    {
        EventLog.CreateEventSource(eventSourceName, logName);
    }

    eventLog1.Source = eventSourceName;
    eventLog1.Log = logName;
}
```

```
Imports System.Diagnostics

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 EventLog()
    If (Not EventLog.SourceExists(eventSourceName)) Then
        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 startup parameters that the user supplies. If no arguments are supplied, it uses default values.

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

```
protected override void OnBeforeInstall(IDictionary savedState)
{
    string parameter = "MySource1\" \"MyLogFile1";
    Context.Parameters["assemblypath"] = "\"" + Context.Parameters["assemblypath"] + "\" \"" +
parameter + "\"";
    base.OnBeforeInstall(savedState);
}
```

```
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
```

Typically, this value contains the full path to the executable for the Windows service. For the service to start up correctly, the user must supply quotation marks for the path and each individual parameter. A user can change the parameters in the **ImagePath** registry entry to change the startup parameters for the Windows service. However, a better way is to change the value programmatically and expose the functionality in a user-friendly way, such as by using a management or configuration utility.

Build the service

1. In **Solution Explorer**, choose **Properties** from the shortcut menu for the **MyNewService** project.

   The property pages for your project appear.

2. On the **Application** tab, in the **Startup object** list, choose **MyNewService.Program**, or **Sub Main** for Visual Basic projects.

3. To build the project, in **Solution Explorer**, choose **Build** from the shortcut menu for your project (or press **Ctrl**+**Shift**+**B**).

Install the service

Now that you've built the Windows service, you can install it. To install a Windows service, you must have administrator credentials on the computer where it's installed.

1. Open Developer Command Prompt for Visual Studio with administrative credentials. From the Windows **Start** menu, select **Developer Command Prompt for VS 2017** in the Visual Studio folder, then select **More** > **Run as Administrator** from the shortcut menu.

2. In the **Developer Command Prompt for Visual Studio** window, navigate to the folder that contains your project's output (by default, the *\bin\Debug* subdirectory of your project).

3. Enter the following command:

   ```
   installutil MyNewService.exe
   ```

   If the service installs successfully, the command reports success.

   If the system can't 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 64-bit version is *%windir%\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe*.

If the **installutil.exe** process fails, 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:

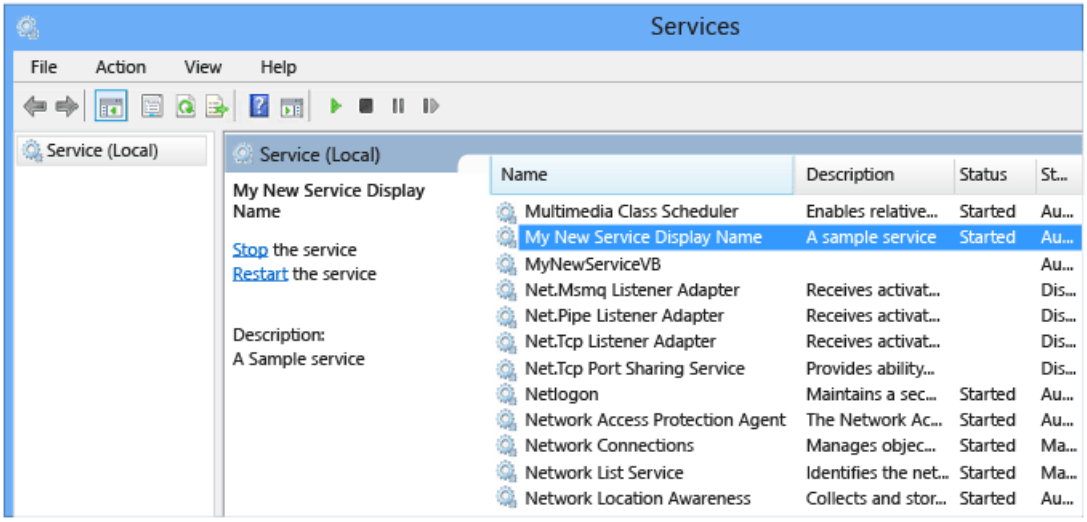
- The RunInstallerAttribute class isn't present on the `ProjectInstaller` class.
- The attribute isn't set to `true`.
- The `ProjectInstaller` class isn't defined as `public`.

For more information, see How to: Install and uninstall services.

Start and run the service

1. In Windows, open the **Services** desktop app. Press **Windows**+**R** to open the **Run** box, enter *services.msc*, and then press **Enter** or select **OK**.

   You should see your service listed in **Services**, displayed alphabetically by the display name that you set for it.



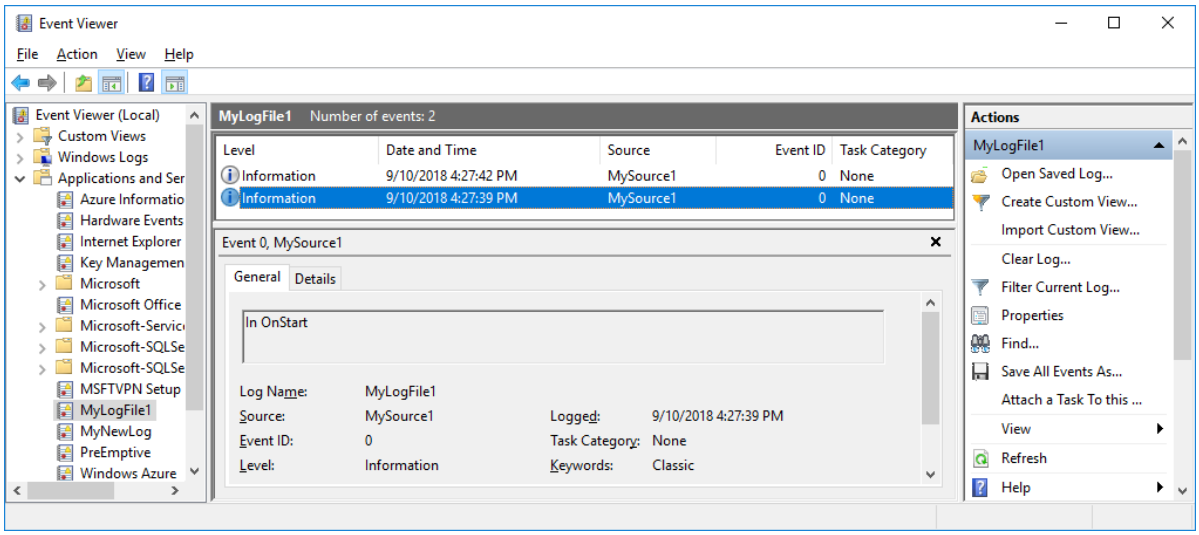
2. To start the service, choose **Start** from the service's shortcut menu.
3. To stop the service, choose **Stop** from the service's shortcut menu.
4. (Optional) From the command line, use the commands **net start <service name>** and **net stop <service name>** to start and stop your service.

Verify the event log output of your service

1. In Windows, open the **Event Viewer** desktop app. Enter *Event Viewer* in the Windows search bar, and then select **Event Viewer** from the search results.

   > **TIP**
   > In Visual Studio, you can access event logs by opening **Server Explorer** from the **View** menu (or press **Ctrl**+**Alt**+**S**) and expanding the **Event Logs** node for the local computer.

2. In **Event Viewer**, expand **Applications and Services Logs**.
3. Locate the listing for **MyNewLog** (or **MyLogFile1** if you followed the procedure to add command-line arguments) and expand it. You should see the entries for the two actions (start and stop) that your service performed.



Clean up resources

If you no longer need the Windows service app, you can remove it.

1. Open **Developer Command Prompt for Visual Studio** with administrative credentials.
2. In the **Developer Command Prompt for Visual Studio** window, navigate to the folder that contains your project's output.
3. Enter the following command:

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

   If the service uninstalls successfully, the command reports that your service was successfully removed. For more information, see How to: Install and uninstall services.

Next steps

Now that you've created the service, you can:

- Create a standalone setup program for others to use to install your Windows service. Use the WiX Toolset to create an installer for a Windows service. For other ideas, see Create an installer package.
- Explore the ServiceController component, which enables you to send commands to the service you've installed.
- Instead of creating the event log when the application runs, use an installer to create an event log when you install the application. The event log is deleted by the installer when you uninstall the application. For more information, see EventLogInstaller.

See also

- Windows service applications
- Introduction to Windows service applications
- How to: Debug Windows service applications
- Services (Windows)

Service Application Programming Architecture

4/8/2019 • 2 minutes to read • Edit Online

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
OnStart	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.
OnPause	Indicate what should happen when your service is paused.
OnStop	Indicate what should happen when your service stops running.
OnContinue	Indicate what should happen when your service resumes normal functioning after being paused.
OnShutdown	Indicate what should happen just prior to your system shutting down, if your service is running at that time.
OnCustomCommand	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:

```
System.ServiceProcess.ServiceBase[] ServicesToRun;
ServicesToRun = new System.ServiceProcess.ServiceBase[]
    { new Service1() };
System.ServiceProcess.ServiceBase.Run(ServicesToRun);
```

```
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

How to: Create Windows Services

4/9/2019 • 2 minutes to read • Edit Online

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:

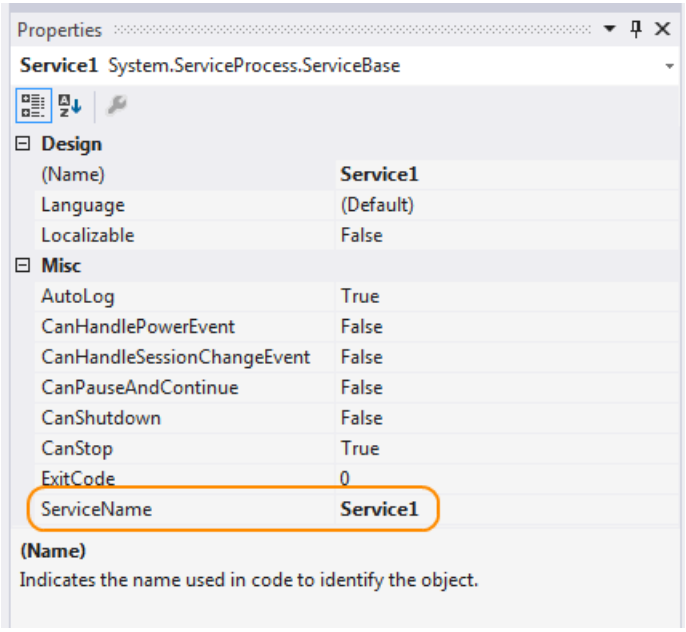
- Set the ServiceName property.
- Create the necessary installers for your service application.
- Override and specify code for the OnStart and OnStop methods to customize the ways in which your service behaves.

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.

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

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

4/9/2019 • 2 minutes to read • Edit Online

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:

```
public class UserService1 : System.ServiceProcess.ServiceBase
{
}
```

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

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

```
public UserService1()
{
    this.ServiceName = "MyService2";
    this.CanStop = true;
    this.CanPauseAndContinue = true;
    this.AutoLog = true;
}
```

```
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:

```
public static void Main()
{
    System.ServiceProcess.ServiceBase.Run(new UserService1());
}
```

```
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.

```
protected override void OnStart(string[] args)
{
    // Insert code here to define processing.
}
```

```
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

4/9/2019 • 2 minutes to read • Edit Online

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 Personalize the Visual Studio IDE.

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.
5. To determine how your service will be started, click the ServiceInstaller component and set the StartType property to the appropriate value.

VALUE	RESULT
Manual	The service must be manually started after installation. For more information, see How to: Start Services.

VALUE	RESULT
Automatic	The service will start by itself whenever the computer reboots.
Disabled	The service cannot be started.

6. 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.
7. Override any methods for which you need to perform custom processing.
8. 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

4/9/2019 • 2 minutes to read • Edit Online

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

How to: Install and uninstall Windows services

4/9/2019 • 2 minutes to read • Edit Online

If you're developing a Windows service with the .NET Framework, you can quickly install your service app by using the *InstallUtil.exe* command-line utility. Developers who want to release a Windows service that users can install and uninstall should use InstallShield. For more information, see Create an installer package (Windows client).

> **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 **Apps** in Settings to uninstall that program. Note that many services are integral parts of Windows; if you remove them, you might cause system instability.

To use the steps in this article, you first need to add a service installer to your Windows service. For more information, see Walkthrough: Creating a Windows service app.

You can't run Windows service projects directly from the Visual Studio development environment by pressing F5. Before you can run the project, you must install the service in the project.

> **TIP**
> You can use **Server Explorer** to verify that you've installed or uninstalled your service. For more information, see How to use Server Explorer in Visual Studio.

Install your service manually

1. From the **Start** menu, select the **Visual Studio <*version*>** directory, then select **Developer Command Prompt for VS <*version*>**.

   The Developer Command Prompt for Visual Studio 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 Developer Command Prompt for Visual Studio, *InstallUtil.exe* should be on the system path. Otherwise, you can add it to the path, or use the fully qualified path to invoke it. This tool is installed with the .NET Framework in *%WINDIR%\Microsoft.NET\Framework[64]\<framework_version>*.

   For example:

   - For the 32-bit version of the .NET Framework 4 or 4.5 and later, if your Windows installation directory is *C:\Windows*, the default 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 and later, the default path is *C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe*.

Uninstall your service manually

1. From the **Start** menu, select the **Visual Studio <*version*>** directory, then select **Developer Command Prompt for VS <*version*>**.

The Developer Command Prompt for Visual Studio appears.

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

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

3. After the executable for a service is deleted, the service might still be present in the registry. If that's the 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

4/9/2019 • 2 minutes to read • Edit Online

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	**Automatic**
When an explicit user action starts the service	**Manual**

> **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: Pause a Windows Service (Visual Basic)

4/8/2019 • 2 minutes to read • Edit Online

This example uses the ServiceController component to pause the IIS Admin service on the local computer.

Example

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

```
' 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)
- 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 to set permissions in the ServiceControllerPermission.

Access to service information may be restricted by using the PermissionState to set permissions in the SecurityPermission.

See also

- ServiceController
- ServiceControllerStatus
- WaitForStatus

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

How to: Continue a Windows Service (Visual Basic)

4/8/2019 • 2 minutes to read • Edit Online

This example uses the ServiceController component to continue the IIS Admin service on the local computer.

Example

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

```
' 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)

How to: Debug Windows Service Applications

4/9/2019 • 5 minutes to read • Edit Online

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. Download the Windows Driver Kit (WDK) 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

How to: Log Information About Services

4/9/2019 • 3 minutes to read • Edit Online

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`.

```
this.AutoLog = false;
```

```
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.

```
public UserService2()
{
    eventLog1 = new System.Diagnostics.EventLog();
    // Turn off autologging

    this.AutoLog = false;
    // create an event source, specifying the name of a log that
    // does not currently exist to create a new, custom log
    if (!System.Diagnostics.EventLog.SourceExists("MySource"))
    {
        System.Diagnostics.EventLog.CreateEventSource(
            "MySource","MyLog");
    }
    // configure the event log instance to use this source name
    eventLog1.Source = "MySource";
    eventLog1.Log = "MyLog";
}
```

```
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
```

```
protected override void OnStart(string[] args)
{
    // write an entry to the log
    eventLog1.WriteEntry("In OnStart.");
}
```

```
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

Troubleshooting: Debugging Windows Services

4/8/2019 • 2 minutes to read • Edit Online

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

Troubleshooting: Service Application Won't Install

4/8/2019 • 2 minutes to read • Edit Online

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