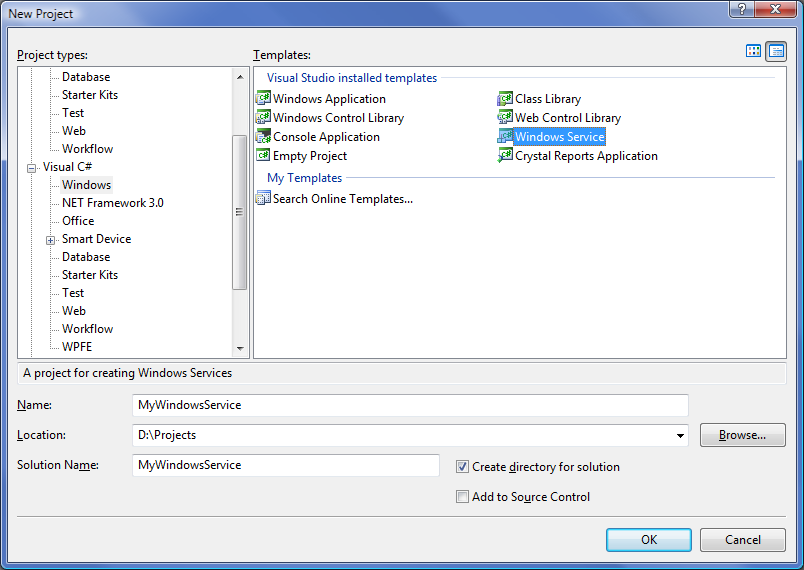
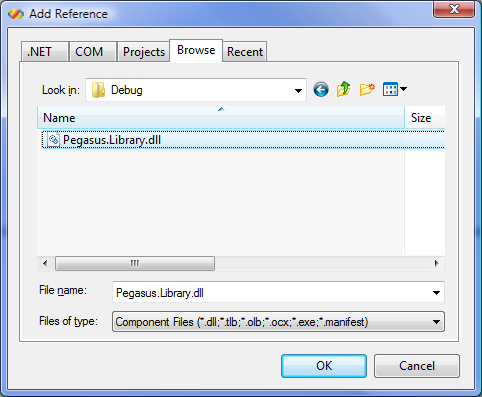
Windows Services Made Simple

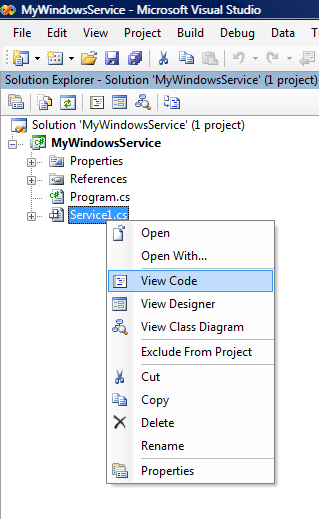
I’ve been programming backend services for some years now and have come up with a .NET base class that makes writing a Windows Service just like building an application.

I’ve incorporated the classes into the Pegasus Library (<http://www.codeplex.com/Pegasus>). You can download the code and sample from there.

# Creating a Windows Service

To build a service is not that hard since .NET came around. Let’s setup a project in Visual Studio 2005.

1. Create a new project by selecting File | New | Project.
2. Select the C#, Windows Service template. 
3. Select the name of the project and location and click OK.
4. Add a reference to the Pegasus.Library.dll by selecting Project | Add Reference. 
5. Navigate to the location where you placed the Pegasus.Library.dll, select it and click OK.
6. By default Visual Studio will open component objects (Like Service1) in design mode. We need to change the code. Open the code by right mouse clicking on Service1.cs and select the View Code item.



1. Add the Using statement for the Pegasus.ServiceProcess namespace.
2. Add 3 string constants for the name of the service, the display name of the service, and a brief description of the service. The name of the service is the name that Windows uses internally to keep track of the service in the registry and the service control manager. The name therefore should not contain spaces or odd characters. The display name and the description are what the user sees, so they should be user friendly.
3. Change the base class ServiceBase to the Pegasus class ServiceExcutable.
4. The ServiceExecutable class has a constructor that takes the name of the service, so we need to explicitly call the base constructor and use the string const we defined for the service name.

using System;

using System.ServiceProcess;

using Pegasus.ServiceProcess;

namespace MyWindowsService

{

public partial class Service1 : ServiceExecutable

{

// Public Const Values

public const string MYSERVICE\_NAME = "MyWinSvc";

public const string MYSERVICE\_DISPLAYNAME = "My Windows Service";

public const string MYSERVICE\_DESCRIPTION = "This service is cool.";

public Service1()

: base( MYSERVICE\_NAME )

{

InitializeComponent();

}

protected override void OnStart( string[] args )

{

}

protected override void OnStop()

{

}

}

}

1. Now we just have one last thing to do. Change the Main() function in the program.cs file to use the new RunService method in place of the one provided by .NET.

public static void Main( string[] args )

{

Service1 service = new Service1();

service.RunService( args );

}

Now we have our Window Service…

# Things to Think About

At this point we have a service but it doesn’t do anything. As you design the rest of the functionality of a Windows Service there are a couple of things you need to keep in mind.

**1 – Keep Service Constructor Simple**

The constructor is called when the application's executable runs, not when the service runs. Windows may load the EXE into memory (and thus call the Main() method) but not start the service right away.

When the services continue from a paused status, for example, the constructor is not called again because the Windows Service Control Manager (SCM) already holds the object in memory. If OnStop() releases resources allocated in the constructor rather than in OnStart(), the needed resources would not be created again the second time the service is called.

So because of this you should not use the constructor to perform startup processing, rather use the OnStart() method to handle all initialization of your service and the OnStop() to release any resources.

**2 - You don’t own the main thread.**

An application owns all of the threads in the process and a developer can pretty much do as they want. In a service the main thread is owned by the SCM and it’s just loaned out to your service as needed (the SCM uses a thread pool to talk to services). In fact it is possible to have the SCM create the service object, call the OnStart(), and call the OnStop() methods all on different threads.

To avoid any conflict that this can cause it’s a good idea to spawn a thread in the OnStart() method and do the work of the service on your own thread. For our example we add the following code to our service class.

private Thread m\_thread = null;

private AutoResetEvent m\_done = new AutoResetEvent( false );

protected override void OnStart( string[] args )

{

m\_done.Reset();

m\_thread = new Thread( new ParameterizedThreadStart( args ) );

m\_thread.Name = "MyWinSvc Working Thread";

m\_thread.Start();

}

protected override void OnStop()

{

m\_done.Set();

if( !m\_thread.Join( 2000 ) )

{

RequestAdditionalTime( 5000 );

}

m\_thread = null;

}

private void DoWork( object parameters )

{

while( !m\_done.WaitOne( 0, false ) )

{

// This is where the real work happens

Thread.Sleep( 1000 );

}

}

For our example the service will just sleep for 1 second and then check to see if the work is done. If not it will sleep for another second and check again, etc, etc, etc…

**3 – The SCM is not very patient.**

When the SCM calls the OnStart(), OnStop(), OnPause(), or OnContinue() method, you only have so long (60 sec or so) before you must return. If you do not return within this time, then the SCM will mark that service as not responding. After a while it can out right kill the process you are running in. If you need more time to process things during these methods you can call the RequestAdditionalTime() method and tell the SCM that things are talking a bit longer that it expects. (See the OnStop() method in the example above)

public void RequestAdditionalTime( int milliseconds );

**4 –Pause and Continue**

For most services the OnStart() and OnStop() methods will be enough for a service to startup and stop. However some services may need to have the ability to pause and continue. To support this functionality we must tell the SCM that the service supports the Pause and Continue methods. This is done by setting the CanPauseAndContinue property to true in the OnStart() method.

protected override void OnStart( string[] args )

{

CanPauseAndContinue = true;

…

}

You can then override the OnPause() and OnContinue() methods to handle the proper functionality for each context. In our example we will suspend our working thread when pause is called and resume the thread when continue is called. (Yes, I know that Suspend and Resume are deprecated but it’s just an example)

protected override void OnPause()

{

m\_thread.Suspend();

}

protected override void OnContinue()

{

m\_thread.Resume();

}

**5 – Shutdown, Power Events, and Session Events.**

There are three other sets of system event methods that you can use to tell what is happening on the system. Each of these has their own CanXXXX property that must be set to true for the SCM to call the event handler.

System Shutdown

public bool CanShutdown

protected virtual void OnShutdown();  
  
Called when the system is shutting down. This is different that a stop event. In this case not only is the service being stopped but the whole system is being shutdown.

System Power Events

public bool CanHandlePowerEvent

protected virtual bool OnPowerEvent( PowerBroadcastStatus powerStatus )

Called when the power status of the system is changing. A service can know that there has been some type of power event, like a power outage or the system is on batteries, or the batteries are low, etc… This method has a return value of type Boolean. The needs of your service determine what value to return. For example, if a QuerySuspend broadcast status is passed, you could cause your service to reject the query by returning false.

System Session Events

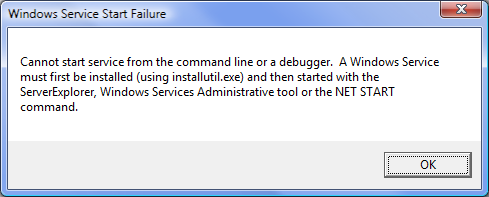
public bool CanHandlePowerEvent

protected virtual void OnSessionChange( SessionChangeDescription changeDescription )

Called when a system session event occurs. Session events are caused when a new user logs in or out of the console or the terminal services. The SessionChangedDescription is a structure that contains the reason for the event and the session id the event occurred in.

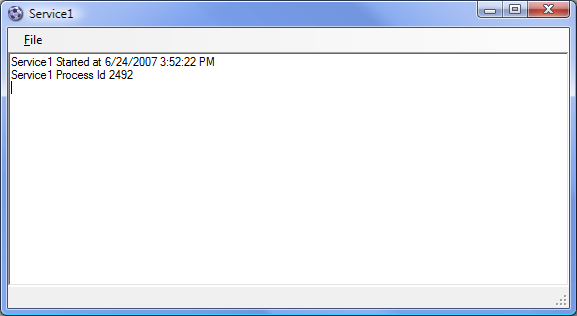
# Debugging the Service

Up to this point we have not done anything Microsoft did not already provide in the ServiceBase class. But here is where the ServiceExecutable class comes into play. If you just create a .NET service project and select Debug | Start Debugging [F5] you get the following error.



Now if you follow the instruction, you have to register the service, start the service and then attach the debugger to the running process; and you have to do this each time you want to debug the code. I found this to be a pain and I’m a bit spoiled by Visual Studio’s [F5] key to just compile and run an application in debug mode. The ServiceExecutable class allows us to do just that.

The ServiceExecutable first checks to see if the process is running under a Debugger (Visual Studio in our case). If so, the ServiceExecutable class then simulates the SCM and creates a Window (for Debug/Trace messages) for the service and calls the OnStart() method. So all you need to do now is press [F5] to compile and run the service and you should see a window similar to this.



To stop the service just close the window or select File | Exit from the menu. The OnStop() methods will be called just like the service was being stopped by the SCM.

The ServiceExecutable class will register a listener interface to catch Trace/Debug messages. Pegasus also uses the Log4Net API and connects an adapter interface to catch any Log4Net messages as well. The File | Log4Net Configuration menu item brings up a dialog box that allows you to set/reconfigure the logging level at runtime.

The menu also provides a menu options for testing/debug the OnPause(), OnContinue() and OnShutdown() methods of your service.

A couple of Got-ya’s

There are a couple of things that you need to be aware of when running in debug mode this way. The Power and Session APIs do not work (even if you set the proper CanXXXX properties). To debug these you have debug the Microsoft way, register the service, start the service, and attach to the debugger. I will try to add these in a future version of the class.

The other thing to look out for is security. When running in debug mode the service is running under your account’s security context. As most developers run as an administrator when developing code, you (and thus the service) have full access to system resources and services. But if you deploy under a different account, say Local Service, then the service may not have access to a resource at runtime that it did when you where debugging it. So just keep this in mind when designing your service.

# Deploying the Service

The last item on our list is how to deploy a service. This process again starts with the basic .NET installer classes. Add the following class (MyInstaller.cs) to your service project.

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Configuration.Install;

using System.ServiceProcess;

namespace MyWindowsService

{

[RunInstaller( true )]

public class MyInstaller : Installer

{

// Local Instance Values

private ServiceProcessInstaller m\_processServiceInstaller;

private ServiceInstaller m\_windowsServiceInstaller;

public MyInstaller()

{

// Set the name of the service, display name and description

m\_windowsServiceInstaller = new ServiceInstaller();

m\_windowsServiceInstaller.ServiceName = Service1.MYSERVICE\_NAME;

m\_windowsServiceInstaller.DisplayName = Service1.MYSERVICE\_DISPLAYNAME;

m\_windowsServiceInstaller.Description = Service1.MYSERVICE\_DESCRIPTION;

// Set the account that the service will run under

m\_processServiceInstaller = new ServiceProcessInstaller();

m\_processServiceInstaller.Account = ServiceAccount.LocalSystem;

m\_processServiceInstaller.Username = null;

m\_processServiceInstaller.Password = null;

Installers.Add( m\_windowsServiceInstaller );

Installers.Add( m\_processServiceInstaller );

}

}

}

You must also add a reference to your project for the System.Configuration.Install assembly.

I’m not going to discuss the ins and outs of the Installer classes in this article, if you want to understand it more, there are a couple of good articles on Microsoft;’s MSDN web site (<http://www.msdn.com>) and on the CodeProject web site (<http://www.codeproject.com>).

To help install a service, Microsoft wrote a little application called installutil.exe which looks for the install class (the one marked with the RunInstaller attribute) and then executes an install or uninstall depending on command line options. That’s all fine and good but they forgot one thing; to include it in the .NET 2.0 redistributable packages. Now you can always copy this file around with your service or you could write a MSI install application for your service but that’s just silly.

The ServiceExecutable class provides several command line options if you run your service EXE from the command prompt.

/install

Add the proper entries in the registry and SCM to register to install the service.

/uninstall

Remove the registry and SCM entries to uninstall the service.

/app

Launch the service in “Application” mode even if it is not running under a debugger.

/? /h

Shows the different command line options and help information about each.

There is one thing we need to do to support this functionality, and that is to tell the ServiceExecutable class about our installer. To do this we just need to add the following line of code to our Service class’s constructor.

public Service1()

: base( MYSERVICE\_NAME )

{

InitializeComponent();

// Register the install class

ServiceInstallerType = typeof( MyInstaller );

}

This tells the Service Executable class which installer to use to install or uinstall the service.

# Conclusions

Windows services used to be difficult to write. With the ServiceExcutable class developers now can create Windows services easily to support a varity of backend services and business logic. Debugging and be done without having to launch the service and attach to the process, developers can debug the service like they do for applications.