

Windows Services

How to create and deploy a Windows Service



About Walter Gameiro

- In IT industry since 1990
- BS (Hons) Business Information Technology from University of Portsmouth in U.K.
- Using .Net since Beta 1
- Co-Founded Phi Engineering in 2003
- Founded Phi Consulting in 2007



About Phi Consulting

- Consultancy Business
- Specialized in Custom Software Development
 - Web Applications
 - Desktop Applications
 - Windows Services
 - Web Services
- Architected Software Solutions
- Good quality at a reasonable price
- We make Business simpler and easier

Agenda

- What is a Windows Service
- What can a Service be used for
- Considerations in Designing a Service
 - What will it do, Error Handling, Testing/Unit Testing, How will it be triggered
- Monitoring a Service
- Deployment options and considerations
 - InstallUtil, Custom Installer



Goal for today

- Build a service that creates thumbnails for any images dropped in a custom folder
- Create a monitor application to track activity in the service
- Create a custom installer for our service and monitor console
- Learn about how to build and deploy a service application

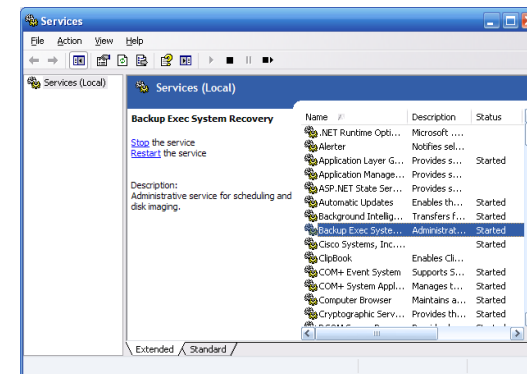
The finished application

Quick Survey

- How many code in VB.Net?
- How many code in C#?
- How many have coded Windows Services?

What is a Windows Service

- Unattended application
- Runs in background
- No User Interface (besides SCM)
- Windows equivalent to Daemons in Unix/Linux
- In .Net, an executable app derived from `System.ServiceProcess.ServiceBase`





DEMO

Creating a Service in Visual Studio



What can a Service be used for?

- Possibilities are endless
- Monitor file systems and take action
- Monitor status of web services
- Host Workflow or Communication Foundation applications
- Fetch credit reports
- Fetch updates to software
- Etc... Anything you want computer to do for you without your intervention

Considerations in Designing a Service

- How a Service works
- What will it do, what is its purpose
- Tracking Activity (no interface)
- Error Handling/Logging (no interface)
- Testing/Unit Testing
- How will it be triggered (no interface)
- **NO UI!!!** – avoid using any UI components, such as Dialog boxes or Message boxes – it will stop the service from continuing execution

Quick Survey

- How many have seen Scott Hanselman's "What Great .NET Developers Ought To Know " Interview Questions?
- How many think they know how to answer the question "What is a Windows Service and how does its lifecycle differ from a "standard" EXE?"

How a Service Works

- Out of the box Services respond to only a few system events
 - OnStart – when service receives start signal from SCM or ServiceController
 - OnStop – when service receives stop signal from SCM or ServiceController
- The most important thing to know about Services is all they do is Start and Stop
- If you understand this you can design a solid service that can handle any task
- Everything else you want your Service to do is outside the scope of Starting and Stopping

How a Service Works

- If you want to get fancy you can add support for
 - **OnCustomCommand** – when service receives a custom command from a ServiceController
 - **OnPowerEvent** – when the machine goes into or comes out of suspended mode or hibernation
 - **OnShutdown** – when the machine is about to be shutdown
 - **OnPause** – when service receives a Pause command from a ServiceController
 - **OnContinue** – when service receives a Continue command from a ServiceController
 - **OnSessionChanged** – when a change event is received from a Terminal Server session
- Each event also has support for **Before and After**

How a Service Works

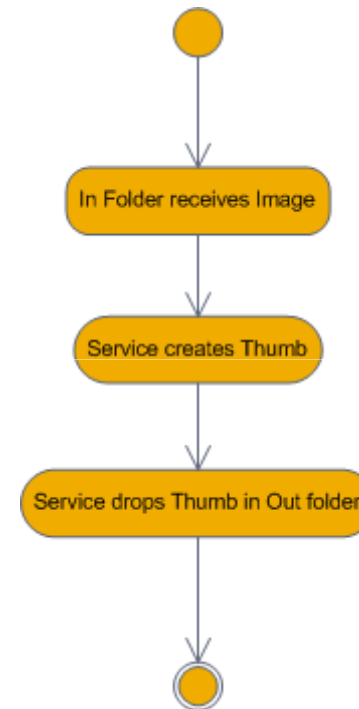
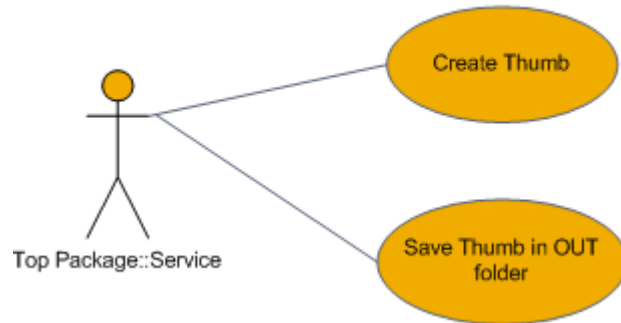
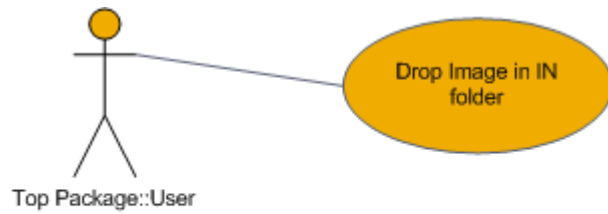
- Your goal, in the Service, should be to handle these events, and decide how your program will behave when they are received
- **OnStart** – initialize all the settings for the service, to allow new configurations to be picked up; start new instances of all the objects you'll need
- **OnStop** – clean things up, bring it down to a zero level – clear all the memory, release all the handles, dispose of all the objects
- Once started it will be up to your component to perform its required design function

What will it do?

- Use your favorite design methodology to determine what the function of the service will be
- Isolate that into a discreet component that can operate outside of a service environment
- Respect the principle of Separation Of Concerns
- Your service will be responsible only for Starting and Stopping your component
- Your component should do all the work

Thumb Maker Service Design

What will it do? What is its purpose?



Error Handling/Logging

- Since no UI, it is very important to handle errors and log exceptions
- No buttons for user to click
- System must make decisions when exceptions occur
- Logging errors is vital to fix bugs and correct changes to environment
- Log to file, system event, database, email alerts for critical errors, etc.

Error Handling/Logging

- What kind of logs do you need?
- Depends on complexity and needs of application
- Pick from
 - Error Log
 - Activity Log
 - Install Log
 - Notifications/Alerts

Unhandled Exceptions

- In services, to catch Unhandled Exceptions use CurrentDomain's UnhandledException event
- At start of service wire a listener to:
`AppDomain.CurrentDomain.UnhandledException += new UnhandledExceptionHandler(CurrentDomain_UnhandledException);`
- Once the event is caught:
 - Log exception
 - Stop the service gracefully
 - Call `Environment.Exit(1)` to prevent default CLR message box from popping up (bubbling)

Testing/Unit Testing

- Although it is possible to step through code in a service using Visual Studio, the process is complicated and requires installation of the service on your machine, starting it and attaching the service process to the Visual Studio Process Debugger
- The process is detailed by Microsoft in <http://msdn.microsoft.com/en-us/magazine/cc301845.aspx>
- However, you cannot debug the OnStart event nor the initialization, because you can only attach to running processes
- You may not always be able to fully replicate in your development environment, the production environment in which the service will run
- You may not want to install/uninstall applications on your development environment to avoid kludging/corrupting the registry

Testing/Unit Testing

- For those reasons we recommend keeping the design of the service simple
- Have the service contain only code that will start/stop and otherwise control the lifetime of the service
- Have all the operations that your service needs to perform in a separate DLL that can be tested/unit tested, with NUnit, VS Test, custom GUI, or your preferred testing method
- Also a good design for Separation Of Concerns

Testing/Unit Testing

- If you forgo the service testing by attaching to debugger it is vitally important to include lots of logging at various steps of the application – *remember the old days of Print/Printf?*
- Use System.Diagnostics.Trace/Debug liberally
- Remember – Debug symbols are only used for Debug builds, but Trace symbols are used for Release builds as well

Testing/Unit Testing

- If you really want to debug in Visual Studio anyway, these articles might help. They contain a detailed description of the problems and proposed solutions and workarounds:
 - <http://www.ondotnet.com/pub/a/dotnet/2003/09/02/debuggingsvcs.html>
 - <http://theimes.com/archive/2006/12/28/Debugging-Windows-Services-is-a-Pain.aspx>
 - <http://www.codeproject.com/KB/vb/ServiceDebug.aspx>

How will it be triggered?

- Since no UI, something needs to trigger the service to action
- Possibilities are (among others):
 - Timer
 - FileSystemWatcher
 - Custom Command
 - WCF/MSMQ/Remoting
 - WF/WWF
 - Email
 - Network Port

Timer

- Add a Timer object to your class and handle Tick events
- When a certain time is past or a certain time is reached, then perform the actions your service was designed to perform

FileSystemWatcher

- Add a FileSystemWatcher object to your class and handle events that you care about
- You can handle file creation, file modification, file deletion, etc.
- When the event is fired, perform the actions the service was designed to do

Custom Command

- Use a ServiceController object from a Desktop app or Console app to send custom commands to your service, via the ExecuteCommand() method
- It can accept commands in the form of integers between 128 and 255
- In the service override the OnCustomCommand() method to perform your custom actions

WCF/Remoting/MSMQ

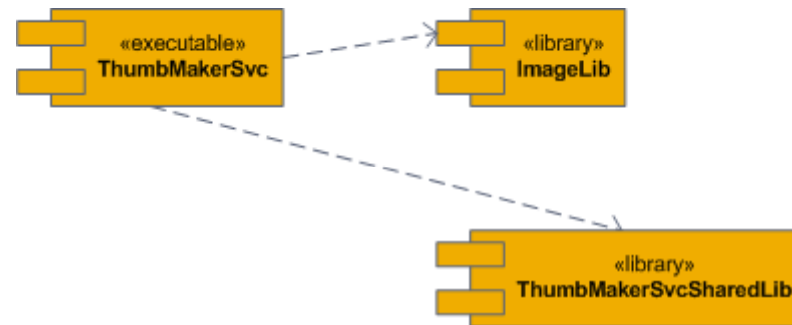
- Use WCF/Remoting/MSMQ messages to send information to the service
- Using these as triggers your service can then perform custom actions you designed it to do

- Use a Timer to periodically connect to an email box using Collaborative Data Objects or whatever your preferred method for accessing a Mailbox
- Or register for Exchange events
- Or use any of 13 other possible methods to get to an email box
- Then parse the emails as they arrive and take appropriate action, as per requirements

Network Port

- Create a custom Port listener using `System.Net.Sockets.TcpListener` to accept information from the network
- Create user interface to send information using `System.Net.Sockets.TcpClient` to your `TcpListener`
- You'll have to come up with your own "protocol" for sending commands and data to the service (i.e. defining the messages)
- Process the custom behavior as required

Thumb Maker Service Design



Our service will monitor the IN folder defined in our App.Config using a FileSystemWatcher. When it senses files of type image dropped in there it will create thumbnails and copy them to the OUT folder defined in our App.Config. The work will be performed by the ImageLib DLL and the ThumbMakerSvc will start and stop it. The library will send events whenever logging is to take place and the service will persist those requests to the file system as log files.



DEMO

Service Implementation

Monitoring a Service

- Since no UI besides SCM, if you want to “see” what is happening with the service a good solution is a Monitor Console
- Windows GUI app that attaches to the service via `System.ServiceProcess.ServiceController`
- Attaches to any file/database/etc. log files your service uses
- Lets your start/stop/send custom commands and view status of your service

Monitoring a Service

- ServiceController provides very limited access to the service
- If you need more Input/Output access, options are:
 - File System
 - Database
 - Shared Memory
 - WCF/Remoting/MSMQ
 - Email Alerts
 - System Events

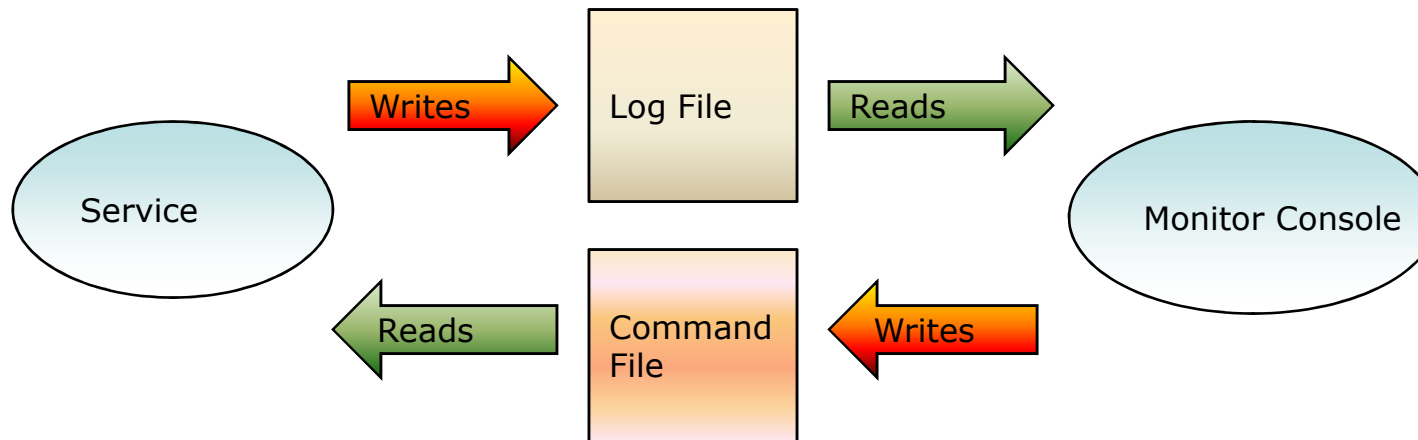
Service Controller

- The ServiceController has a limited selection for feedback
- Most involves status of the service, but nothing fancy like activity tracking
- It can send custom commands, but only in the form of Integers between 128 and 255
- You have to come up with your own convention
- If your commands need extra data, then you have to pick it up from file/database/etc.
- Not useful for all types of situations

File System

- Good way to track what service is doing
- Service writes progress to a log file (Action Log) and Monitor Console reads the file using a `FileSystemWatcher`
- To send commands you can write to a special command file through the Monitor Console and have the service monitor that file, read the commands and data, then clear it

File System



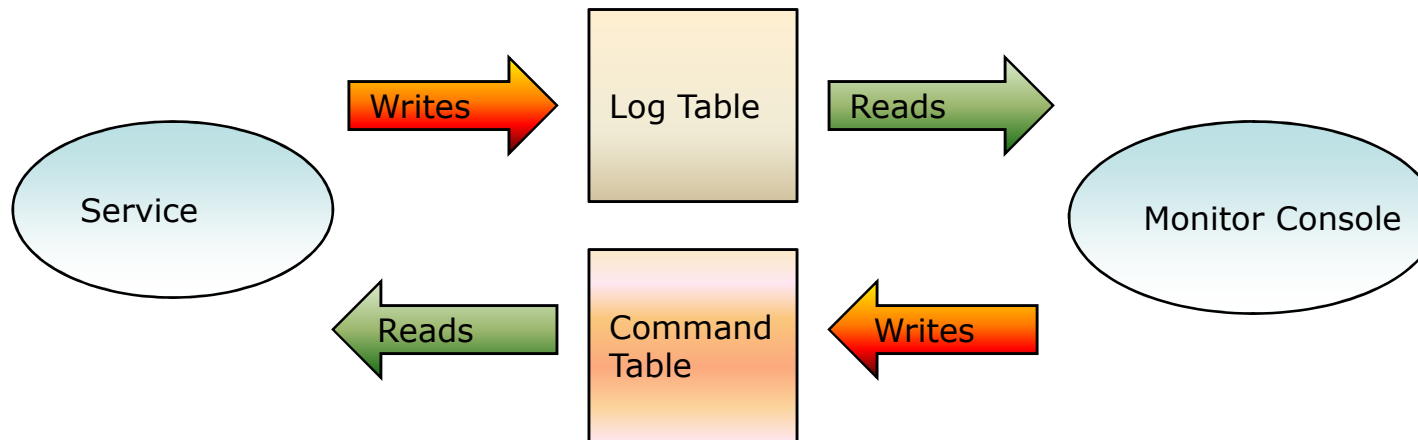
Pros/Cons:

- File System is persistent. Good solution for keeping logs
- FileSystemWatcher is fast for receiving feedback while reading logs
- Keeps clutter out of system events
- FileSystemWatcher is not 100% reliable – not a good solution for sending commands
- Also, commands would have to be parsed and agreed upon on both Service and Monitor Console – may be more work than it's worth

Database

- Reasonable way to track what service is doing
- Service writes progress to a log table and Monitor Console reads it using a Timer to fetch updates
- To send commands you can write to a special command table through the Monitor Console and have the service monitor that table with a timer, read the commands and data, then clear it
- It's quite a slow response and onerous to write

Database



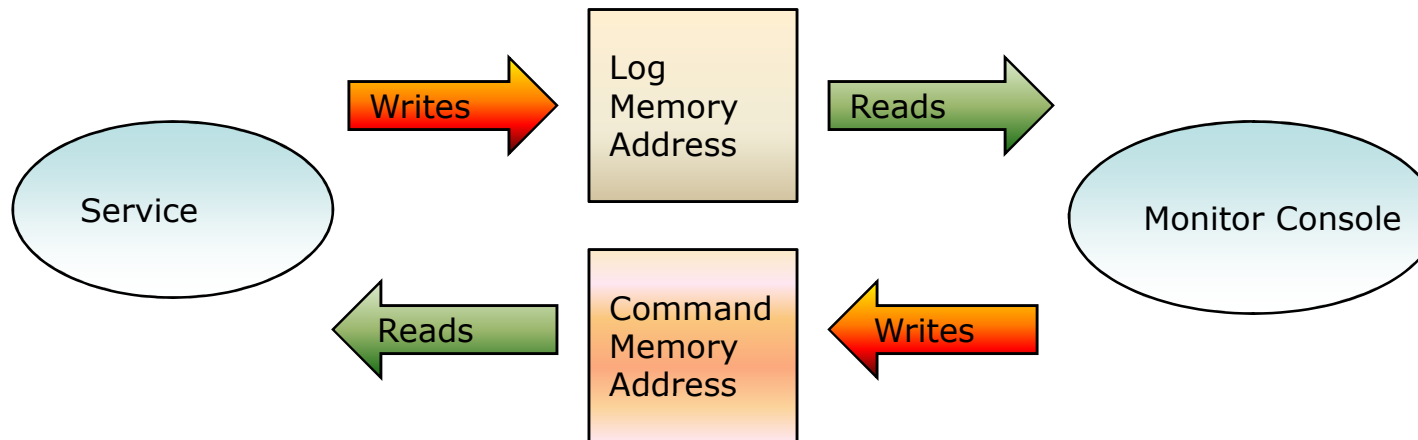
Pros/Cons:

- Non-volatile – decent solution for keeping logs
- You may consider this clutter in the database
- Logs easy to access across machines
- Feedback is slow as it depends on a timer
- Commands are slow as it depends on a timer, plus need to make up a convention of your own – not a good solution for sending commands

Shared Memory

- Use an area of Shared Memory that both the Service and Monitor Console can write/read to
- It is onerous to maintain as you have to create conventions for sending commands and if you need to send large amounts of data you could run out of memory

Shared Memory

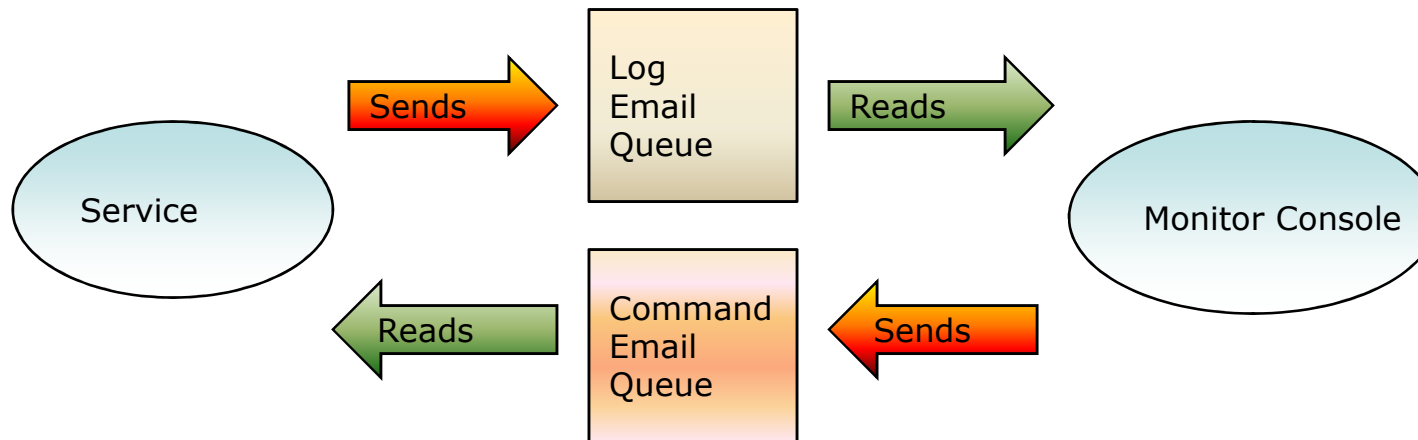


Pros/Cons:

- Memory is volatile – not a good solution for keeping logs
- Commands are probably not required to be persistent – reasonably good solution for sending commands
- Only works on the same machine the service is in – cannot access remotely

- Create a mailbox for the logging and have the Monitor Console read from that mailbox
- Create a mailbox for the commands and have the Service read from that mailbox
- The process is slow, as email systems are not typically fast
- Also requires convention for commands to be setup, which can be onerous

Email



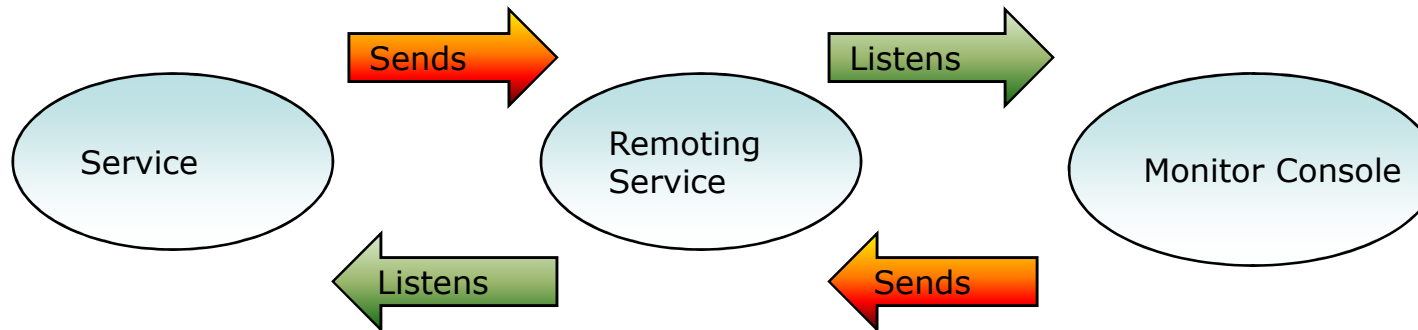
Pros/Cons:

- Not the best way to store logs – email is slow and storage inefficient, hard to search
- Good for alerts to more important errors or events as it stands out from day-to-day logs
- Not a good way to send commands. Commands would have to be parsed, not type-safe, slow...
- Depends on email infrastructure, which is not always 100% reliable

Remoting

- Prior to .Net 3.0 this was the best option
- Create a Remoting Server (extra service) and an interface library to allow the Service and Monitor Console to communicate using a Singleton pattern
- Service and Monitor Console attach to the Remoting Server as clients and send commands/data back and forth

Remoting



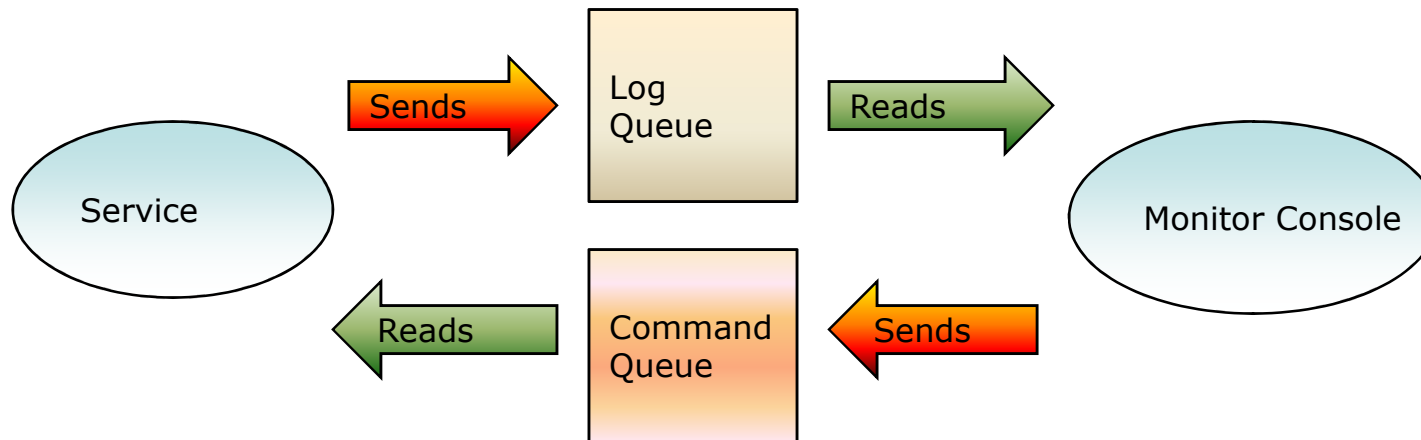
Pros/Cons:

- Remoting is a network protocol, so traffic will be chatty if you want to keep vast logs
- Logs are not persistent, unless you are also writing to a file or database
- OK for sending brief status info
- Great for sending commands, using type safe commands and data
- Requires another service to be the mediator, or server for the Remoting calls - the Service cannot be at the same time both Client and Server to the Remoting calls

MSMQ

- In some ways similar to Email system
- Create a Log Queue for sending the log data. Service sends logs to this queue. Monitor would listen to events and display results
- Create a Command Queue for sending custom commands from the Monitor Console. Service would listen to events and act on commands
- Similar problems to email system, but can work if the monitor and service are separated in different networks and you need a reliable transport mechanism

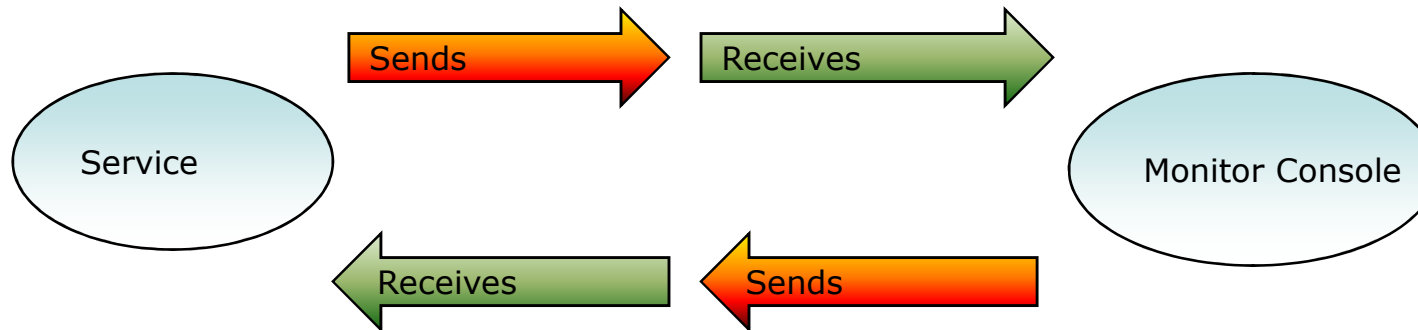
MSMQ



Pros/Cons:

- Probably not the best way to store logs – hard to view, slow
- Probably a decent way to send commands
- Requires Queues to be setup for logging and sending commands
- Depends on a Queuing infrastructure

- Since .Net 3.0 the perfect solution came along for communication with the Service
- WCF allows a duplex pattern whereby both the Service and the Monitor Console can establish their own built-in remoting servers, doing away with the need for a middleman



Pros/Cons:

- Memory is volatile – not a good solution for keeping logs, unless also storing in file system or database
- Commands are fast and responsive
- WCF allows duplex pattern, so Service and Monitor Console can both host and register as listeners to each others' calls
- Commands and data are type safe
- Probably the best way to send commands and receive custom feedback, if your service requires that level of functionality

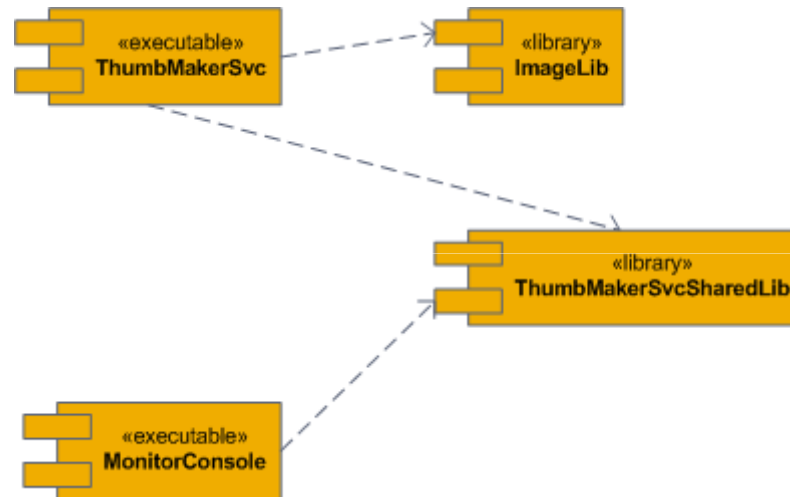
System Events

- Great for install logs
- OK for error logs
- Probably too messy for activity logs
- You'll need admin access to view the Event Logs
- Your application will require higher security privileges if writing to Event Log
- If you use a monitor console it's harder to view the logs than with a file based log
- Of course, can't send commands with Event Log

Best Solution

- Depends on your applications' needs
- For most applications you probably want a combination:
 - File system or database for storing logs
 - Email alerts for important events or errors
 - ServiceController for sending custom commands
 - WCF for exotic needs

Thumb Maker Service Design



- Our service will persist logs to file system
- The Monitor Console will read the logs to display activity
- The Monitor Console will send custom commands via Service Controller



DEMO

Monitor Console Implementation



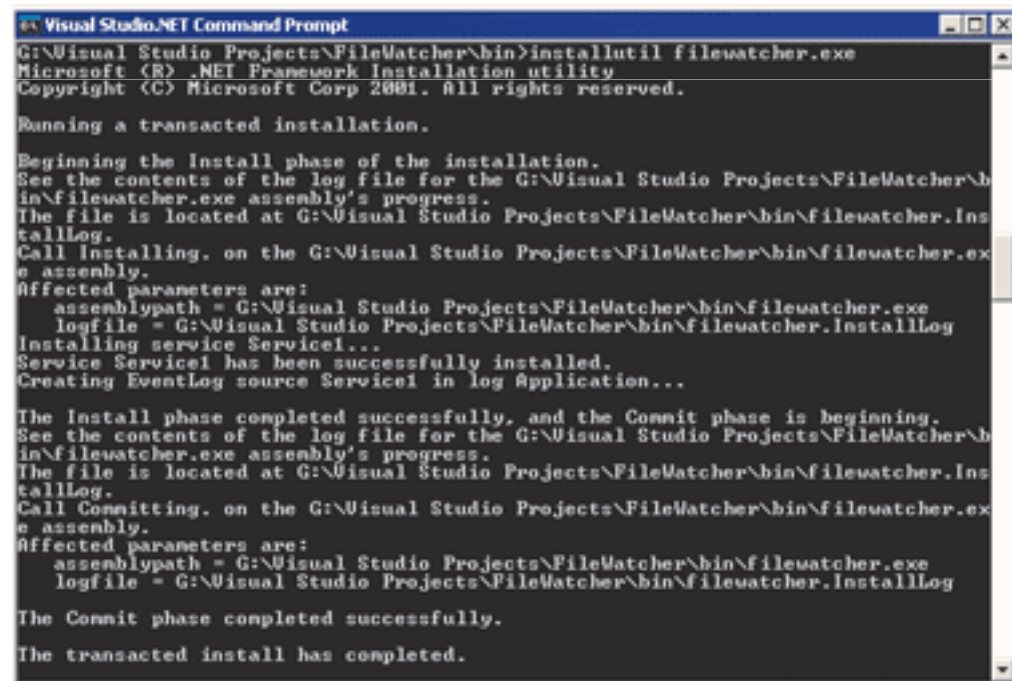
Quick Survey

- How many have written installers?
- How many have used VS version of InstallShield?

- There are two main ways to deploy the Service
 - InstallUtil
 - OK for simple services
 - Must do everything manually
 - No extra work to write
 - Custom Installer
 - Needed for more complex installation scenarios
 - Extra work to write it
 - Ease of use during installation and uninstallation
 - More options for fine-tuning installation process

InstallUtil

- Command Line utility
- Found in .../Microsoft.Net/<ver>/InstallUtil.exe
- /i for install, /u for uninstall

A screenshot of a Visual Studio .NET Command Prompt window. The window title is "Visual Studio .NET Command Prompt". The command prompt shows the execution of the command "G:\Visual Studio Projects\FileWatcher\bin>installutil filewatcher.exe". The output text is as follows:

```
G:\Visual Studio Projects\FileWatcher\bin>installutil filewatcher.exe
Microsoft (R) .NET Framework Installation utility
Copyright (C) Microsoft Corp 2001. All rights reserved.

Running a transacted installation.

Beginning the Install phase of the installation.
See the contents of the log file for the G:\Visual Studio Projects\FileWatcher\bin\filewatcher.exe assembly's progress.
The file is located at G:\Visual Studio Projects\FileWatcher\bin\filewatcher.InstallLog.
Call Installing. on the G:\Visual Studio Projects\FileWatcher\bin\filewatcher.exe assembly.
Affected parameters are:
  assemblypath = G:\Visual Studio Projects\FileWatcher\bin\filewatcher.exe
  logfile = G:\Visual Studio Projects\FileWatcher\bin\filewatcher.InstallLog
Installing service Service1...
Service Service1 has been successfully installed.
Creating EventLog source Service1 in log Application...

The Install phase completed successfully, and the Commit phase is beginning.
See the contents of the log file for the G:\Visual Studio Projects\FileWatcher\bin\filewatcher.exe assembly's progress.
The file is located at G:\Visual Studio Projects\FileWatcher\bin\filewatcher.InstallLog.
Call Committing. on the G:\Visual Studio Projects\FileWatcher\bin\filewatcher.exe assembly.
Affected parameters are:
  assemblypath = G:\Visual Studio Projects\FileWatcher\bin\filewatcher.exe
  logfile = G:\Visual Studio Projects\FileWatcher\bin\filewatcher.InstallLog

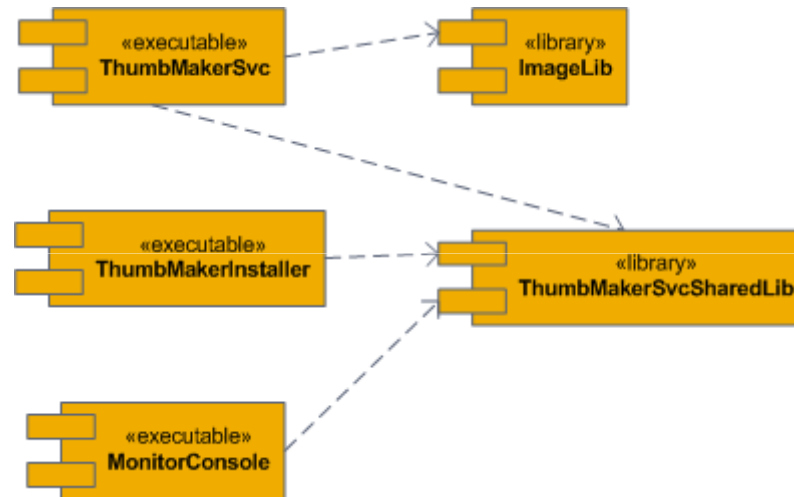
The Commit phase completed successfully.

The transacted install has completed.
```

Custom Installer

- Allows customization of installation procedure to unlimited fine-grained level
- Builds as an MSI application
- GUI or unattended installation
- Easy to collect information for writing config files, choosing install location, etc.
- Can select various options such as automatic start-up on reboot, user under which services will run, etc.
- Easy to uninstall through Add/Remove Programs

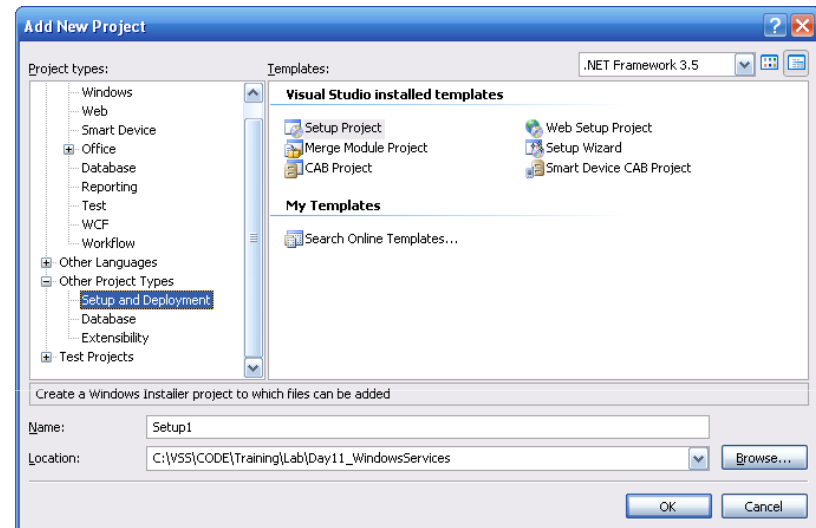
Thumb Maker Service Design



Our Thumb Maker Service will have a custom installer

Custom Installer

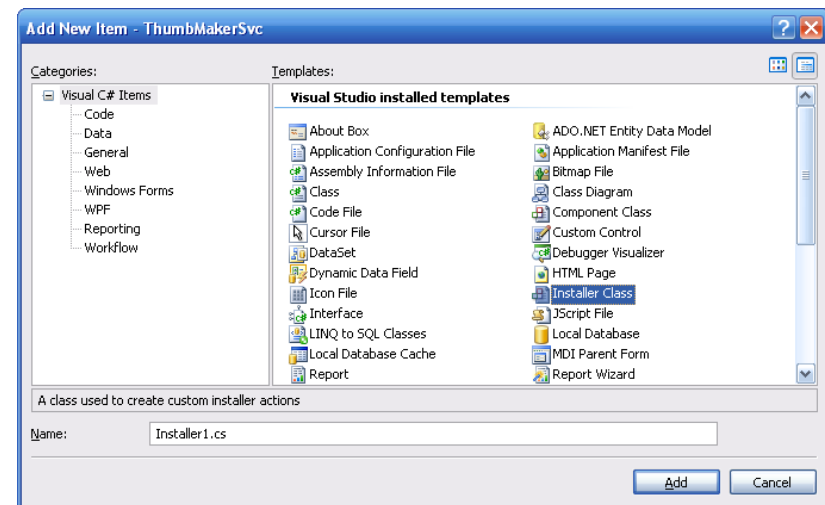
- Support through Visual Studio via Setup & Deployment Projects
- Can add to any Solution, or create stand-alone
- Easy to setup
- Easy to configure custom installation events



Creating a Setup Project

Custom Installer

- The Setup and Deployment project only gives you basic install functionality
- Where it gets interesting is the customization of the installation process
- To do that you need to create a new Installer class in your service
- You can then override a few methods: Install, Commit, Uninstall and Rollback



Custom Installer

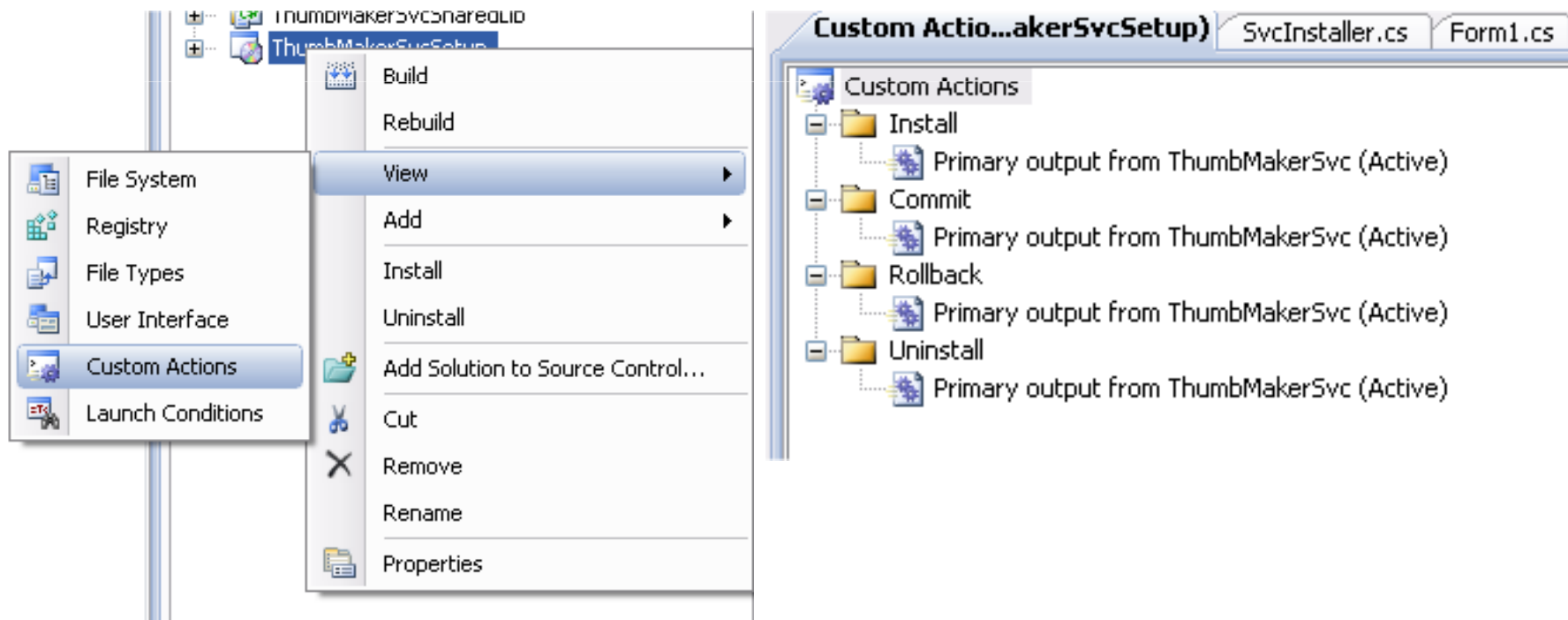
- **INSTALL:** this method is called during the installation process – you can use it to create configuration files, interject service settings, like prompt user for account that service will run on, etc.
- **COMMIT:** this is called when the installation completes – you can use it to start the service automatically when the MSI closes

Custom Installer

- **UNINSTALL:** if you want to uninstall your service, this method will be called – use this to clean up any config files or any settings that you created during the **INSTALL** phase.
- **ROLLBACK:** The counterpart to **COMMIT**. This method is used to restore the state if the installation fails – use this to clean up after a failed installation, if needed.

Custom Installer

- To get your custom code to be called, wire up the Setup project's Custom Actions to your Service's Installer class



Tip

- Your custom installer can also uninstall your service
- If you are going to uninstall your service DO NOT have the SCM open
- Your service will NOT uninstall
- It will be marked for deletion and you'll have to reboot to complete uninstallation



DEMO

Creating a Custom Installer

- Where can I get the slides?
 - <http://www.pedco-usa.com/Training/Default.aspx>
- Where can I get the code?
 - <http://www.pedco-usa.com/Training/Default.aspx>
- Audience Questions