# 1. Для чего предназначен WindowsServiceToolkit?

WindowsServiceToolkit – .NET библиотека, предназначенная для упрощения создания, отладки и развертывания служб Windows. Вот проблемы при разработке, которые решает эта библиотека:

1. На настоящий момент, нет никаких средств для отладки служб ввиду того, что их невозможно запускать как обычные приложения. Эта библиотека позволяет создавать службы, которые могут запускаться в консольном режиме, если это требуется. Например, при отладке службы в Visual Studio или при запуске службы из обозревателя.
2. Не весь функционал по установке служб из ОС поддерживается в .NET: невозможно указать действия при отказе в работе службы (перезапуск, перезагрузка, выполнение команды), автоматически запустить её сразу после установки, установить службе отложенный автозапуск. Весь этот функционал добавлен в WindowsServiceToolkit.
3. Setup Project имеет очень ограниченный функционал по созданию диалоговых окон для инсталляторов, а показывать диалоговые окна из Installer классов стандартными средствами нельзя. WindowsServiceToolkit решает эту проблему, позволяя отображать любые диалоговые окна из Installer классов (см. UIThreadAction).
4. WindowsServiceToolkit содержит классы, решающие частые задачи при установке приложений:
   * Для создания и удаления правил межсетевого экрана для Windows старше XP (см. WindowsFirewall).
   * Администрирование учетных записей пользователей (см. SAMUserAccounts).
   * Журналирование (см. UniversalLog).

# 2. Простая служба на WindowsServiceToolkit

1. Создайте проект консольного приложения.
2. Подключите WindowsServiceToolkit к своему проекту.
3. Создайте класс вашей службы. Он может быть таким:

using Granch.WindowsServiceToolkit;

// WindowsServiceBase – абстрактный класс для создания служб.

// Все методы и свойства создаются автоматически с помощью Refactor -> Implement Abstract Class.

public class MyService : WindowsServiceBase

{

// Пока m\_run = true, служба работает.

volatile bool m\_run;

// Системное имя службы. Не должно содержать пробелов.

public override string ServiceName

{

get { return "MyService"; }

}

// Отображаемое имя службы. Ограничений на набор символов нет.

public override string DisplayedName

{

get { return "My Service"; }

}

// Описание.

public override string Description

{

get { return "Описание вашей службы."; }

}

// Список служб, без которых эта служба работать не может.

// В списке нужно указывать системные имена служб.

// ОС автоматически запустит все службы из этого списка

// перед запуском вашей.

public override string[] ServicesDependedOn

{

get { return null; }

}

// Метод, выполняющий всю работу службы.

// Должен иметь возможность прекратить свою работу по вызову RequestStop.

protected override void RunService()

{

m\_run = true;

while (m\_run)

{

Console.WriteLine("Working...");

Thread.Sleep(1000);

}

}

// Запрос остановки. Должен прекращать работу метода RunService.

protected override void RequestStop()

{

m\_run = false;

}

}

1. Добавьте этот код в Main:

MyService service = new MyService();

service.Run();

1. Готово! Для того, чтобы прервать работу службы в консольном режиме, нажмите CTRL+C (Не работает при отладке приложения в Visual Studio 2010, issue page <http://connect.microsoft.com/VisualStudio/feedback/details/524889/debugging-c-console-application-that-handles-console-cancelkeypress-is-broken-in-net-4-0>).

# 3. Описание классов библиотеки

### 3.1 WindowsServiceBase

WindowsServiceBase – базовый абстрактный класс для служб Windows. При наследовании от него необходимо реализовать несколько абстрактных методов и свойств:

Свойство **ServiceName** – системное имя службы, не должно содержать пробелов и нелатинских символов.

Свойство **DisplayedName** – отображаемое имя службы, ограничений на набор символов нет.

Свойство **Description** – описание службы.

Свойство **ServicesDependedOn** - Список служб, без которых данная служба работать не может. В списке нужно указывать их системные имена. ОС автоматически запустит все службы из списка, даже если в их настройках указан запуск вручную. При добавлении службы в список зависимостей нужно указывать её системное имя. Если вы не знаете системного имени службы, его можно посмотреть в консоли services.msc (Выполнить -> services.msc): найдите нужную службу в списке, вызовите её свойства и скопируйте значение поля «Имя службы».

Метод **RunService()** – метод, в котором выполняется служба.

Метод **RequestStop()** - метод, останавливающий службу. После выполнения этого метода, работа **RunService()** должна завершиться.

Служба запускается методом **Run()**. Режим работы (консольный или не интерактивный) определяется автоматически на основе родительского процесса. Если родительским процессом является services.exe (процесс ОС, работающий со службами), то служба запускается в не интерактивном режиме. Иначе, служба запускается в консольном режиме.

В режиме консольного приложения метод RunService() запускается в главном потоке. Метод StopHandler() в таком режиме вызывается при нажатии Ctrl+C.

В неинтерактивном режиме метод RunService() выполняется в отдельном потоке, а метод StopHandler() выполняется при остановке службы системой.

Для записи событий сервиса используйте Log.Write(). Log – экземпляр класса **UniversalLogger**. Изначально он настроен на вывод сообщений в консоль, Debug output и системный журнал событий (за имя источника берется значение поля ServiceName).

### 3.2 WindowsServiceInstaller

WindowsServiceInstaller — класс, представляющий собой универсальный инсталлятор для служб. В качестве шаблонного параметра он принимает тип, унаследованный от WindowsServiceBase. Этот шаблонный параметр определяет тип устанавливаемой службы.

Следующие свойства позволяют менять настройки установки службы:

**StartType** — Тип запуска службы. Может принимать одно из следующих значений:

* Disabled — служба отключена.
* Automatic — служба стартует при загрузке Windows.
* Manual – службу можно запустить только вручную.
* Delayed — служба стартует при загрузке Windows, но с небольшой задержкой. Полезно, когда служба запускается быстрее компонентов, от которых она зависит.

**FailureActions** — Набор действий при отказе в работе службы. При первом отказе выполняется первое действие из списка, при втором — второе и т. д. Действием при отказе может быть перезагрузка, перезапуск службы или запуск программы.

**Account** – Тип учетной записи, используемый службой. Может принимать одно из следующих значений:

* LocalService – соответствует стандартной локальной учетной записи с ограниченными правами.
* LocalSystem – соответствует стандартной локальной учетной записи с повышенными правами.
* NetworkSystem – соответствует стандартной сетевой учетной записи с повышенными правами.
* User – соответствует учетной записи пользователя. При использовании этого типа учетной записи небходимо указывать имя пользователя и пароль(Свойства Username и Password).

**Username** – имя пользователя учетной записи, которую будет использовать служба. Если значение Account не равно User, то Username игнорируется. Если пользователь размещен не в локальном хранилище (SAM), необходимо указывать префикс его размещения (например, GRANCH\\sharov). Создание новых локальных пользователей упрощает класс **SAMUserAccounts**. Для того, чтобы автоматически добавить префикс к имени пользователя, используйте метод **SAMUserAccounts.AppendMachineNamePrefix**.

**Password** — пароль указанной учетной записи.

**FailCountResetTimeHours** - Через сколько часов с момента последнего отказа счетчик отказов будет сброшен до нуля.

**FailureRebootMessage** - Сообщение при перезагрузке системы по причине отказа службы (FailureActionType.Reboot).

**FailureRunCommand** — Системная команда, запускаемая при отказе службы (FailureActionType.RunCommand).

**StartOnInstall** — Запуск службы сразу после завершения установки.

**StartTimeoutSeconds** — Максимальное время ожидания запуска службы в секундах. Если служба не запустилась за это время, она будет считаться отказавшей.

### 3.3. WindowsFirewall

WindowsFirewall – класс для работы с межсетевым экраном Windows версий старше XP. Его функционал — удаление и добавление правил доступа. Правила доступа представлены классом WindowsFirewallRule.

Методы:

**AddRules** — Добавление набора правил межсетевого экрана. Свойство **WindowsFirewallRule.Name** каждого правила должно быть уникально.

**RemoveRules** – Удаление набора правил межсетевого экрана. Поиск правила для удаления происходит по свойству **WindowsFirewallRule.Name**.

### 3.4. WindowsFirewallRule

WindowsFirewallRule — Класс, представляющий собой правило для межсетевого экрана.

Свойства:  
**Name** — Название правила. Может содержать любые символы, но обязательно должно быть уникальным. Если при попытке добавления правила его название совпадет с другим, уже существующим в системе, то такое правило добавлено не будет.

**Action** — Тип действия (Разрешить/запретить).

**Description** – Описание правила. Ограничений на длину и набор символов нет.

**Protocol** — Тип сетевого протокола, к которому применяется это правило (TCP/UDP).

**Direction** — направление соединения, которое обрабатывает правило (Входящее/Исходящее).

**Ports** — Список портов, которые обрабатывает правило. Диапазон портов можно указать так:

Ports = Enumerable.Range(6000, 10); //Порты 6000-6009

**Program** — Путь к приложению, к которому применяется правило. Если это свойство оставлено пустым, правило будет применяться ко всем приложениям. Это свойство нельзя использовать одновременно со свойством Service.

**Service** – Название службы, к которой применяется правило. Если это свойство оставлено пустым, правило будет применяться ко всем приложениям. Это свойство нельзя использовать одновременно со свойством Program.

### 3.5. SAMUserAccounts

Класс для работы с учетными записями на локальном компьютере. Позволяет проверять существование учетной записи, создать новую учетную запись или удалить её.

Метод **Exists** — Определяет, существует ли учетная запись с указанным именем пользователя.

Метод **Create** – Создает новую учетную запись с указанным именем пользователя и паролем.

Метод **Remove** — Удаляет учетную запись с указанным именем пользователя.

Метод **MakeAdmin** – Дает пользователю права администратора.

Метод **AppendMachineNamePrefix** – Добавляет к имени пользователя префикс имени компьютера (если префикса еще нет).

### 3.6. SystemCommand

Класс, упрощающий запуск системных команд в невидимом режиме, т.е. без отображения консоли и окон.

Свойства:

**ExecutablePath** — путь к исполнительному файлу.

**Parameters** — параметры, отправляемые исполнительному файлу.

Методы:

**Execute** — Запуск команды и ожидание её выполнения. Метод возвращает код завершения приложения (В случае успешного выполнения большинство приложений возвращает 0).

### 3.7. UIThreadAction

Класс, позволяющий отображать диалоговые окна из Installer класса. Он полезен потому, что при обычных условиях Installer работает в потоке с MTA атрибутом и не позволяет создавать окна и другие объекты, которым требуется поток с STA атрибутом.

Этот класс принимает функциональный объект в качестве параметра конструктора. При вызове метода **Invoke**(), функциональный объект запускается в отдельном потоке с STA атрибутом, и происходит ожидание завершения этого потока. По завершению метод **Invoke**() возвращает результат выполнения функтора.

### 3.8. UniversalLogger

Класс, упрощающий журналирование событий в работе приложения. Позволяет писать одновременно в несколько журналов (файл, консоль, debug output, системный журнал событий). По умолчанию происходит запись только на консоль и debug output.

Свойства:

**RedirectToConsole** - Отображать сообщения на консоли. Изначально равен true.

**RedirectToEventLog** - Выводить сообщения в системном журнале событий. Консоль просмотра системных событий можно вызвать при помощи Run → eventvwr.msc. Сообщения от вашей службы должны добавляться в журнал «Приложение», расположенный в папке «Журналы Windows».

**EventSource** - Имя источника для системного журнала событий.

**RedirectToFile** - Записывать сообщения в файл.

**PathToFile** - Путь к файлу.

**RedirectToDebug** - Записывать сообщения в Debug output. Изначально равен true.

Методы:

**Write** – запись сообщения. Можно передавать не только строку, но и исключение. В таком случае будет записано описание исключения и stack trace.

# 4. Описание примеров

### 4.1. Основной функционал (Basic и BasicSetup)

В этом примере реализована служба с зависимостями от других служб, инсталлятор, настраивающий действия при отказе и проект инсталляционного пакета.

В проекте Basic:

BasicServiceExample.cs – реализация службы. В качестве примера прописана зависимость от службы «Веб-клиент». Зависимости автоматически запускаются перед запуском службы, как в консольном, так и в не интерактивном режимах.

InstallerClass — инсталляционный класс для службы BasicServiceExample. Содержит пример создания действий при отказе службы.

Проект BasicSetup – инсталляционный пакет для службы, созданной в проекте Basic.

### 4.2. Расширенный функционал (Advanced и AdvancedSetup)

В этом примере показано как отображать диалоговое окно из Installer класса, создавать правила для межсетевого экрана и как создавать пользователей во время работы инсталлятора.

В проекте Advanced:

AdvancedServiceExample.cs – минимальная реализация службы.

InstallerClass.cs — инсталляционный класс для службы AdvancedServiceExample. Выводит диалоговое окно, настраивает межсетевой экран и создает пользователя для службы.

Проект AdvancedSetup - инсталляционный пакет для службы, созданной в проекте Advanced. Использует Custom Actions для вызова инсталляционного класса InstallerClass.

# 5. Полезная информация

### 5.1. Отладка инсталляторов

Для отладки инсталляционного класса используйте вызов **Debugger.Launch()**. Если к процессу не прикреплен отладчик и был вызван Debugger.Launch() - отображается стандартный запрос на запуск отладчика. Так как инсталляция запускается с повышенными привилегиями, то и экземпляр Visual Studio, при помощи которого вы собираетесь отлаживать, должен быть запущен от администратора.

У Debugger.Launch() есть нежелательный побочный эффект — если пользователь отказался от запуска отладчика, инсталлятор впоследствии может вести себя некорректно (в моем случае переставали вызываться обработчики на события BeforeInstall, AfterInstall и т.д.). Поэтому всегда подтверждайте вызов отладчика и не вызывайте его без надобности. В Release сборках обязательно избавляйтесь от вызовов Debugger.Launch() (через #if DEBUG, например).

### 5.2. Последовательность событий при инсталляции

Эта информация часто вылетает из головы, так что стоит добавить её сюда.

1. BeforeInstall
2. AfterInstall
3. Committing
4. Committed

### 5.3. Управление службами при помощи средств ОС

Для управления службами используется консоль services.msc (Run → services.msc). В нем можно останавливать, запускать и настраивать службы. У неё есть единственный недостаток - отсутствует возможность удаления служб. Для удаления служб можно использовать команду «sc delete Имя службы» (требуются повышенные привилегии).