Урок 1

1. Введение в SOA

Технология Windows Communication Foundation (WCF) позволяет создавать распределенные системы и реализует новую технологию программирования – SOA (Service Oriented Architecture). Буквальный перевод этого термина – «архитектура, ориентированная на службы». Основой таких распределенных систем являются службы (вместо термина «служба» можно использовать синоним «сервис»). Особо следует отметить, что такая архитектура допускает взаимодействие приложений, написанных и выполняемых на разных платформах.

Другими словами, SOA-приложение представляет собой набор служб. Службы эти могут быть локальными или удаленными. Они могут быть разработаны разными

разработчиками, на разных языках, в разных операционных системах и выполняться они могут на разных платформах.

Можно привести много показательных и поучительных примеров SOA-приложений. Остановимся на нескольких.

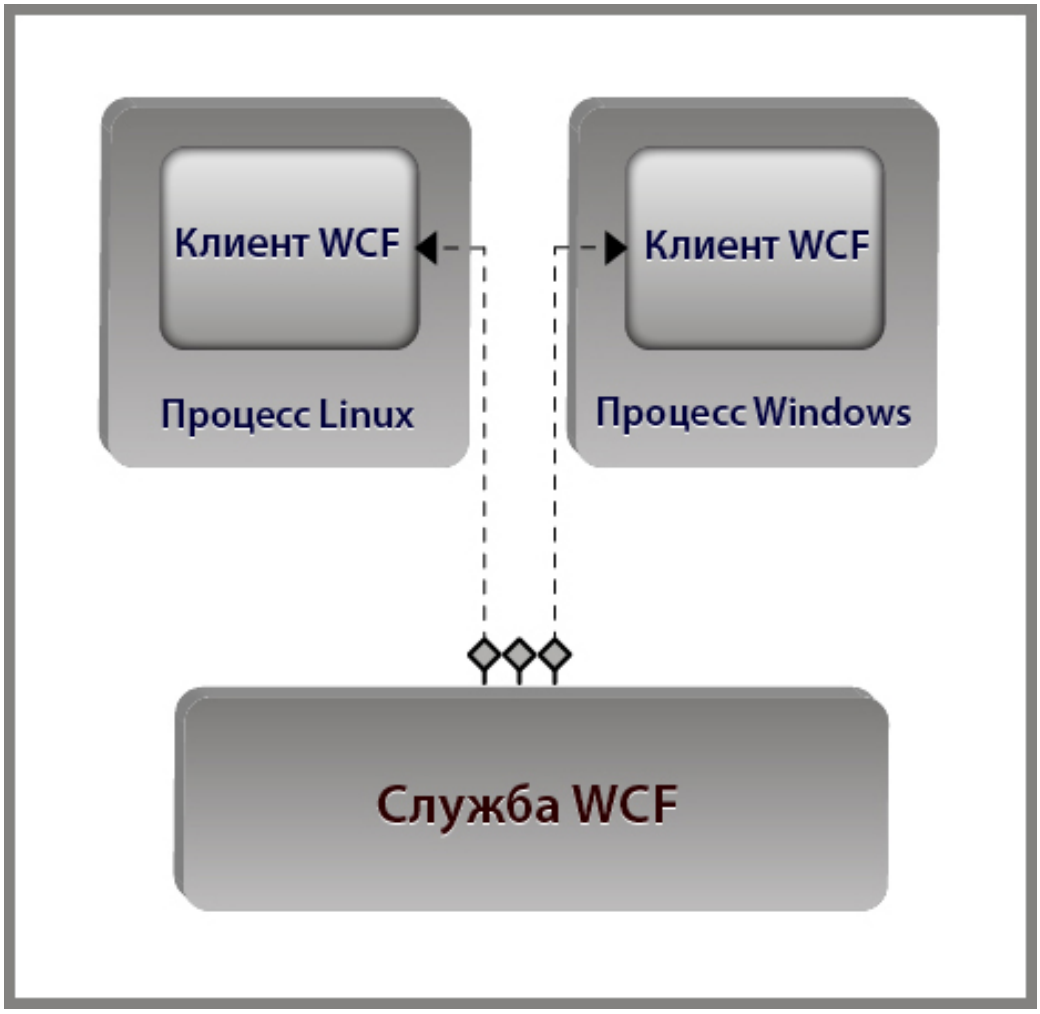
Австралийский Commonwealth Bank of Australia работает на системе, разработанной целиком в .NET. Клиентские части приложения представляют собой WinForms приложения, общающиеся с .NET Web службами. Сразу при запуске эта система обеспечивала одновременную работу 30000 клиентов на 1700 сайтах по всей Австралии. Сейчас эти показатели значительно выше.
Одна из крупнейших в США компаний по прокату автомобилей Dollar Thrifty Automotive Group 66% своих сделок совершает удаленно. Применение SOA технологии в разработке новой версии программного обеспечения позволило компании повысить качество и надежность этих сделок и экономить при этом ежегодно $135000, по сравнению с использованием предыдущего программного обеспечения.

Аэропорт Цюриха, обслуживающий ежегодно более 20 миллионов туристов, насчитывает в своем штате 24000 сотрудников, задействованных в 180 видах активности. Здесь WCF был задействован для реализации визуальной системы контроля и мониторинга буквально всего, что происходит в любой части аэропорта. Эта интерактивная система способна в реальном времени выдавать информацию по таким направлениям, как прогнозы пассажиропотоков, местонахождение самолетов, даты и время рейсов, погодные условия, количество багажа и много других данных. Можно сказать еще о Лондонской бирже, крупнейшей в Европе, которая тоже применяет в своей деятельности SOA-приложения.

Итак, службы SOA-приложения могут представлять собой очень разные по природе приложения. Потребителями этих служб будут выступать клиенты. Клиент может быть чем угодно – WPF-приложением, приложением WinForms, страницей ASP.NET или другой службой. При этом, SOA-приложение будет представлять собой логически связанное одно целое! Как же такая разнородная «компания» может работать согласованно?

Это является возможным потому, что клиенты и службы взаимодействуют, обмениваясь сообщениями. Это, как правило, независимые от транспортных протоколов SOAP-сообщения. Сообщения могут пересылаться между клиентом и сервером (но не напрямую, как мы скоро увидим). Также сообщения могут пересылаться через какого-либо посредника, например, через Microsoft Message Queue (MSMQ). Способы пересылки сообщений мы рассмотрим дальше. Особо при этом

надо отметить тот факт, что реализация как службы, так и клиента не зависит от того, как они будут использоваться в приложении – как локальные или удаленные.



Однако, вы всегда должны помнить о том, что данные, пересылаемые между клиентом и службой должны быть сериализуемыми. Дело в том, что когда клиент обращается к службе, его запрос прежде всего сериализуется. Затем, этот запрос передается по цепочке каналов. Каналы – это такие абстракции, которые отвечают за выполнение специфических обработок запроса. Каналы делятся на две категории: транспортные, отвечающие за пересылку сообщений, и протокольные, отвечающие за преобразование сообщений в соответствии с необходимыми требованиями. Точный набор и структура каналов (а значит и преобразования, выполняемые над запросом) зависят от привязок. Например, именно протокольные каналы выполняют такие действия, как обеспечение безопасности сообщения, его перекодировку, поддержку транзакций и так далее. Пройдя через цепочку протокольных каналов, сообщение передается транспортному каналу, который и доставляет его до службы. На стороне службы происходит обратный процесс. Сообщение проходит в обратном порядке набор соответствующих протокольных каналов, затем десериализуется и попадает к

службе в первозданном виде. После того, как служба обработает запрос, этот запрос снова на пути к клиенту пройдет, сериализацию, цепочку каналов и т.д.

Как вы скоро узнаете, создавать каналы можно явно, с помощью класса ChannelFactory<T> либо неявно – с помощью прокси-класса.

WCF – это самый простой способ создавать и использовать службы на платформе Microsoft. С точки зрения реализации WCF представляет собой набор классов, надстроенных над .NET Framework's Common Language Runtime (CLR). Большинство из этих классов определены в пространстве имен System.ServiceModel. WCF позволяет создавать и службы, и клиентские приложения. Клиентские приложения отправляют запросы службам, выполняющим роли серверных приложений. Зная характеристики служб, легко согласиться с тем, что службы очень хорошо подходят для выполнения серверных ролей. Они находятся в активном состоянии, особо никому не мешая, и ожидают поступления сообщений от клиентов. Когда такие сообщения поступают, и поступают в правильном формате, служба имеет возможность прочитать в сообщении, чего от нее хочет клиент. В сообщении, как правило, указано имя метода службы, который клиент просит выполнить, а также могут находиться параметры, переданные клиентом для запускаемого метода. Если все сделано правильно, служба выполнит указанный метод и отправит клиенту сообщение с результатом. Давайте рассмотрим, как использовать технологию WCF для реализации SOA приложений.

2. Архитектура WCF

a. Сервисы или службы

В технологии программирования WCF, о которой мы начинаем разговор, важнейшим понятием является служба. Поэтому, прежде чем начинать разговор непосредственно о WCF, следует сказать несколько слов о службах.

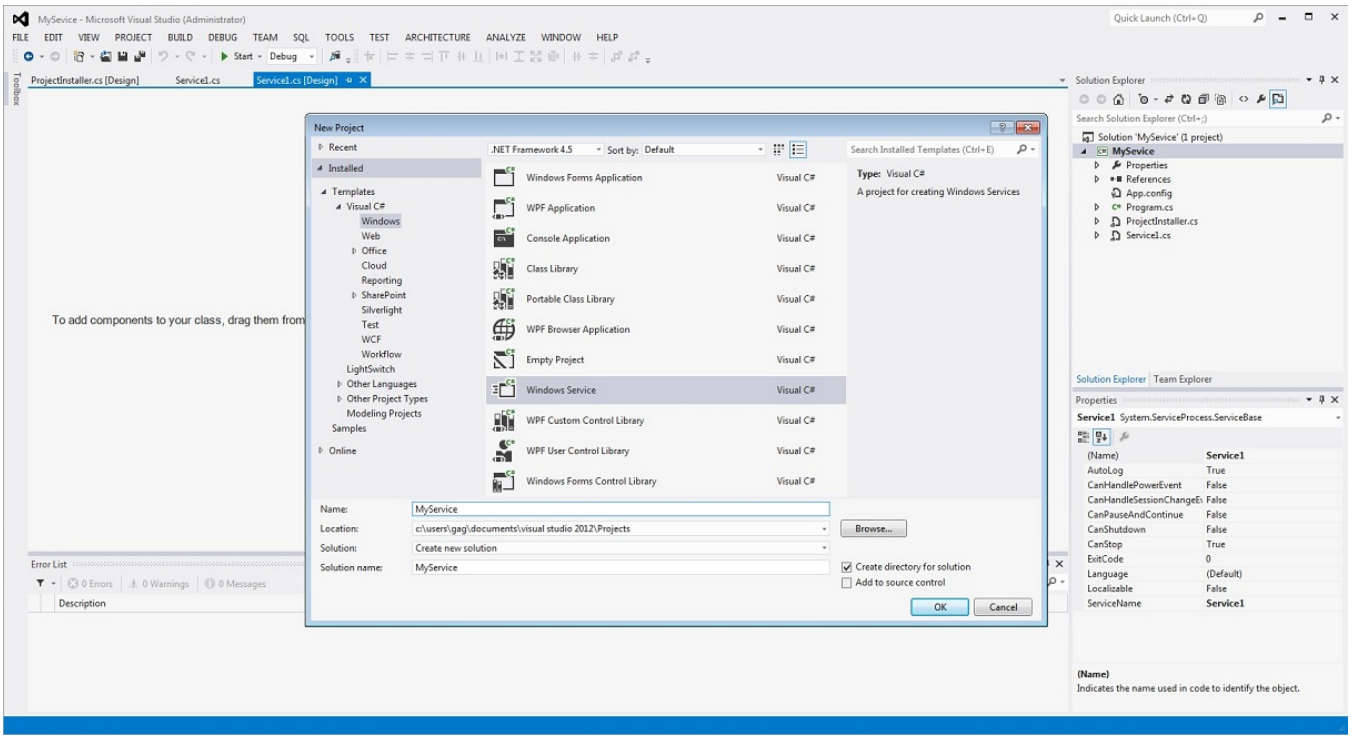
Служба (или сервис) – это специфическое приложение. Как правило, служба не имеет пользовательского интерфейса и не создает видимого вывода. Службу нельзя запустить как обычное приложение – простой активацией соответствующего .exe файла. Службу надо сначала установить, а уже потом запускать или останавливать. Установить службу можно с помощью утилиты InstallUtil.exe, поставляемой вместе с .NET Framework. Или же можно использовать установщик Microsoft Installer (MSI), который создается специальным проектом.

С точки зрения развития технологии программирования, службы являются следующей структурной единицей, из которой строятся программы. Следующей, после функций, классов и объектов.

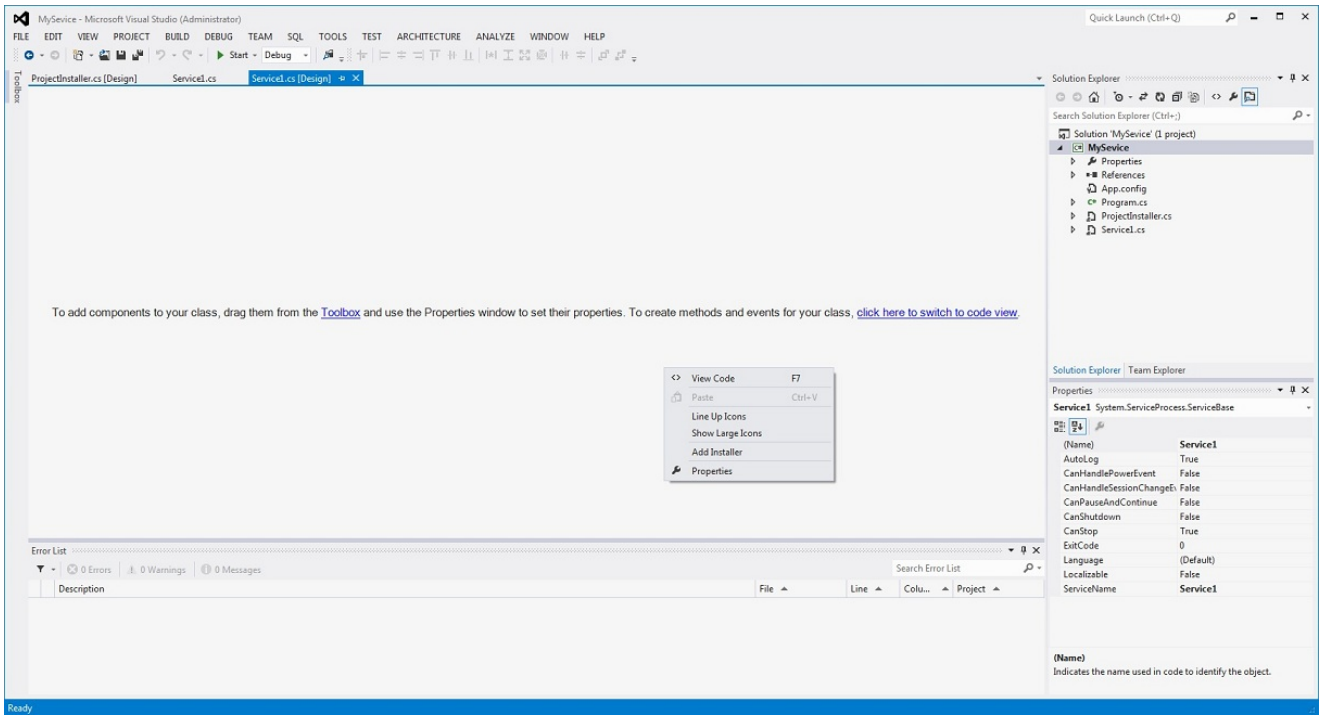
Службы управляются оболочкой Service Control Manager, в которой их можно запускать, останавливать, перезапускать. Службы идеально подходят для выполнения серверных ролей. В ОС Windows службы появились начиная с версии Windows NT.

Создадим в качестве примера обычную службу Windows. Она не будет иметь никакого отношения к WCF, но будет являться хорошей иллюстрацией тех специфических черт служб, о которых мы только что сказали. Ничего практически полезного наша служба делать не будет, но всякий раз при запуске или остановке она будет заносить запись в некоторый текстовый файл, указывая о произошедшем событии и отмечая момент времени, в который это событие произошло. Все проекты приводимые в этих уроках разработаны в Visual Studio 2012 под управлением 64-разрядной ОС Windows 7.
Для создания такой службы с помощью Visual Studio 2012 необходимо выполнить следующее:

1. Начать новый проект, например с именем MyService;
2. Из списка доступных шаблонов выбрать Windows Service;



3. Visual Studio предлагает вам просмотр файла Service1.cs в режиме дизайнера.
4. Нажмите на правую кнопку мыши, чтобы переключиться в режим отображения кода.



Visual Studio предлагает шаблон кода, в котором предусмотрены два виртуальных метода – OnStart() и OnStop(), унаследованных из класса System.ServiceProcess.ServiceBase. Эти методы выполняются при передаче службе диспетчером служб команды запуска (при этом будет вызываться метод OnStart()) или команды остановки (при этом будет вызываться метод OnStop()). Если же наша служба будет настроена так, чтобы запускаться при загрузке операционной системы, то метод OnStart() будет вызываться именно в этот момент. В теле этих методов следует указать действия, которые наша служба должна выполнять при запуске и остановке. Реализуем эти методы следующим образом:

```
protected override void OnStart(string[] args)
{
    this.MyLog("Service Started");
}

protected override void OnStop()
{
    this.MyLog("Service Stopped");
}
```

Как вы видите, оба метода ссылаются на метод MyLog(). Создадим этот метод, который и будет выполнять запись в файл:

```
private void MyLog(string msg)
{
    StreamWriter sw = null;
    try
    {
        sw = new StreamWriter("svclog.txt", true);
        msg += "\t\t";
        msg += DateTime.Now.ToLongTimeString();
```

```
            sw.WriteLine(msg);
        }
        catch (Exception ex)
        {
            StreamWriter error = new StreamWriter("errors.txt", true);
            error.WriteLine(ex.Message);
            error.Close();
        }
        finally
        {
            sw.Close();
        }
    }
```

Метод MyLog() добавляет к полученному строковому параметру значение текущего момента времени, тоже в виде строки, и записывает все это в файл. Обратите внимание, что в случае возникновения исключительной ситуации информация об этом будет записана в другой файл.

Теперь наша служба готова. Смотрите ***приложение к уроку «просто служба Windows»***. Но запускать службу как обычное приложение нельзя. Службу надо сначала установить с помощью специальной утилиты InstallUtil.exe. Чтобы утилита InstallUtil.exe могла установить службу, необходимо добавить к нашей службе установщик. Для этого надо выполнить следующие действия в Visual Studio:

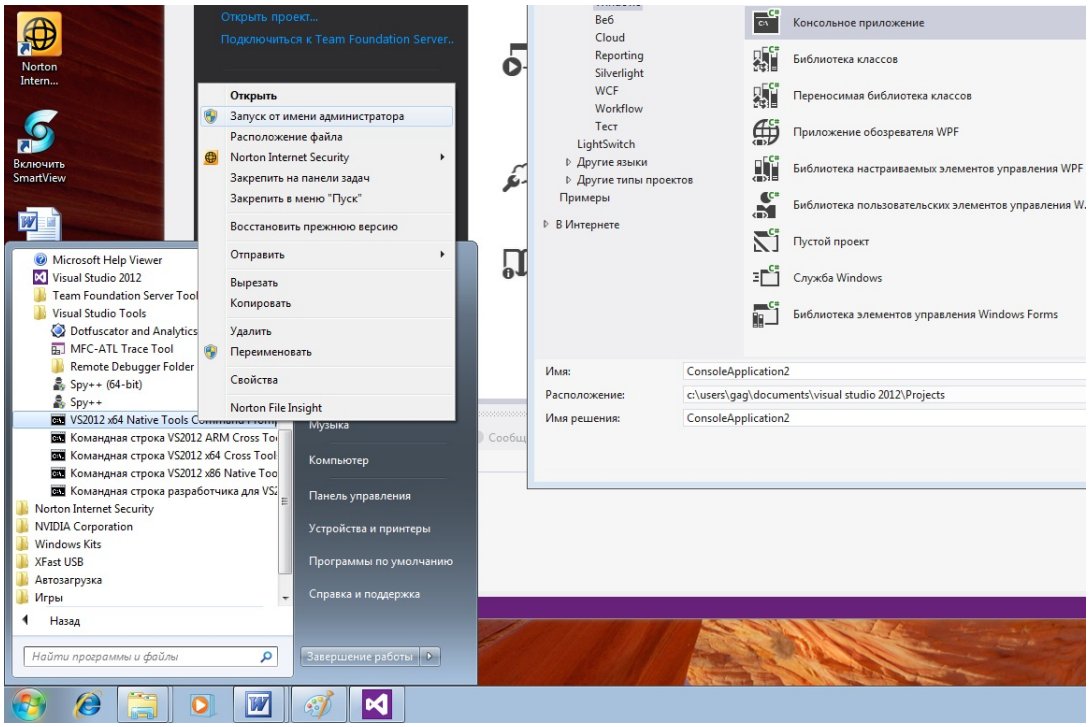
1. Переключиться в Visual Studio в режим дизайнера;
2. На пустом месте в дизайнере нажать правую кнопку мыши и выбрать команду «добавить установщик»;
3. В обозревателе решения проекта появится компонента ProjectInstaller, ее надо открыть в режиме дизайнера;
4. Установить значения для двух свойств компоненты serviceInstaller1: свойству ServiceName присвоить значение MyService, а свойству StartType – значение Automatic;
5. Установить свойство компоненты serviceProcessInstaller1 Account равным LocalSystem;
6. Перестроить проект.

Теперь надо установить службу, чтобы можно было с ней работать. Для установки службы воспользуемся утилитой InstallUtil.exe. Для этого надо:

1. Запустить командную строку Visual Studio 2012 от имени администратора (для Visual Studio 2012 надо набрать – Пуск-Все программы-Microsoft Visual Studio 2012 -Visual Studio Tools-командная строка разработчика для VS2012, затем нажать правую кнопку мыши и активировать команду «запустить от имени администратора»);
2. В консольном окне командной строки перейти в папку bin\Debug (или bin\Release) нашего проекта (в консольном окне командной строки наберите

команду cd «путь_к_exe_файлу_службы» и нажмите Enter, при этом путь к службе можно скопировать из окна проводника и вставить в консольное окно правой кнопкой мыши);

3. В консольном окне командной строки набрать и выполнить команду InstallUtil MyService.exe. (Чтобы удалить нашу службу, надо будет ввести команду InstallUtil /u MyService.exe).



Если служба нормально установлена, мы можем получить к ней доступ через Service Control Manager. Service Control Manager можно запустить, введя в командное окно (Win+R) команду services.msc. В появившемся окне менеджера служб надо найти нашу службу с именем MyService. Используя правую клавишу мыши, несколько раз выполнить запуск и остановку нашей службы. Теперь надо найти в каталоге C:\Windows\SysWOW64 наш файл svclog.txt, открыть его и просмотреть сообщения, записанные при запусках и остановках службы.
%SystemRoot%\SysWOW64 – это еще один системный каталог для 64-разрядной ОС. Странное имя этого каталога представляет собой аббревиатуру (WindowsOnWindows64). SysWOW64 является системным для 32-разрядных приложений, запускаемых в 64-разрядной ОС, в то время как в System32 хранятся 64-разрядные библиотеки. Да, да, это не оговорка. В папке %SystemRoot%\System32 лежат 64-разрядные dll, а в папке %SystemRoot%\SysWOW64 лежат 32-разрядные dll, а в папке %SystemRoot%\System лежат, конечно же, 16-разрядные dll.
Мы кратко рассмотрели процесс создания и установки службы и теперь с этим опытом перейдем к изучению WCF. Начнем с основы WCF – со служб.

Служба WCF представляет собой программу, состоящую из трех основных компонент:

- **класса службы**, созданного на любом языке .NET и содержащего один или несколько методов;
- **набора оконечных точек** (endpoints). Каждая оконечная точка – это средство взаимодействия службы с внешним миром.
- **процесса**, в котором данная служба выполняется (хостится);

b. Контракты

Разберем поочередно все три компоненты **WCF** служб. В классе службы надо определять те методы, которые служба сможет выполнять по запросам клиентов. Такие методы называются операциями. Забегая вперед, можно сказать, что не все методы, определенные в классе службы, доступны клиентам. Разработчик класса службы сам определяет, какие методы будут доступны клиентам. Набор методов, доступных клиентам, представляет собой так называемый контракт службы.

При работе в **WCF** важную роль играют такие инструменты программирования, как атрибуты. Атрибуты, как и другие классы WCF, определены в пространстве имен System.ServiceModel. Поэтому каждое WCF-приложение должно включать ссылку на это пространство имен. Чтобы превратить класс, написанный на любом языке .NET, в класс службы, определение этого класса надо предварить атрибутом [**ServiceContract**]. А чтобы указать, какие из методов этого класса будут доступны клиентам, перед каждым таким методом надо указать атрибут [**OperationContract**]. Набор методов с атрибутами [**OperationContract**] представляет собой контракт службы. Однако, как вы скоро увидите, существует более гибкий способ задания контракта службы – в интерфейсе. При этом атрибуты [**ServiceContract**] и [**OperationContract**] применяются к интерфейсу и его членам. В этом случае класс службы должен наследовать интерфейсу, в котором определен контракт службы.

Обратите внимание на то, что именно атрибут [**OperationContract**], указанный возле метода в классе службы, делает этот метод доступным клиенту. Этот атрибут можно применять только к методам, private или public, но никак не к свойствам, полям или другим членам класса. Класс службы может содержать свойства, индексаторы, статические члены, но все они будут недоступны потребителям службы и никакими атрибутами их нельзя сделать видимыми для клиентов. Кроме этого, **WCF** может использовать в классе службы только конструктор по умолчанию. Конструкторы с параметрами не доступны **WCF**.

Итак, контракт службы – это набор методов с атрибутами [**OperationContract**], определенных в классе с атрибутом [**ServiceContract**] или в классе, наследующем интерфейс с атрибутом [**ServiceContract**]. Именно контракт службы определяет функциональность, которую служба представляет внешнему миру. По умолчанию имя

контракта совпадает с именем интерфейса (или класса). Однако свойство Name атрибута [**ServiceContract**] позволяет явно задать имя контракта службы.

Например, определим контракт в интерфейсе. В этом случае имя контракта будет таким же, как имя интерфейса – **IContract**:

```
[ServiceContract]
interface IContract
{
    [OperationContract]
    void Method1(int a);
}
```

А в этом случае – имя контракта будет отличаться от имени интерфейса и будет **ISpecialContract**:

```
[ServiceContract(Name = "ISpecialContract")]
interface IContract
{
    [OperationContract]
    void Method1(int a);
}
```

Аналогичным образом свойство Name атрибута [**OperationContract**] позволяет явно задавать имя операции, которое по умолчанию совпадает с именем метода. Например, в этом случае имя операции реализованной в Method1() будет SpecialMethod:

```
[ServiceContract]
interface IContract
{
    [OperationContract(Name = "SpecialMethod")]
    void Method1(int a);
}
```

Контракт **WCF** службы описывается стандартным классом **ContractDescription**. Подробное описание этого класса приводится в MSDN. Однако будет уместным отметить здесь некоторые основные характеристики этого класса. Объект ContractDescription описывает контракт WCF службы и все операции контракта. Свойства Name и Namespace содержат имя самого контракта и имя пространства имен, в котором определен контракт. Коллекция Operations содержит описание операций, предоставляемых контрактом внешнему миру. Коллекция Behaviors содержит набор поведений данного контракта. Объект ContractDescription можно создать одним из перегруженных конструкторов класса, а можно получить с помощью статического метода ContractDescription.GetContract(), тоже перегруженного. Имея в своем распоряжении объект ContractDescription можно получить много полезной информации о службе.

c. Оконечные точки

Оконечные точки службы позволяют клиентам находить службу и общаться с ней. Каждая оконечная точка имеет три важные характеристики: адрес, привязку и контракт. О контрактах службы мы поговорим немного позже. По-английски эта аббревиатура звучит как ABC (address, binding, contract). Запомните этот термин, так как он часто используется и в русскоязычной литературе.

i. адрес оконечной точки

Каждая служба имеет уникальный адрес, который заключает в себе две важные характеристики службы – место расположения службы и транспортный протокол, по которому со службой можно взаимодействовать. Структура адреса службы такая:

[transport]://[machinename or domainname][: port]/[URI]

Здесь:
transport - указывает транспортный протокол;
machinename or **domainname** – указывает имя машины, на которой размещена служба;
port – указывает номер порта на машине (может быть не указанным, в этом случае будет использоваться значение порта по умолчанию);
URI (Universal Resource Identifier) – произвольная строка, указывающая имя службы;
Часть адреса службы, состоящую из [transport]://[machinename or domainname][:port] часто называют базовым адресом. В дальнейшем мы еще будем пользоваться этим понятием.
Обратите внимание на тот факт, что в WCF реализация службы не зависит от того, будет ли эта служба использоваться как локальная или как удаленная. В WCF всегда службы реализуются, как удаленные.

Рассмотрим примеры адресов.
http://localhost:8001/MyService - пример адреса службы с именем MyService, использующей HTTP в качестве транспортного протокола. Вместо HTTP можно использовать также HTTPS протокол, для обеспечения защиты пересылаемых сообщений. Номер порта в таком адресе можно не указывать, в этом случае будут использоваться порты по умолчанию: 80 для протокола HTTP, и 443 – для протокола HTTPS.

net.tcp://127.0.0.1:8001/MyService - пример адреса службы с именем MyService, использующей TCP в качестве транспортного протокола. Номер порта в таком адресе тоже можно не указывать, в этом случае по умолчанию будет использоваться порт 808.

net.pipe://localhost/MyService - пример адреса службы с именем MyService, использующей в качестве транспортного протокола именованные каналы (named pipes). Надо сказать, что в WCF именованные каналы можно использовать только в пределах одной машины, поэтому в качестве machinename можно указывать только либо имя локальной машины, либо localhost. К тому же на машине можно открывать только один именованный канал.

net.msmq://localhost/MyService - пример адреса службы с именем MyService, использующей в качестве транспортного протокола очередь сообщений Microsoft Message Queue (MSMQ). MSMQ – это специальная служба очередей сообщений. Эта служба входит в стандартную поставку Windows, начиная с версии Windows 2000 Server. Использование MSMQ позволяет реализовать общение приложений, работающих на разных компьютерах в распределенной и разнородной среде. Эта служба позволяет приложениям обмениваться сообщениями даже в том случае, когда некоторые из приложений какое-то время не активны, или какие-либо из сетей временно отключаются. Такая возможность достигается тем, что сообщения заносятся в специальные очереди и считываются оттуда, когда соответствующие приложения начинают работу или возобновляют работу после перерыва. Данный вид пересылки сообщений обеспечивает контроль доставки, эффективную маршрутизацию, безопасность и очередность обработки сообщений.

Поскольку именно оконечные точки службы являются средством общения службы с клиентом, то к указанным адресам службы можно добавлять еще и имя конкретной оконечной точки, т.к. служба может иметь несколько оконечных точек. Например, для оконечной точки с именем ep1, полный адрес этой точки в случае протокола HTTP может иметь вид http://localhost /myservice/ep1, а в случае протокола TCP – net.tcp://localhost:8002/myservice/ep1. Таким образом, адрес оконечной точки содержит в себе определения места расположения, привязки и контракта службы. Каждая служба обязана предъявить внешнему миру по меньшей мере одну оконечную точку. В свою очередь, каждая оконечная точка может представлять только один контракт. Повторимся, что одна служба может иметь много оконечных точек, но все эти точки должны иметь уникальные адреса. Адрес оконечной точки WCF описывается стандартным классом **EndpointAddress**. Подробное описание этого класса приводится в MSDN. Имя контракта в оконечной точке говорит о том, какой контракт службы доступен через эту оконечную точку. Дело в том, что если описывать контракты службы с помощью интерфейсов, то в одной службе можно реализовать сразу несколько контрактов. Для этого надо просто указать в определении класса службы, что он наследует нескольким интерфейсам. Так вот, разные оконечные точки службы могут предоставлять клиентам доступы к разным контрактам, а следовательно – к разным методам службы.

d. Привязка

Давайте подумаем, каким образом может осуществляться пересылка сообщений между клиентом и службой. Эта пересылка может быть синхронной, когда клиент отправляет службе запрос и блокируется до момента получения ответа от службы, а может быть асинхронной, когда блокировка клиента не осуществляется. Сообщения могут пересылаться либо только от клиента к службе, либо в обоих направлениях. Сообщения могут доставляться немедленно, а могут выстраиваться в очереди. Они

могут кодироваться при пересылке, а могут пересылаться в открытом виде. Для пересылки сообщений может использоваться один из следующих транспортных протоколов – HTTP, HTTPS, TCP, named pipes или MSMQ. Иногда сообщения можно обрабатывать в той последовательности, в которой они получены, а иногда перед обработкой необходимо восстанавливать очередность отправки сообщений. Как вы видите, существует очень много нюансов пересылки сообщений. Вы должны понимать, что количество возможных комбинаций всех этих аспектов очень велико. К тому же необходимо учитывать, что некоторые опции не совместимы друг с другом, а некоторые требуют только определенного соответствия с другими. Именно для управления всеми этими моментами в WCF применяется привязка (binding). Если вы помните, что было сказано ранее о цепочках каналов, то вам легко будет понять, что привязка – это и есть такой набор каналов. Иначе говоря, привязка – это набор определенного числа протокольных и транспортного каналов. Часто используют термин «стек каналов».

Чтобы упростить выбор привязки, WCF предлагает набор предопределенных привязок, которые удовлетворяют большинство случаев применения. Каждая из предопределенных привязок может в определенной степени настраиваться, путем изменения своих свойств. Если же стандартные привязки не подходят в каком-либо случае, WCF предлагает механизм конструирования специфических привязок.

Перечислим основные предопределенные привязки WCF для версии .NET Framework 3.5:

- **basicHttpBinding** – простая привязка для веб-служб, не обеспечивает безопасность сообщений;
- **wsHttpBinding** – привязка для веб-служб, дополнительно реализующая обеспечение безопасности пересылаемых данных, возможность создания транзакций. Основное отличие этой привязки от привязки basicHttpBinding заключается в способе пересылки данных между службой и клиентом. При использовании basicHttpBinding, данные пересылаются в виде простого текста, а при использовании wsHttpBinding, данные автоматически кодируются;
- **wsDualHttpBinding** – привязка для веб-служб, поддерживающая двусторонний обмен сообщениями для дуплексных контрактов, но не обеспечивающая безопасность на транспортном уровне;
- **netTcpBinding** – привязка, реализующая обмен сообщениями между .NET приложениями как на одной машине, так и в сети, на основе протокола TCP/IP, поддерживающая дуплексные контракты;
- **netNamedPipeBinding** – привязка, реализующая обмен сообщениями по именованным каналам в пределах только одной машины;

- **netMsmqBinding** – привязка, реализующая обмен сообщениями с использованием служб Microsoft Message Queueing (MSMQ), реализует асинхронный способ общения;
- **netPeerTcpBinding** – привязка, реализующая обмен сообщениями в пиринговых сетях, т.е. в сетях, где не существует явных узлов-серверов и узлов-клиентов, а каждый узел может выступать и в той, и в другой роли;

Полезно будет запомнить следующие моменты. Если имя привязки начинается с префикса ws (Web-service) это значит, что такая привязка может использоваться для общения между службами и клиентами, написанными на разных платформах. А если имя привязки начинается с префикса net, то такая привязка должна использоваться для общения между службами и клиентами, написанными в среде .NET.

Надо также отметить, что элементы привязки WCF представляет класс стандартный **Binding**. Подробное описание этого класса приводится в MSDN. Три упоминавшиеся ранее стандартные классы (EndpointAddress, Binding и ContractDescription) собраны воедино в стандартном классе **ServiceEndpoint,** который представляет оконечную точку и имеет три важных свойства: свойство Address типа EndpointAddress, свойство Binding типа Binding и свойство Contract типа ContractDescription.

i. контроль доставки сообщений

Помните, чем отличается протокол TCP/IP от протокола UDP в транспортном, так сказать, смысле? TCP/IP это протокол с контролем доставки. Этот протокол позволяет нам быть уверенными в том, что для доставки нашего сообщения будет предпринято очень много усилий. Возникает вопрос: предоставляет ли WCF нам с вами основания быть уверенными в том, что сообщения будут обязательно доставлены адресату? Ответ на этот вопрос заключается в том, что WCF предоставляет нам механизм, позволяющий реализовать систему контроля доставки и не только доставки. Понятно, что этот механизм также реализуется привязками. Некоторые привязки позволяют настроить себя так, чтобы обеспечивать надежность доставки сообщений, другие привязки не могут это делать принципиально.

Давайте рассмотрим, какие привязки позволяют контролировать доставку сообщений, и кроме этого – соблюдать очередность отправки сообщений. Привязка **BasicHttpBinding** не умеет делать ни одного, ни другого. Привязка **NetTcpBinding** умеет обеспечивать контроль доставки сообщений, хотя по умолчанию эта опция выключена. Кроме того, эта привязка обеспечивает очередность сообщений и эта опция по умолчанию включена. Точно так же как **NetTcpBinding** ведет себя привязка **WSHttpBinding**. И значения умолчаний у нее тоже совпадают с привязкой **NetTcpBinding**. А привязка **NetMsmqBinding** ведет в этом отношении себя так же, как привязка **BasicHttpBinding**.

Управлять контролем доставки можно как программно, так и альтернативным способом, о котором вы узнаете позже в этом уроке. Программно это можно сделать двумя способами.

Первый способ предполагает использование конструктора. Класс **NetTcpBinding** имеет перегруженный конструктор с двумя аргументами. Первый – типа перечисления **SecurityMode** позволяющий задавать уровень защиты пересылаемых сообщений:

- если этот параметр равен **SecurityMode**.**None** – никакая защита не обеспечивается;
- если этот параметр равен **SecurityMode.Transport** – защита обеспечивается при условии, что ее поддерживает транспортный протокол, например HTTPS;
- если этот параметр равен **SecurityMode.Message** – защита обеспечивается механизмом SOAP;
- если этот параметр равен **SecurityMode.TransportWithMessageCredential** – транспортный протокол обеспечивает защиту сообщений, а SOAP средства обеспечивают механизм аутентификации клиентов.

Второй аргумент конструктора – булевского типа. Если он равен true, значит механизм контроля доставки будет включен. Вызов этого конструктора может выглядеть так:

```
NetTcpBinding tcp = new NetTcpBinding(SecurityMode.Message, true);
```

Второй способ обеспечения контроля доставки заключается в инициализации свойства **ReliableSession** после создания привязки любым конструктором. Это свойство имеет тип **OptionalReliableSession**, а для инициализации надо обращаться к его булевскому свойству Ordered.

```
NetTcpBinding tcpbind = new NetTcpBinding();
tcpbind.ReliableSession.Ordered = true;
```

Чтобы обеспечить очередность обработки сообщений в том порядке, как они были отправлены, нам необходимо познакомится с еще одним атрибутом – **DeliveryRequirements**. Как правило, этот атрибут указывается у контракта службы, например таким образом:

```
[ServiceContract]
[DeliveryRequirements(RequireOrderedDelivery = true)]
interface IContract
{
      [OperationContract]
      void Method1(int a);
}
```

Значение свойства RequireOrderedDelivery этого атрибута по умолчанию равно false, и это значит, что очередность сообщений не отслеживается. Чтобы изменить такое поведение службы, значение этого свойства надо инициализировать true. Но при этом свойства привязки и их текущие выставленные значения должны обеспечивать и контроль доставки сообщений (без чего очередность невозможна) и саму очередность доставки. Если мы будем требовать от контракта службы соблюдения очередности, а

привязка не будет для этого настроена соответствующим образом – мы получим исключение InvalidOperationException.

У этого атрибута есть еще одно полезное свойство TargetContract, которое позволяет уточнить, операции какого контракта службы должны поддерживать очередность сообщений. Это полезно в случае наличия нескольких контрактов служб. В таком случае этот атрибут указывают не возле контракта службы, а возле класса, реализующего разные контракты:

```
[DeliveryRequirements(TargetContract = typeof(IMyContract),
      RequireOrderedDelivery = true)]
public class MyService : IMyContract, ISecondContract, IThirdContract
{
      . . .
}
```

При таком использовании атрибута DeliveryRequirements обеспечивать очередность обработки сообщений будут только методы, реализующие контракт IMyContract.

ii. создание пользовательской привязки

Мы уже говорили о том, что привязка – это набор определенного числа протокольных и транспортного каналов. Сейчас мы проделаем эксперимент, подтверждающий, что это утверждение истинно. Вы должны помнить, что все привязки реализуются на основании абстрактного класса **Binding**, определенного в пространстве имен System.ServiceModel.Channels. В этом классе определен метод **CreateBindingElements()**, который предназначен для отображения коллекции элементов базовых привязок, из которых состоит текущая привязка. Базовые привязки – это привязки, из которых состоят все системные привязки WCF, и из которых мы сами можем создавать пользовательские привязки.

Давайте создадим простое консольное приложение на C# следующего вида:

Пример 1. Рассмотрение структуры привязки

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace LookAtBinding
{
    class Program
    {
        static void Main(string[] args)
        {
            WSHttpBinding b = new WSHttpBinding();
            Console.WriteLine("Анализируемая привязка {0} состоит из:",
                  b.GetType().Name);
```

```
            BindingElementCollection elements = b.CreateBindingElements();
            foreach (BindingElement element in elements)
                Console.WriteLine("Базовая привязка {0}", element.GetType().FullName);
            Console.WriteLine();
        }
    }
}
```

Если вы запустите это приложение, то увидите следующее:

```
Анализируемая привязка WSHttpBinding состоит из:
Базовая привязка  System.ServiceModel.Channels.TransactionFlowBindingElement
Базовая привязка  System.ServiceModel.Channels.SymmetricSecurityBindingElement
Базовая привязка  System.ServiceModel.Channels.TextMessageEncodingBindingElement
Базовая привязка  System.ServiceModel.Channels.HttpTransportBindingElement

Для продолжения нажмите любую клавишу . . .
```

Другими словами, наше маленькое приложение показывает нам, что анализируемая системная привязка WSHttpBinding состоит из четырех базовых привязок, среди которых одна – транспортная (HttpTransportBindingElement), остальные – протокольные. В частности, элемент TextMessageEncodingBindingElement определяет тип кодирования сообщения при пересылке с использованием анализируемой нами привязки WSHttpBinding. Аналогичным образом, в этом приложении вы можете рассмотреть внутреннюю структуру других системных привязок WCF.

Однако иногда может оказаться, что ни одна из имеющихся готовых системных привязок не удовлетворяет нас по каким-либо причинам. В этом случае мы можем создавать собственные привязки из имеющихся базовых. Рассмотрим этот процесс. Чаще всего для этого используют класс **CustomBinding**, определенный в пространстве имен System.ServiceModel.Channels. Этот класс является чем-то вроде контейнера, содержащего коллекцию базовых элементов. Эту коллекцию мы можем формировать самостоятельно. Необходимо только учитывать состав и порядок расположения элементов в коллекции. А именно – в любой привязке обязательно должны присутствовать кодировочный и транспортный элементы. Кроме этого, транспортный элемент должен располагаться в самом низу коллекции. Создание пользовательской привязки может выглядеть, например, таким образом:

```
CustomBinding MyBinding = new CustomBinding();
MyBinding.Elements.Add(new TextMessageEncodingBindingElement());
MyBinding.Elements.Add(new HttpsTransportBindingElement());
```

Назначение двух базовых элементов должно быть вам понятно. О других способах создания пользовательских привязок поговорим в конце текущего урока.

3. Рассмотрение механизма хостинга

Служба обязательно должна быть размещена в каком-нибудь работающем процессе ОС. Таким процессом может быть служба Windows, IIS или обычное консольное приложение. Служба создает экземпляр класса ServiceHost, который отвечает за

создание оконечных точек службы и реализует среду выполнения службы. Все способы хостинга служб мы рассмотрим далее.

Начнем это знакомство с механизмами хостинга в консольных приложениях C#. Любой метод Main() запускается операционной системой Windows в отдельном процессе, поэтому вполне может использоваться для хостинга WCF службы. Все что надо сделать для этого программисту – это реализовать среду выполнения для класса службы. Для этих целей служит класс ServiceHost.

Объект класса ServiceHost выполняет большой объем работы. Он автоматически формирует полное описание службы в своем свойстве Description, представляющем собой объект класса ServiceDescription. Кроме этого, ServiceHost создает и конфигурирует оконечные точки службы, применяет настройки безопасности и начинает прослушивание входящих запросов от клиентов. Различные способы использования этого класса мы рассмотрим в примерах.

Новые термины и понятия

Мы встретили целый ряд новых понятий, поэтому будет полезно еще раз повторить их:

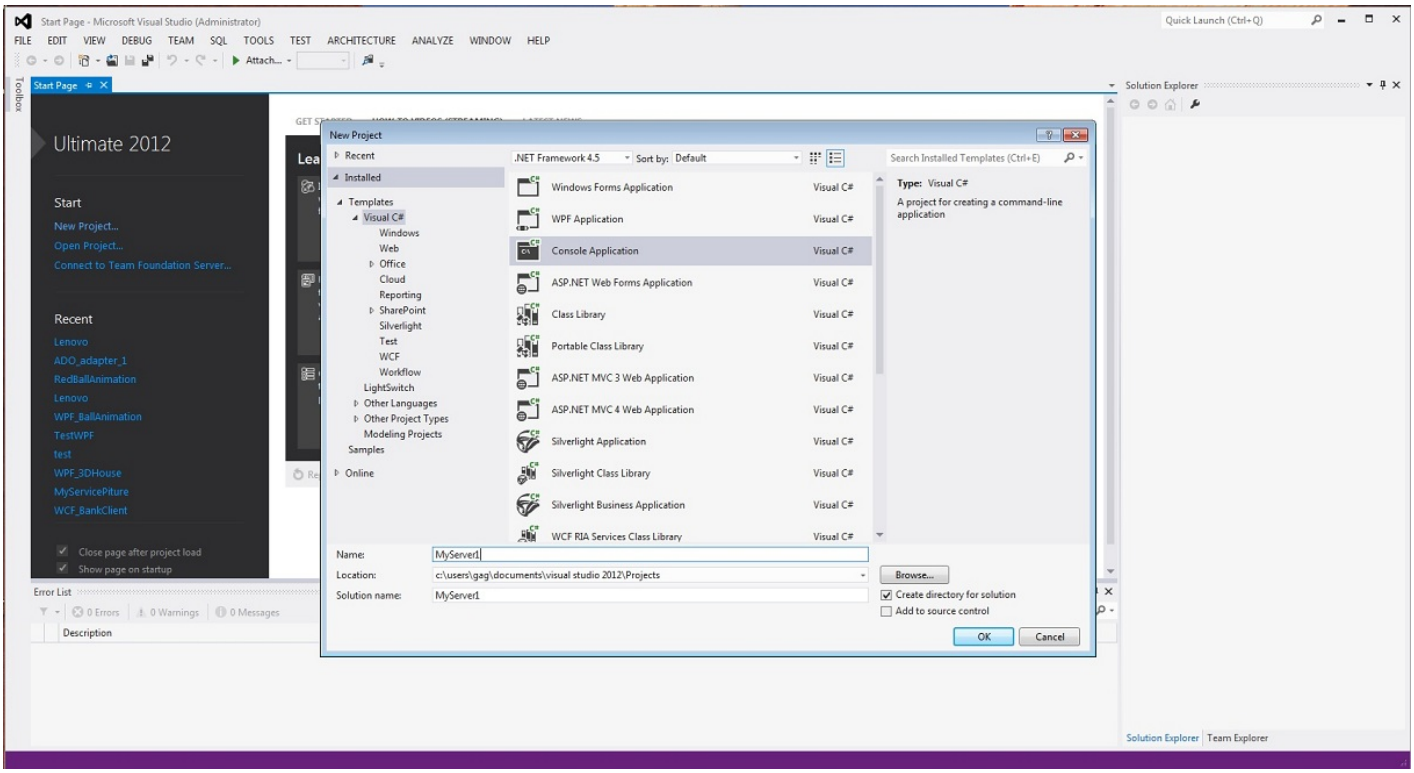
Служба (или сервис) – особый вид приложения, не имеющий пользовательского интерфейса и активируемый с помощью специальных инструментов;
Служба WCF – приложение, состоящее из трех основных компонент: класса службы, набора оконечных точек и процесса, в котором данная служба выполняется;
Контракт службы – методы, объявленные в классе службы с атрибутом [OperationContract];
Оконечные точки – средство доступа к службе из внешнего мира, описывает адрес службы, привязку и контракт;
Привязка – это набор протокольных и транспортного каналов.

4. Пример простой программы

a. Служба в консольном приложении

Приступим к созданию простого SOA приложения, состоящего из службы WCF (в роли серверной части) и клиентского приложения. В ходе реализации этого приложения рассмотрим базовые принципы WCF, а затем, усложняя этот пример, будем разбираться с WCF глубже. Создадим службу, которая будет складывать передаваемые ей клиентами два целочисленных значения и возвращать их сумму.

Начнем с серверной части – со службы. Наша служба будет хоститься в консольном приложении. Как мы упоминали ранее, никаких ограничений для размещения служб не существует, и консольное приложение вполне может быть тем процессом, в котором выполняется служба. Рассмотрим сначала принципы создания класса службы и определение контракта, а затем перейдем к работе в Visual Studio.
Создадим консольный проект.



Мы могли бы создать в пространстве имен проекта класс службы MyMath, содержащий метод с именем, например, Add();

```
public class MyMath
{
	public int Add(int a, int b)
	{
		return a + b;
	}
}
```

Можно отметить, что имена класса и метода (MyMath и Add) совершенно произвольны.

Чтобы указать, что данный класс является классом службы, перед ним надо добавить атрибут `[ServiceContract]`, определенный в пространстве имен System.ServiceModel. А перед теми методами этого класса, которые мы хотим сделать доступными клиентам, надо указать атрибут `[OperationContract]`:

```
[ServiceContract]
public class MyMath
{
	[OperationContract]
	public int Add(int a, int b)
	{
		return a + b;
	}
}
```

В классе службы может быть определено много методов, но внешнему миру будут доступны только методы, помеченные атрибутом `[OperationContract]`.

Если в нашем классе атрибут [OperationContract] указан только у метода Add(), только этот метод и будет доступен клиентам нашей службы. Если вспомнить, что контракт – это набор методов, предоставляемых службой своим клиентам, то теперь мы могли бы сказать, что наш класс MyMath реализует контракт службы, в котором определена только одна операция с именем Add(). Другими словами, все клиенты, работающие с нашей службой, могут вызывать метод Add() этой службы и только его.

Однако на практике вышеописанным способом создания контрактов не пользуются. Вместо этого определение контракта выносят в интерфейс:

```
[ServiceContract]
public interface IMyMath
{
	[OperationContract]
	int Add(int a, int b);
}

public class MyMath: IMyMath
{
	public int Add(int a, int b)
	{
		return a + b;
	}
}
```

Обратите внимание – теперь атрибуты указаны в интерфейсе, а не в классе, а класс наследует интерфейсу. Такой подход является более гибким, так как позволяет одному классу реализовать несколько разных контрактов, описанных в разных интерфейсах. Для этого надо, чтобы класс наследовал всем интерфейсам, реализующим контракты. Кроме того, такой поход позволяет гибко модифицировать службу, добавляя новые интерфейсы, реализующие обновления, и оставляя активными старые интерфейсы для сохранения преемственности. Этим способом мы и будем пользоваться.

После определения класса службы необходимо будет решить вопрос о том, в каком процессе служба будет хоститься. Мы рассмотрим разные способы решения этого вопроса по ходу наших занятий. А когда служба будет готова, рассмотри способы создания клиента.

После этих замечаний перейдем к практике.

1. Запустим Visual Studio и начнем новый консольный проект с именем MyServer1;
2. Добавим ссылку на пространство имен `System.ServiceModel` и добавим строку «`using System.ServiceModel;`». Это действие будем выполнять всякий раз при написании как службы, так и клиента WCF;
3. В пространстве имен MyServer1 создадим класс службы MyMath, в котором определим метод Add(), а контракт опишем в интерфейсе:

```
[ServiceContract]
public interface IMyMath
{
	[OperationContract]
	int Add(int a, int b);
}

public class MyMath: IMyMath
{
	public int Add(int a, int b)
	{
		return a + b;
	}
}
```

4. В методе Main() необходимо реализовать среду выполнения для класса MyMath. Для этих целей служит класс ServiceHost. Мы поговорим об этом классе полдробнее немного позже.

```
ServiceHost sh = new ServiceHost(typeof(MyMath));
```

5. В классе ServiceHost определен метод AddServiceEndpoint(), позволяющий создавать оконечные точки службы. Воспользуемся этим методом, чтобы создать в нашей службе оконечную точку с именем Ep1:

```
sh.AddServiceEndpoint(
	typeof(IMyMath), //контракт
	new WSHttpBinding(), //тип привязки
	"http://localhost/MyMath/Ep1"); //адрес и имя оконечной точки
```

Метод AddServiceEndpoint() является перегруженным и позволяет указывать характеристики оконечных точек разными способами. Все варианты перегрузки можно посмотреть здесь:
http://msdn.microsoft.com/en-us/library/system.servicemodel.servicehost.aspx

6. Активизируем среду выполнения:

```
sh.Open();
```

7. Наша служба после запуска должна оставаться активной, чтобы принимать подключения клиентов. Реализуем это простым способом, предусмотрев следующее завершение работы службы:

```
Console.WriteLine("Для завершения нажмите <ENTER>\n");
Console.ReadLine();
sh.Close();
```

Как видите, наша служба будет оставаться активной, пока пользователь не нажмет клавишу ENTER. Готовый код сервера должен выглядеть следующим образом:

Пример 2. Служба в консольном приложении

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;

namespace MyServer1
{
```

```
        [ServiceContract]
        public interface IMyMath
        {
               [OperationContract]
               int Add(int a, int b);
        }

        public class MyMath: IMyMath
        {
               public int Add(int a, int b)
               {
                      return a + b;
               }
        }

        class Program
        {
          static void Main(string[] args)
           {
               ServiceHost sh = new ServiceHost(typeof(MyMath));

               sh.AddServiceEndpoint(
                 typeof(IMyMath),
                 new WSHttpBinding(),
                 "http://localhost/MyMath/Ep1");

               sh.Open();
               Console.WriteLine("Для завершения нажмите <ENTER>\n");
               Console.ReadLine();
               sh.Close();
           }
    }
}
```

Обратите внимание на добавление пространства имен `System.ServiceModel`. Осталось запустить наше приложение и полюбоваться консольным окном. Но чтобы радость была полнее, надо создать и клиента, который будет общаться с нашей службой. **Обратите внимание, что запускать наши приложения в Windows 7 надо от имени администратора**. Самый простой способ сделать это – создать ярлык для запуска Visual Studio 2012 и в свойствах ярлыка, под кнопкой «Дополнительно» выставить отметку для опции «Запуск от имени администратора». Можно еще запускать созданные студией .exe файлы проектов по клику правой кнопкой мыши, выбирая опцию «Запуск от имени администратора», но это менее удобный случай. А можно еще добавлять файл манифеста к создаваемым приложениям, но об этом мы поговорим позже.

b. Объект ServiceHost – смотрим внимательнее

Мы создавали среду выполнения для класса нашей службы с помощью объекта класса ServiceHost:

```
ServiceHost sh = new ServiceHost(typeof(MyMath));
```

В нашей службе мы передавали конструктору ServiceHost только тип контракта службы. Но класс ServiceHost предлагает в наше распоряжение перегруженный конструктор. При создании объекта этого класса мы можем также передать конструктору один или несколько базовых адресов службы. Например, таким образом:

```
Uri http_adr1 = new Uri("http://localhost:8081/");
Uri tcp_adr2 = new Uri("net.tcp://localhost:8082/");

ServiceHost sh =
   new ServiceHost(typeof(MyMath), http_adr1, tcp_adr2);
```

Созданная таким образом служба сможет принимать сообщения на разные адреса и по разным протоколам. Обратите внимание, что все указываемые в конструкторе адреса должны использовать разные протоколы. Создаваемые таким образом адреса службы называются базовыми адресами. Так вот, если мы создаем объект класса ServiceHost с указанием базового адреса, тогда мы можем по-разному задавать адрес оконечной точки службы. Рассмотрим такой пример с одним базовым адресом:

```
Uri tcp_base_adr = new Uri("net.tcp://localhost:8000/");
ServiceHost sh = new ServiceHost(typeof(MyMath), tcp_base_adr);
```

Вы помните, что оконечные точки службы мы создавали методом AddServiceEndpoint(). Учитывая, что этому методу адрес передается в виде строки, мы можем указать вместо адреса пустую строку. При этом адрес созданной оконечной точки будет совпадать с базовым адресом (в нашем примере – "net.tcp://localhost:8000/"):

```
sh.AddServiceEndpoint(typeof(MyMath), tcpBinding, "");
```

Если методу AddServiceEndpoint() передать только URI оконечной точки, то полный адрес созданной оконечной точки получится путем слияния базового адреса с URI:

```
sh.AddServiceEndpoint(typeof(MyMath), tcpBinding, "MathService");
```

В этом примере адресом оконечной точки будет – "net.tcp://localhost:8000/MathService".

Если же методу AddServiceEndpoint() передать полный адрес оконечной точки, то он перебьет заданный базовый адрес и оконечная точка будет создана с новым адресом:

```
sh.AddServiceEndpoint(typeof(MyMath), tcpBinding,
      "net.tcp://localhost:8001/MathService ");
```

В этом примере адресом оконечной точки будет – "net.tcp://localhost:8001/MathService".

Базовые адреса службы можно также указывать в конфигурационном файле. Это можно сделать в элементе <services> во вложенном элементе <service> в разделе <host>. Например, таким образом:

```
<services>
      <service name = "Module1Service.MathService">
      <host>
            <baseAddresses>
                  <add baseAddress = "http://localhost:8081/"/>
                  <add baseAddress = "net.tcp://localhost:8082"/>
            </baseAddresses>
      </host>
      ...
      </service>
```

```
    </services>
```

В таком случае объект класса ServiceHost будет использовать как базовые адреса службы, указанные в конфигурационном файле, так и базовые адреса, заданные в коде. Понятно, что базовые адреса указанные разными способами не должны дублироваться.

Продолжаем разговор. Конструктору класса ServiceHost явным образом передается тип контракта службы, поэтому создаваемый объект жестко связан с типом контракта. Если мы захотим чтобы в нашем методе Main() хостилось несколько контрактов служб, мы должны будем создавать свой объект ServiceHost для каждого контракта.

Для закрытия службы необходимо вызвать метод Close() класса ServiceHost. При вызове этого метода среда выполнения службы блокируется на некоторое время, чтобы позволить всем клиентам завершить общение со службой. Только по истечении этого таймаута среда начинает закрываться. По умолчанию время для завершения работы клиентов равно 10 секундам. Однако его можно изменить, используя свойство CloseTimeout, например, таким образом:

```
ServiceHost sh = new ServiceHost(typeof(MyMath));
sh.CloseTimeout = TimeSpan.FromSeconds(20);
sh.Open();
```

Кроме этого, значение таймаута можно также изменять в конфигурационном файле. Это можно сделать, добавив элемент <timeouts> в тот же раздел <host>:

```
<services>
      <service name = "Module1Service.MathService">
      <host>
            <baseAddresses>
                  <add baseAddress = "http://localhost:8081/"/>
                  <add baseAddress = "net.tcp://localhost:8082"/>
            </baseAddresses>
            <timeouts closeTimeout = "00:00:20"/>
      </host>
      ...
      </service>
</services>
```

c. Клиент в консольном приложении

Перейдем к клиенту. Клиент также будет консольным приложением. Никаких ограничений для типа клиентского приложения нет. Клиент может быть и приложением WinForms, и приложением WPF. Но консольный клиент не будет отвлекать вас от сути происходящего и в данной ситуации выглядит предпочтительнее.

Клиент – это программа, которая обменивается сообщениями с одной или несколькими оконечными точками службы. Клиент может также иметь собственную оконечную точку, чтобы получать сообщения от службы. Клиент должен посылать сообщения оконечным точкам службы. Сообщения представляют собой данные в формате XML. Чтобы передать службе осмысленную информацию, клиент должен

знать адрес, привязку и контракт оконечной точки службы. Адрес указывает, куда надо отправлять сообщения, чтобы оконечная точка их получила. Привязка определяет канал для общения с оконечной точкой. Контракт определяет действия, предоставляемые службой через оконечные точки, т.е. операции, которые служба может выполнять, и форматы сообщений для этих операций. Операции, описанные в контракте, реализуются в виде методов класса службы.

Общение клиента и службы осуществляется с помощью каналов. Каналы – это некие абстракции, по которым передаются сообщения. Каналы делятся на транспортные и протокольные. Транспортные каналы предназначены для передачи сообщений между клиентом и службой с помощью какого-либо транспорта, например **HTTP** или **TCP**. Протокольные каналы призваны реализовывать контроль доставки сообщений, обеспечивать безопасность и т.п. Основа протокольных каналов – **SOAP**. Для выполнения своих функций протокольные каналы могут даже преобразовывать сообщения. Однако вся работа по созданию каналов обычно скрыта от пользователя и выполняется автоматически, например с помощью класса **ChannelFactory<>.** Но для использования класса **ChannelFactory<>** необходимо знать тип контракта службы.

Поэтому мы явно укажем в клиенте интерфейс, реализованный в службе, так как именно в интерфейсе описан контракт службы. Хочу подчеркнуть, что сейчас мы создадим наш клиент немного «не по-честному». Предположим, что кто-то подсмотрел контракт службы и «передал» его нам, а мы воспользовались этой информацией при создании клиента. Обратите внимание, что конкретная реализация методов службы клиенту не известна, ему известен только интерфейс, в котором, как вы понимаете, описаны лишь прототипы методов. В нашем случае – прототип лишь одного метода Add(). Теперь перейдем к программированию:

1. Запустим Visual Studio и начнем новый консольный проект с именем MyClient1;
2. Добавим ссылку на пространство имен `System.ServiceModel` и добавим строку `using System.ServiceModel;`
3. Определим интерфейс, к которому собирается обратиться наш клиент.

```
[ServiceContract]
public interface IMyMath
{
        [OperationContract]
        int Add(int a, int b);
}
```

4. Создадим экземпляр класса ChannelFactory<>, который реализует среду выполнения со стороны клиента, а также создает каналы для передачи сообщений между клиентом и службой. Конструктору этого класса надо указать через параметры привязку и адрес оконечной точки, к которой подключается наш клиент, а через generic – контракт:

```
ChannelFactory< IMyMath > factory = new ChannelFactory< IMyMath >(
      new WSHttpBinding(),
      new EndpointAddress("http://localhost/MyMath/Ep1"));
```

5. Теперь посредством экземпляра класса ChannelFactory мы создаем канал, по которому будут передаваться сообщения между нашим клиентом и службой:

```
IMyMath channel = factory.CreateChannel();
```

6. По созданному каналу обращаемся к методу нашей службы:

```
int result = channel.Add(35, 38);
```

Готовый код клиента должен выглядеть следующим образом:

Пример 3. Клиент в консольном приложении

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;

namespace Module1Client
{
      [ServiceContract]
      public interface IMyMath
      {
            [OperationContract]
            int Add(int a, int b);
      }

      class Program
      {
            static void Main(string[] args)
            {
            ChannelFactory< IMyMath > factory = new ChannelFactory< IMyMath >(
                new WSHttpBinding(),
                new EndpointAddress("http://localhost/MyMath/Ep1"));
            IMyMath channel = factory.CreateChannel();
            int result = channel.Add(35, 38);
            Console.WriteLine("result: {0}", result);
            Console.WriteLine("Для завершения нажмите <ENTER>\n");
            Console.ReadLine();
            factory.Close();
        }
    }
}
```

Снова обратите внимание на добавление пространства имен `System.ServiceModel` (больше напоминать об этом не будем).

Давайте протестируем наше приложение. Запустите серверную часть нашего приложения и оставьте ее активной. Затем запустите клиента. Если вы все сделали

правильно, то в консольном окне клиента вы увидите сумму двух переданных службе чисел. Еще раз обратите внимание на то, что конкретная реализация метода Add() клиенту неизвестна. Но поскольку клиент обратился к службе по правилам, описанным в контракте, служба запустила для него метод Add() и вернула результат.

5. Конфигурирование WCF-приложений с помощью файла конфигурации

Мы создали пару приложений, демонстрирующих базовые принципы технологии WCF. Я полагаю, вы все прекрасно понимаете, что серверная часть нашего проекта не является Windows-службой. Для Windows наш сервер – обычное консольное приложение. Но мы знаем, что в этом консольном приложении хостится WCF-служба. Давайте определим два основных недостатка наших службы и клиента. Я думаю, вы согласитесь с тем, что недостаток службы – это строка, в которой вызывается метод `AddServiceEndpoint()`. Обратите внимание, в этой строке явно указаны адрес, привязка и контракт оконечной точки службы. Что в этом плохого? Если мы захотим изменить привязку или адрес оконечной точки, нам придется изменять код и перекомпилировать службу. Если говорить о недостатках нашего клиента, то главный его недостаток – это тот способ, с помощью которого мы сообщили клиенту о контракте службы. Клиент просто обязан иметь другой механизм, позволяющий ему узнать контракт службы.

Давайте исправим эти недостатки. Вы уже встречались ранее с файлами конфигурации приложения. Вспомните курс ADO.NET и класс ProviderFactory. Файл конфигурации приложения позволяет описать некоторые настройки вне тела приложения. Приложение после запуска будет считывать необходимые значения из этого файла. Для изменения таких настроек не надо будет перекомпилировать приложение, достаточно просто изменить конфигурационный файл.

Аналогичный механизм реализован и в технологии WCF, где широко применяются конфигурационные файлы. Конфигурационный файл представляет собой XML-файл с именем App.config. Корневой элемент этого файла – <configuration>. В него включаются другие элементы, описывающие те или иные параметры приложения. Нас будет интересовать элемент <system.ServiceModel>, в котором расположены несколько основных разделов.

Элемент <services> реализует раздел служб, размещенных в приложении. Он включает в себя элементы <service>, каждый из которых описывает службу. Для описания оконечных точек службы используются элементы <endpoint>, располагаемые внутри элементов <service>. Элемент <endpoint> имеет такие основные атрибуты:

- address – задает адрес оконечной точки, либо как абсолютный адрес, либо относительно базового адреса службы. Если это значение не указано, то адрес оконечной точки – это базовый адрес службы;
- binding – задает тип привязки;
- contract – задает интерфейс службы, определяющий контракт. Этот интерфейс должен быть реализован в службе с именем, которое задано атрибутом name элемента service.

Следующий важный раздел содержит спецификации привязок для оконечных точек службы. Этот раздел описывается элементом <bindings>, в котором содержатся элементы <binding>.

Еще один раздел содержит описания поведений службы. Этот раздел описывается контейнером <behaviors>, в котором располагаются элементы <behavior>.

Со всеми этими разделами мы ознакомимся в ходе нашего разговора. Пока отметьте, что не обязательно все разделы должны содержаться в конфигурационном файле. Ниже приведен пример файла App.config, в котором определен только раздел <services>:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <services>
            <service name="Module1Service.MathService">
                <endpoint address="http://localhost/MathService"
                        binding="basicHttpBinding"
                        bindingConfiguration="" contract="Module1Service.IMath" />
            </service>
        </services>
    </system.serviceModel>
</configuration>
```

а. Пример службы с конфигурационным файлом

Сейчас мы научимся создавать конфигурационный файл для WCF службы. Давайте создадим новый проект консольного приложения:

1. Запустим Visual Studio и начнем новый консольный проект с именем WCF_MyServiceConfigFile;
2. Добавим ссылку на пространство имен `System.ServiceModel;`
3. В пространстве имен WCF_MyServiceConfigFile создадим класс службы MathService, в котором определим метод Add(), а контракт опишем в интерфейсе:

```
[ServiceContract]
public interface IMyMath
```

```
{
        [OperationContract]
        int Add(int a, int b);
}

public class MathService: IMyMath
{
        public int Add(int a, int b)
        {
                return a + b;
        }
}
```

4. В методе Main() необходимо реализовать среду выполнения для класса MathService. Для этих целей служит класс ServiceHost.

```
ServiceHost sh = new ServiceHost(typeof(MathService));
```

5. Активизируем среду выполнения:

```
sh.Open();
```

6. Наша служба после запуска должна оставаться активной, чтобы принимать подключения клиентов. Реализуем это простым способом, предусмотрев и завершение службы:

```
Console.WriteLine("Для завершения нажмите <ENTER>\n");
Console.ReadLine();
sh.Close();
```

Вот и все, готовый код сервера должен выглядеть следующим образом:

Пример 4. Служба с конфигурационным файлом

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;

namespace WCF_MyServiceConfigFile
{
        [ServiceContract]
        public interface IMyMath
        {
                [OperationContract]
                int Add(int a, int b);
        }

        public class MathService: IMyMath
        {
                public int Add(int a, int b)
                {
                        return a + b;
                }
        }

        class Program
        {
          static void Main(string[] args)
          {
              ServiceHost sh = new ServiceHost(typeof(MathService));
              sh.Open();
              Console.WriteLine("Для завершения нажмите <ENTER>\n");
              Console.ReadLine();
```

```
            sh.Close();
        }
    }
}
```

Как видите, этот вариант отличается от предыдущего отсутствием вызова метода `AddServiceEndpoint()`. Вы понимаете, что сейчас мы укажем все настройки для оконечной точки в конфигурационном файле. Но не ищите в коде ссылок на этот файл. Их там нет. Дело в том, что при вызове метода `ServiceHost.Open()` WCF автоматически ищет конфигурационный файл и, если он есть, читает из него необходимые настройки. В Visual Studio 2012 конфигурационный файл присутствует в приложении изначально. В случае использования более ранних версий Visual Studio конфигурационный файл надо добавить вручную. Это можно сделать так:

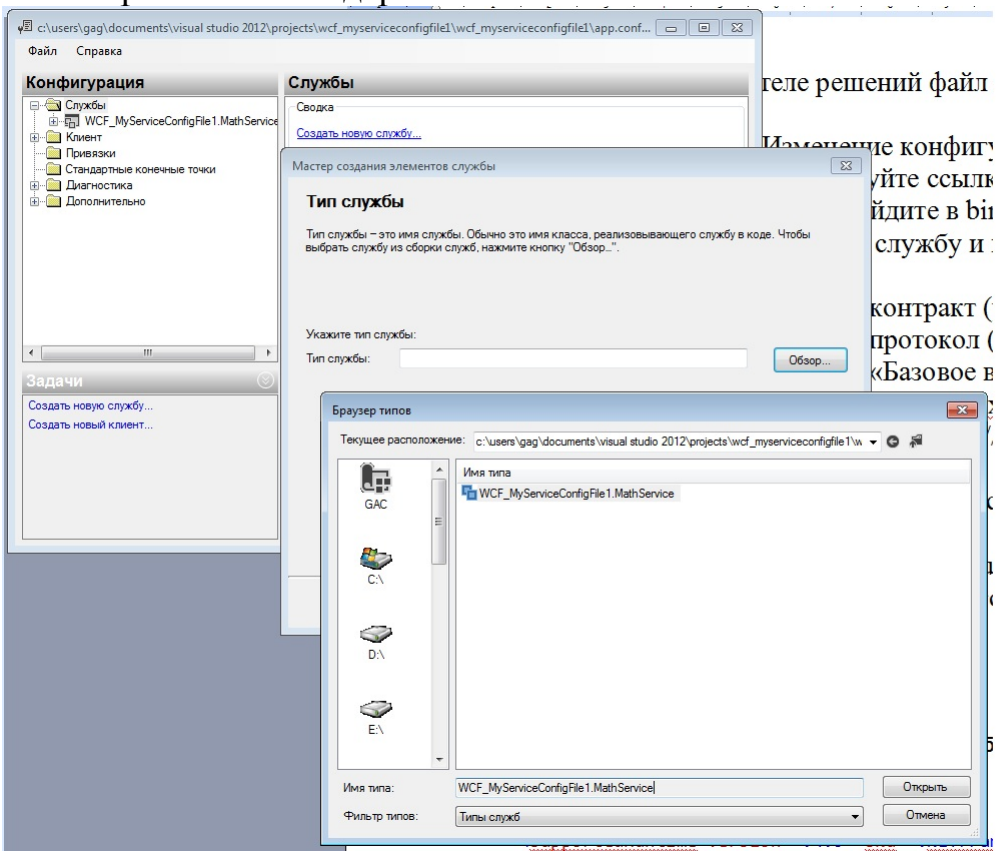
1. Откройте обозреватель решений;
2. Выделите имя пространства имен своего проекта и нажмите правую кнопку мыши;
3. Выберите команду «Add»;
4. Выберите команду «Create»;
5. Выберите из появившегося окна опцию «Application configuration file» и нажмите «Add».

В составе проекта появится файл с именем App.config.

Изначально App.config представляет собой пустой XML-файл. Для заполнения конфигурационного файла можно разобраться с его структурой и изменять файл вручную. Однако удобнее пользоваться специальным редактором – WCF Service Configuration Editor. Чтобы этот редактор можно было вызывать из контекстного меню, надо один раз выполнить в Visual Studio следующее действие:

6. Выберите в Visual Studio меню «Service»;
7. В раскрывшемся меню выберите пункт «Редактор конфигураций службы WCF» или «WCF Service Configuration Editor»;
8. Когда раскроется окно Microsoft Service Configuration Editor, просто закройте его. Теперь вызов этого редактора будет доступен из контекстного меню;
9. Выделите в обозревателе решений файл App.config и нажмите правую кнопку мыши;
10. Выберите команду «Изменение конфигурации WCF»;
11. В появившемся окне активируйте ссылку Create a New Service;
12. Нажмите кнопку Browse перейдите в bin\Debug вашего проекта, выберите .exe-файл проекта, затем выберите службу и нажмите Open;
13. Нажмите Next;
14. В следующем окне выберите контракт (у нас он один) и нажите Next;
15. В следующем окне выберите протокол (HTTP) и нажите Next;

16. В следующем окне выберите «Базовое взаимодействие с веб-службами» Basic Web Service Interoperability и нажите Next;
17. В следующем окне введите адрес (http://localhost/MathService) и нажите Next;
18. В следующем окне нажмите Finish;
19. Зайдите в меню «Файл» Microsoft Service Configuration Editor и выполните команду Save (или нажмите Ctrl+S);
20. Закройте окно Microsoft Service Configuration Editor;
21. Откройте файл App.config (если появится вопрос о сохранении – скажите «Да») и посмотрите на его содержимое.



App.config должен выглядеть так:

Пример 5. Простейший конфигурационный файл службы

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
    <system.serviceModel>
        <services>
            <service name="WCF_MyServiceConfigFile.MathService">
                <endpoint address="http://localhost/MathService" binding="basicHttpBinding"
                    bindingConfiguration="" contract="WCF_MyServiceConfigFile.IMyMath" />
            </service>
        </services>
    </system.serviceModel>
</configuration>
```

Запустите службу и убедитесь в том, что она работает.

Итак, что мы сделали? Мы создали специальный файл конфигурации приложения, в который вынесли настройки оконечной точки. Теперь этот файл можно редактировать с помощью «Блокнота» или любого другого текстового редактора (ведь это текстовый файл). Служба будет читать из него необходимые значения.

b. Конфигурационный файл – смотрим внимательнее

Посмотрим что еще можно определить в конфигурационном файле.

i. оконечные точки в конфигурационном файле

В нашем примере мы описывали оконечные точки службы при вызове метода AddServiceEndpoint() класса ServiceHost:

```
sh.AddServiceEndpoint(
      typeof(IMyMath), //контракт
      new WSHttpBinding(), //тип привязки
      "http://localhost/MyMath/Ep1"); //адрес и имя оконечной точки
```

Однако описание оконечных точек службы в конфигурационном файле является более удобным приемом. В этом случае изменение характеристик службы не требует перекомпиляции службы, что делает использование службы более гибким. Давайте посмотрим, как надо описывать оконечные точки в конфигурационном файле.

```
<system.serviceModel>
<services>
      <service name = "Module1Service.MathService">
            <endpoint
                  address = "http://localhost:8000/MathService"
                  binding = "wsHttpBinding"
                  contract = "Module1Service.IMath"
            />
      </service>
</services>
</system.serviceModel>
```

Отметьте себе, что при указании имени службы и имени контракта необходимо указывать пространство имен – в нашем случае Module1Service.MathService и Module1Service.IMath. Мы уже упоминали, что одна служба может представлять внешнему миру несколько оконечных точек. Конфигурационный файл легко позволяет описать несколько оконечных точек службы. Например, так как это показано в следующем фрагменте конфигурационного файла:

```
<system.serviceModel>
<services>
      <service name = "Module1Service.MathService">
            <endpoint
                  address = "http://localhost:8000/MathService"
                  binding = "wsHttpBinding"
                  contract = "Module1Service.IMath"
            />
            <endpoint
                  address = "net.tcp://localhost:8001/MathService"
```

```
                binding = "netTcpBinding"
                contract = "Module1Service.IContract"
            />
            <endpoint
                address = "net.tcp://localhost:8002/MathService"
                binding = "netTcpBinding"
                contract = "Module1Service.IAnotherContract"
            />
        </service>
    </services>
    </system.serviceModel>
```

Обращайте внимание на тот момент, что если для оконечной точки указан базовый адрес ([transport]://[machinename or domainname][: port]), то протокол в адресе должен согласовываться с типом привязки. Вы должны помнить, что разные привязки используют определенные протоколы. Другими словами, если, например, адрес оконечной точки начинается с «net.tcp://», то привязка должна быть netTcpBinding, но никак не wsHttpBinding.

WCF позволяет разным оконечным точкам иметь один и тот же базовый адрес, но в этом случае URI этих оконечных точек обязательно должны различаться:

```
    <system.serviceModel>
    <services>
        <service name = "Module1Service.MathService">

            <endpoint
                address = "net.tcp://localhost:8001/MathService"
                binding = "netTcpBinding"
                contract = "Module1Service.IMath "
            />
            <endpoint
                address = "net.tcp://localhost:8001/SomeOtherService"
                binding = "netTcpBinding"
                contract = "Module1Service.IAnotherContract"
            />
        </service>
    </services>
    </system.serviceModel>
```

Как это ни странно звучит, но служба может не определять ни одной оконечной точки ни в конфигурационном файле, ни в коде. «И что, такая служба будет работать?» – должны спросить вы. Да, она будет работать, но только в том случае, если задан хотя бы один базовый адрес. При наличии базового адреса и при отсутствии явного определения оконечных точек, WCF самостоятельно создаст оконечную точку для каждого заданного базового адреса. Это будут оконечные точки по умолчанию. В этом случае адресом оконечной точки по умолчанию будет базовый адрес, а привязка будет создана тоже автоматически, исходя из базового адреса. При создании привязки для протоколов TCP, IPC и MSMQ будут использоваться соответственно привязки netTcpBinding, netNamedPipeBinding и netMsmqBinding. При создании привязки для случая использования HTTP или HTTPS протокола, по умолчанию WCF создаст

привязку basicHttpBinding. Если же вы захотите использовать wsHttpBinding привязку, вам придется позаботиться об этом особо. Для этого надо добавить в конфигурационный файл секцию <protocolMapping>:

```
<system.serviceModel>
      <protocolMapping>
            <add scheme = "http" binding = "wsHttpBinding" />
      </protocolMapping>
</system.serviceModel>
```

Особо отметьте, что такое переназначение привязки можно сделать только в конфигурационном файле. Программно это сделать невозможно. Будет еще полезно отметить, что имя оконечной точки по умолчанию представляет собой конкатенацию названия привязки, символа нижнего подчеркивания и имени контракта, например – "netTcpBinding_IMath ".

ii. привязка в конфигурационном файле

Теперь пришел черед рассмотреть, каким образом конфигурационный файл позволяет настраивать привязки. Для этого в элементе <endpoint> необходимо указать атрибут bindingConfiguration и присвоить ему какое-либо имя. Затем создать в секции <bindings> элемент <binding> с таким именем, как значение атрибута bindingConfiguration:

Пример 6. Системная привязка в конфигурационном файле службы

```
<system.serviceModel>
  <services>
    <service name = "Module1Service.MathService">
      <endpoint
          address = "net.tcp://localhost:8001/MathService"
          bindingConfiguration = "TCPwithTransaction"
          binding = "netTcpBinding"
          contract = "Module1Service.IContract"
      />
      <endpoint
          address = "net.tcp://localhost:8002/MathService"
          bindingConfiguration = "TCPwithTransaction"
          binding = "netTcpBinding"
          contract = "Module1Service.IAnotherContract"
      />
    </service>
  </services>
  <bindings>
    <netTcpBinding>
      <binding name = "TCPwithTransaction" transactionFlow = "true" />
    </netTcpBinding>
  </bindings>
</system.serviceModel>
```

В этом примере мы создали привязку, допускающую использование транзакций. Но вы должны в этом примере обратить внимание не на специфику созданной привязки, а

на другой момент. На то, что конфигурационный файл позволяет каждой оконечной точке ссылаться на свой вариант привязки.

Также можно создать, так называемую привязку по умолчанию, которая будет распространяться на все оконечные точки службы. Например, такая неименованная привязка:

```
<netTcpBinding>
      <binding transactionFlow = "true" />
</netTcpBinding>
```

будет воздействовать на все оконечные точки, использующие тип привязки netTcpBinding и не ссылающиеся явно на какую-либо именованную привязку (не содержащие атрибута bindingConfiguration). Для каждого типа привязки (wsHttpBinding, netTcpBinding и т.п.) можно задать только одну привязку по умолчанию. Однако следует отметить, что использование именованных привязок совместно с привязками по умолчанию является нежелательным, так как сильно усложняет понимание конфигурационного файла. Желательно придерживаться какой-либо одной линии: либо использовать только именованные привязки, либо – привязки по умолчанию.

Такие же действия можно выполнить и в коде, не используя конфигурационный файл:

```
ServiceHost sh = new ServiceHost(typeof(IMath));
NetTcpBinding tcpBinding = new NetTcpBinding();
tcpBinding.TransactionFlow = true;
sh.AddServiceEndpoint(typeof(IMath),
      tcpBinding,
      "net.tcp://localhost:8000/MathService");
```

Однако такой способ является менее гибким по сравнению с применением конфигурационного файла.

iii. обеспечение контроля доставки в конфигурационном файле

Со временем вы привыкнете к структуре конфигурационнго файла и найдете его удобным средством настройки многих опций служб и клиентов. А сейчас я просто приведу пример конфигурационного файла, в котором привязка сконфигурирована таким образом, чтобы обеспечивать контроль доставки сообщений. Мы говорили об этом немного ранее в этом уроке.

```
<system.serviceModel>
      <services>
      <service name = "MyService">
            <endpoint
                  address = "net.tcp://localhost:8000/MyService"
                  binding = "netTcpBinding"
                  bindingConfiguration = "ReliableTCP"
                  contract = "IMyContract"
            />
      </service>
      </services>
```

```
            <bindings>
            <netTcpBinding>
                  <binding name = "ReliableTCP">
                        <reliableSession enabled = "true"/>
                  </binding>
            </netTcpBinding>
            </bindings>
      </system.serviceModel>
```

iv. создание пользовательской привязки в конфигурационном файле

Как вы успели заметить конфигурационный файл является хорошей альтернативой кодированию для реализации очень многих возможностей приложений WCF. Давайте вспомним о том, что мы можем создавать собственные привязки, если среди системной привязки не нашли для себя ничего подходящего. Раньше мы рассмотрели, как это можно делать в коде, а теперь посмотрим, как это можно сделать в конфигурационном файле:

Пример 7. Пользовательская привязка в конфигурационном файле службы

```
<system.serviceModel>
  <services>
    <service name="MySrvice">
      <host>
        <baseAddresses>
          <add baseAddress="net.tcp://localhost/MyService" />
        </baseAddresses>
      </host>
      <endpoint address=""
      contract="MathService.IMyService"
      binding="customBinding"
      bindingConfiguration="customBinding"/>
    </service>
  </services>
  <bindings>
    <customBinding>
      <binding name="customBinding">
        <textMessageEncoding />
        <httpsTransport />
      </binding>
    </customBinding>
  </bindings>
</system.serviceModel>
```

Для этого надо реализовать секцию <customBinding>, а затем сослаться на нее в описании соответствующей оконечной точки. В примере выше, соответствующие строки конфигурационного файла выделены жирным шрифтом.

Новые термины и понятия

Конфигурационный файл – XML файл, содержащий информацию, считываемую приложением во время выполнения.

6. Проектирование контрактов

Сейчас будет уместным немного поговорить о том, каким образом создавать в службе контракты, и каким образом реализовывать операции в контрактах. Речь будет идти больше о количественных характеристиках. Я думаю всем понятно, что контракт должен объединять в себе операции, некоторым образом связанные логически, реализующие какую-либо поведенческую черту службы. Но кроме этого еще необходимо принимать в расчет некоторые другие факторы.

Обычно подход должен быть приблизительно таким. Сначала вы должны определить полный перечень всех операций, которые должна реализовать ваша служба. Затем надо подумать о том, на какое количество контрактов эти операции надо разделить, и каким именно образом разделять операции по контрактам. Старайтесь делать контракты такими, чтобы они зависели друг от друга минимальным образом. Пусть каждый контракт объединяет в себе операции, имеющие отношение к одной какой-либо функциональности службы. Старайтесь всегда (не только при использовании WCF) чтобы ваши приложения были масштабируемыми. Т.е., чтобы они состояли из частей, мало пересекающихся друг с другом и допускающих повторное использование. Такое приложение легче сопровождать и, например, добавление к такому приложению новой функциональности, вряд ли приведет к необходимости переделывать уже реализованную структуру приложения.

Давайте представим, что мы проектируем WCF-службу предоставляющую клиентам функциональность калькулятора. Это может быть как калькулятор для школьников, когда достаточно лишь основных математических операций, так и научный калькулятор, позволяющий возводить в степень, работать с логарифмами и т.п. Но и это еще не все. Пусть наша служба будет вести статистику обращений к себе клиентов. Т.е. при поступлении запроса от каждого клиента, служба будет где-то сохранять время прихода этого запроса. А когда мы попросим службу рассказать нам об истории запросов, она каким-то образом пришлет нам статистику вызовов за определенный период.

Вставлять все эти операции в один контракт мы не будем. В нашем случае напрашивается создание трех контрактов. Например, таких:

```
[ServiceContract]
public interface ISchoolCalc            //контракт школьного калькулятора
{
       [OperationContract]        //операция сложения
       int Add(int a, int b);
       [OperationContract]        //операция вычитания
       int Sub(int a, int b);
       [OperationContract]        //операция деления
       int Div(int a, int b);
       [OperationContract]        //операция умножения
       int Mult(int a, int b);
       [OperationContract]        //операция деления по модулю
       int Mod(int a, int b);
```

```
}

[ServiceContract]
public interface IScienceCalc    //контракт научного калькулятора
{
       [OperationContract]       //возведение в степень
       int Pow(int num, int exp);
       [OperationContract]       //вычисление десятичного логарифма
       double Log10(double num);
       [OperationContract]       //вычисление натурального логарифма
       double Log(double num);
       . . .                     //вычисление чего-то там еще
}

[ServiceContract]
public interface IStat           //контракт для учета статистики
{
       [OperationContract]       //определение обращений клиентов за период от p1 до p2
       DateTime[] GetStat(DateTime p1, DateTime p2);
}
```

Слышу ваши замечания о том, что контракт с одной лишь операцией, наверное, не является хорошим примером проектирования. Вообще-то говоря, да. Но существование таких контрактов вполне допустимо. И кроме этого, наверняка в контракте IStat могут находиться еще другие операции, имеющие отношения к статистике использования нашей службы. О точном числе операций в контракте мы поговорим через несколько строк. И еще – не вникайте сейчас глубоко в прототипы указанных операций. Это всего лишь иллюстрация.

Теперь мы можем реализовать нашу службу-калькулятор, например, таким образом:

```
public class SchoolCalculator: ISchoolCalc
{. . .}
```

Такой калькулятор будет выполнять вычисления, необходимые школьнику. Следующий калькулятор будет выполнять вычисления, необходимые школьнику и кроме этого будет уметь сообщать статистику своей работы:

```
public class SchoolCalculator: ISchoolCalc, IStat
{. . .}
```

А вот такой калькулятор:

```
public class ScienceCalculator: ISchoolCalc, IScienceCalc, IStat
{. . .}
```

будет выполнять вычисления, необходимые и школьнику и научному работнику и кроме этого также будет уметь сообщать статистику своей работы.
Чем в данном случае полезно создание отдельного контракта IStat? Тем, что этот контракт может понадобиться не только калькуляторам, а и другим функциональностям службы. В нашем случае никаких таких других функциональностей нет. Но не исключено, что завтра заказчик попросит добавить что-нибудь ☺

А теперь познакомьтесь с еще одним полезным свойством WCF: контракты службы могут наследовать друг другу. Это позволяет нам сконструировать контракты службы таким образом:

```
[ServiceContract]
public interface ISchoolCalc            //контракт школьного калькулятора
{
       [OperationContract]        //операция сложения
       int Add(int a, int b);
       [OperationContract]        //операция вычитания
       int Sub(int a, int b);
       [OperationContract]        //операция деления
       int Div(int a, int b);
       [OperationContract]        //операция умножения
       int Mult(int a, int b);
       [OperationContract]        //операция деления по модулю
       int Mod(int a, int b);
}

[ServiceContract]
public interface IScienceCalc :  ISchoolCalc //контракт научного калькулятора
{
       [OperationContract]        //возведение в степень
       int Pow(int num, int exp);
       [OperationContract]        //вычисление десятичного логарифма
       double Log10(double num);
       [OperationContract]        //вычисление натурального логарифма
       double Log(double num);
       . . .                      //вычисление чего-то там еще
}

[ServiceContract]
public interface IStat             //контракт для учета статистики
{
       [OperationContract]        //определение обращений клиентов за период от p1 до p2
       DateTime[] Add(DateTime p1, DateTime p2);
}
```

Обратите внимание, что IScienceCalc наследует ISchoolCalc, но атрибут [ServiceContract] должен указываться у каждого контракта. Теперь создание «научного» калькулятора будет выглядеть проще, т.к. не надо явно указывать наследование ISchoolCalc, оно подразумевается автоматически:

```
public class ScienceCalculator: IScienceCalc, IStat
{. . .}
```

А теперь пару слов о точных цифрах. Большинство разработчиков сходятся во мнении, что оптимальное число операций для одного контракта – от трех до пяти. Нормой считается и число операций до девяти. Но вот большее число операций в одном контракте не рекомендуется. И уже категорически не рекомендуется создавать контракты с двадцатью операциями (и тем более с большим количеством). Такие контракты должны обязательно разбиваться на несколько меньших контрактов. Но вместе с этим, вы должны понимать, что большое количество маленьких контрактов тоже не есть хорошо. Надо соблюдать баланс между количеством контрактов и их

размерами. Как говорили древние эллины «métron áriston», что значит «Наивысшая благодетель – это чувство меры». А древние эллины были людьми умными. Наверное, даже умнее нас. Но это мое субъективное мнение, к тому же, не имеющее никакого отношения к WCF.

Домашнее задание

Создайте службу с именем MyDiskInfo, которая будет передавать клиентам некоторую информацию о дисках своего компьютера. В службе надо реализовать две операции с именами FreeSpace() и TotalSpace(). Каждая из операций должна принимать параметр типа string и возвращать результат типа string.

Во входном параметре операции должно передаваться имя диска. При этом, если в переданной строке указано больше одного символа – учитываться должен только первый символ. Например, если какой-либо операции клиент передаст строку “disk” – значит работать надо будет с диском d, а если в строке будет “apple” – то работать надо будет с диском a.

В возвращаемом результате FreeSpace() должна возвращать в строке объем в байтах свободного места на указанном диске того компьютера, где хостится сама служба, а TotalSpace() – общий объем в байтах места на указанном диске. Если в FreeSpace() или TotalSpace() передано имя несуществующего диска, в возвращаемой строке должно содержаться что-то типа “Wrong disk!”.

Служба должна содержать конфигурационный файл, описывать в коде mex-точку (конечную точку обмена метаданными) и хоститься в консольном приложении.

Создайте для этой службы два клиента:

- один клиент должен общаться со службой по каналам, созданным с помощью ChannelFactory;
- второй клиент должен использовать прокси-класс;

Проверьте возможность доступа к службе обоих клиентов одновременно. Оба клиента также как и служба должны быть консольными приложениями;