Урок 2

1. Понятие прокси-класса ... 1
a. Публикация метаданных службы (mex-точка) ... 2
Пример 8. Конфигурационный файл, раскрывающий MEX-точку ... 5
i. MEX-точки – смотрим внимательнее ... 6
b. Пример клиента, использующего прокси-класс ... 7
Пример 9. Клиент, использующий прокси-класс ... 8
i. использование прокси-класса – смотрим внимательнее ... 9
c. Поведения службы ... 11
Новые термины и понятия ... 12
2. Контракт данных ... 12
Пример 10. Служба, требующая контракт данных ... 13
a. Контракт данных – смотрим внимательнее ... 16
b. Перегрузка операций ... 19
3. Асинхронный клиент ... 22
Пример 11. Клиент, вызывающий службу асинхронно ... 23
4. Другие способы хостинга службы ... 24
a. Библиотека службы WCF ... 24
b. Хостинг WCF службы в Windows службе ... 26
c. Хостинг WCF-службы в IIS ... 29
d. Хостинг WCF-службы в WAS ... 30
5. Односторонние операции ... 31
Пример 12. Клиент и служба с односторонними методами ... 31
6. Дуплексные операции ... 33
Пример 13. Служба и клиент с дуплексным контрактом ... 35
Домашнее задание ... 38

1. Понятие прокси-класса

Давайте вспомним приложение, которое мы создали на прошлом уроке. В этом приложении клиент обращается к службе с просьбой сложить два целочисленных значения и прислать результат суммирования. Клиент и служба общаются друг с другом с помощью каналов. Каналы создавались объектом класса ChannelFactory<>. Однако, это не единственный способ общения службы и клиента, и не всегда самый удобный. Вместо явного использования класса ChannelFactory<>, на стороне клиента можно создать некоего посредника, прокси-класс, который будет являться образом контракта службы. В этом случае общение клиента со службой будет осуществляться через этот прокси-класс. Использование прокси-класса является более простым способом наладить общение между клиентами и службой. Дело в том, что прокси-класс позволяет нам работать с классом службы, как будто бы с локальным классом, расположенным рядом с нами, а не где-то «там далеко».

Прокси-класс можно создать с помощью утилиты svcutil.exe, поставляемой вместе с .NET Framework, явным образом запустив эту утилиту в режиме командной строки и указав соответствующие ключи. Можно также использовать Visual Studio,

предоставляющую для этого более удобный механизм. Именно так мы и поступим чуть позже.

Чтобы клиент мог создать прокси-класс, он должен знать контракт оконечной точки, к которой обращается. Как вы должны помнить, класс ServiceHost автоматически формирует полное описание службы в своем свойстве Description, представляющем собой объект класса ServiceDescription. Это описание представлено в формате WSDL (Windows Service Description Language). Visual Studio совместно с очень полезной утилитой svcutil.exe, может прочитать WSDL-описание службы, содержащееся в свойстве Description объекта ServiceHost, и на его основании построить прокси-класс при проектировании клиента. Однако необходимо, чтобы служба «позволяла» читать свое описание в свойстве Description объекта ServiceHost.

В нашем клиенте мы явно прописали интерфейс с контрактом. Но, согласитесь, далеко не всегда клиенту известен контракт службы, с которой он хочет общаться. Как же быть в том случае, когда надо сотрудничать со службой, а ее контракт неизвестен? Как ни странно, но сама служба может рассказать клиентам о своих контрактах (если такое ее поведение было предусмотрено при создании службы).

Помните, что служба имеет оконечные точки? Они необходимы для общения с клиентами. Однако среди оконечных точек службы может быть одна специальная оконечная точка с именем mex (Metadata Exchange – обмен метаданными). Эта точка предназначается для того, чтобы рассказать клиентам, как следует обращаться к службе. Сейчас нам надо научиться создавать mex-точку, так как по умолчанию службы не раскрывают свои метаданные.

а. Публикация метаданных службы (mex-точка)

Для раскрытия метаданных службы надо в этой службе создать оконечную mex-точку. Метаданные службы можно раскрыть как в коде, так и в конфигурационном файле. Правило такое: если оконечные точки описаны в коде, то и mex-точку надо описать в коде службы, а если оконечные точки описаны в конфигурационном файле, то и mex-точку надо описать в конфигурационном файле.

Кратко рассмотрим описание метаданных в коде, а затем создадим mex-точку в конфигурационном файле. При добавлении mex-точки в коде сначала службе надо добавить поведение, включающее контракт для mex-точки. Имя контракта – IMetadataExchange. Это предопределенный контракт, реализованный разработчиками WCF. Нам остается только с благодарностью пользоваться им. Поведение добавляется с помощью объекта класса ServiceMetadataBehavior. Следующий код можно добавить в службе перед вызовом метода `ServiceHost.Open()`:

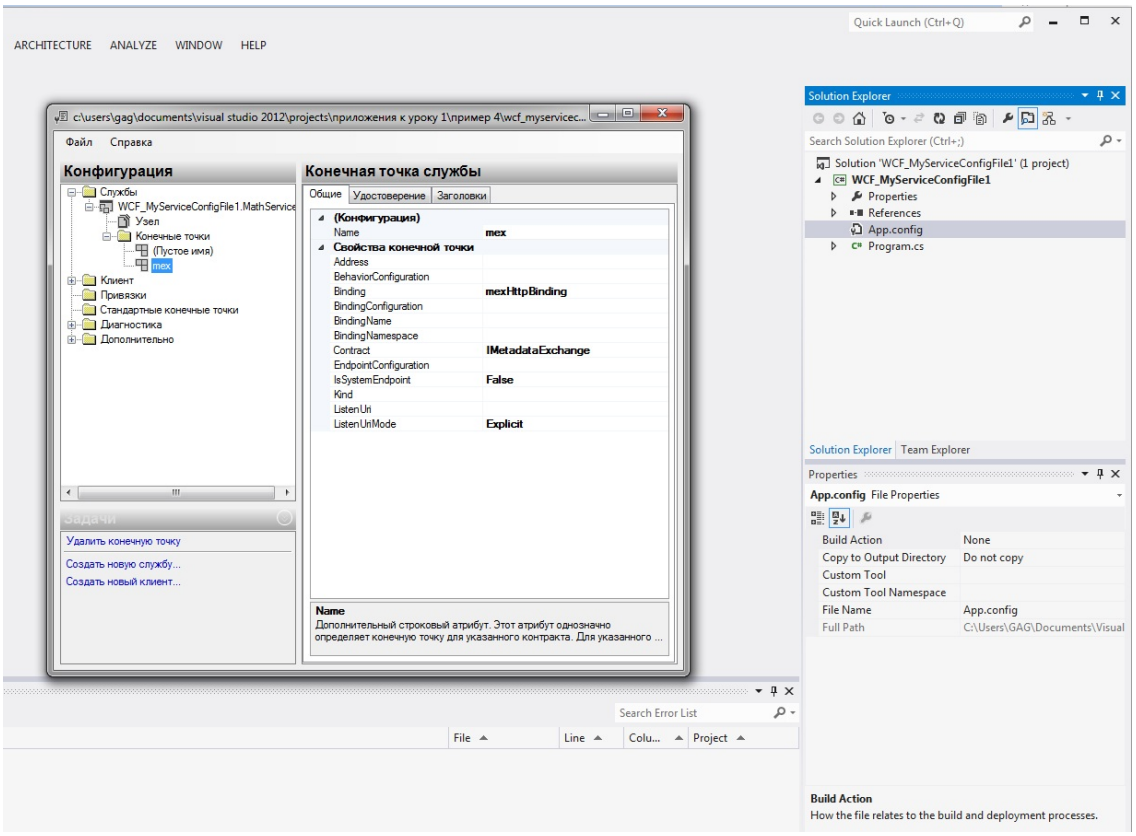
```
//создать новый объект поведения;
ServiceMetadataBehavior behavior = new ServiceMetadataBehavior();
```

```
//разрешить доступ к метаданным по протоколу Http;
behavior.HttpGetEnabled = true;

//добавить новое поведение в описание службы;
serviceHost.Description.Behaviors.Add(behavior);

//добавить оконечную mex-точку;
serviceHost.AddServiceEndpoint(
      typeof(IMetadataExchange),
      MetadataExchangeBindings.CreateMexHttpBinding(),
      "mex");
```

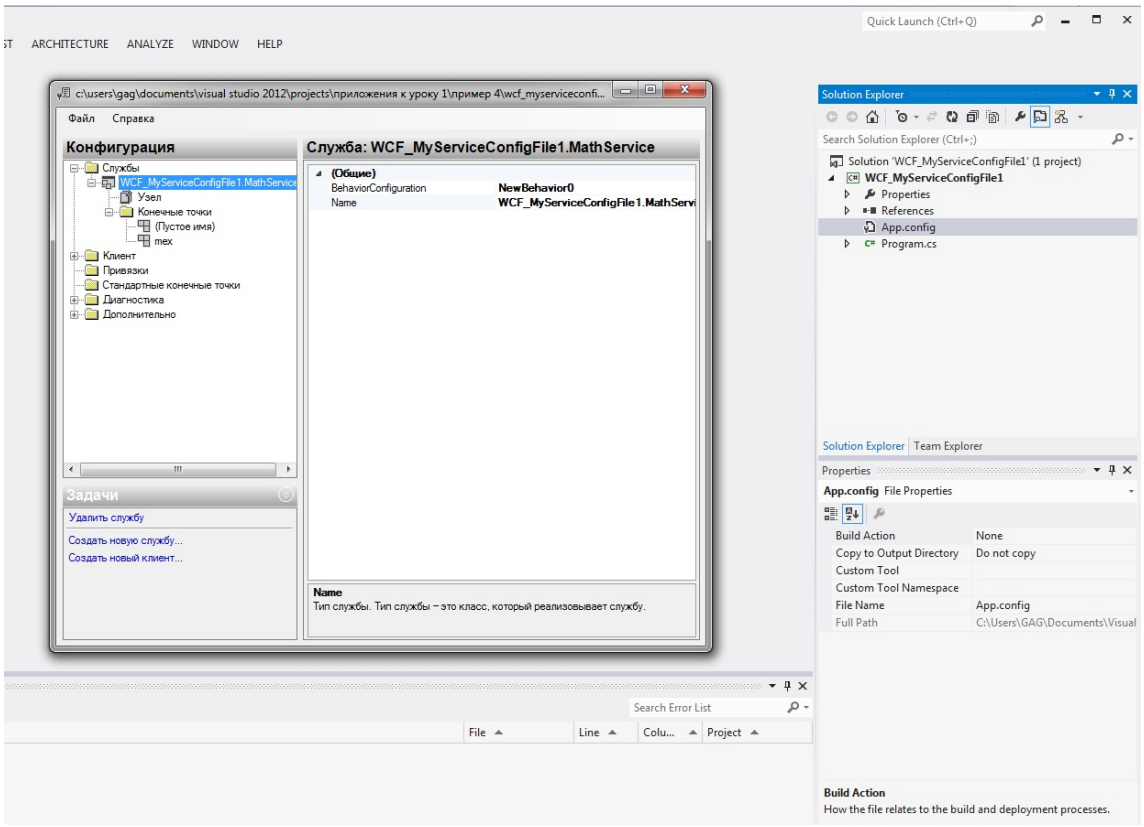
При выполнении этого кода в службе будет создана mex-точка, позволяющая клиентам видеть метаданные службы. При самостоятельной работе попробуйте создать для нашей службы mex-точку описанным здесь способом.

Мы этого делать не будем. Рассмотрим добавление mex-точки в конфигурационном файле. Чтобы опубликовать метаданные службы, надо выполнить следующие действия:

1. Запустите WCF Configuration Editor для App.config;
2. Раскройте узел Services и дойдите до узла Конечные точки (Endpoints);
3. Раскройте узел оконечной точки с именем Пустое имя (Empty Name);
4. Для свойства Address укажите значение http://localhost/MathService/ep1, где ep1 – имя оконечной точки, а http://localhost/MathService/ – базовый адрес службы;
5. Нажатием правой кнопки мыши на узле узла Endpoints добавьте новую Endpoint;
6. Укажите для свойства Name новой конечной точки значение mex;
7. Укажите для свойства Binding новой конечной точки значение mexHttpBinding;
8. Укажите для свойства Contract новой конечной точки значение IMetadataExchange;



9. После этого раскройте узел Advanced (Дополнительно) и в нем – узел Service Behaviours (Поведения служб);
10. В правой части окна кликните по ссылке New Service Behaviours Configuration (Создать конфигурацию поведения службы);
11. В следующем окне нажмите кнопку Add и выберите из появившегося списка значение ServiceMetadata;
12. После этого раскройте появившийся узел ServiceMetadata и в поле HttpGetEnabled установите значение true;
13. После этого надо создать базовый адрес – для этого выделите узел Host (Узел) нашей службы;
14. В правой части окна в поле Base Address нажмите кнопку New (если там уже был занесен какой-либо адрес, сначала удалите его кнопкой Delete);
15. После этого в узле службы добавьте созданное Behaviour (Поведение), для чего выделите узел с именем нашей службы и в поле BehaviourConfiguration выберите имя созданного вами поведения (по умолчанию – NewBehaviour0).



После выполнения указанных действий файл App.config примет вид:

Пример 8. Конфигурационный файл, раскрывающий MEX-точку

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
    <system.serviceModel>
        <behaviors>
            <serviceBehaviors>
                <behavior name="NewBehavior0">
                    <serviceMetadata httpGetEnabled="true" />
                </behavior>
            </serviceBehaviors>
        </behaviors>
        <services>
             <service behaviorConfiguration="NewBehavior0"
                    name="WCF_MyServiceConfigFile1.MathService">
                <endpoint address="http://localhost/MathService/ep1" binding="basicHttpBinding"
                    bindingConfiguration="" contract="WCF_MyServiceConfigFile1.IMyMath" />
                <endpoint binding="mexHttpBinding" bindingConfiguration="" name="mex"
                    contract="IMetadataExchange" />
                <host>
                    <baseAddresses>
                        <add baseAddress=" http://localhost/MathService" />
                    </baseAddresses>
                </host>
            </service>
        </services>
```

```
    </system.serviceModel>
</configuration>
```

Отличия от предыдущего варианта выделены жирным шрифтом. Вы помните, что мы устраняем недостаток клиента, а именно – тот способ, которым клиент узнает о контракте службы. Только что мы изменили именно службу. Наши изменения привели к тому, что служба через mex-точку открыла внешнему миру свои метаданные. Другими словами, теперь любой клиент, который пожелает общаться с нашей службой, сможет получить информацию о контрактах службы без всяких «шпионских штучек». Вполне легально с помощью самой службы. Теперь перейдем к клиенту.

i. MEX-точки – смотрим внимательнее

Вы должны помнить, что мы добавляли MEX-точки, как и другие оконечные точки, с помощью метода AddServiceEndpoint(), которому необходимо передавать тип контракта, привязку и адрес оконечной точки. Привязку для MEX-точки мы должны конструировать самостоятельно, в зависимости от используемого транспортного протокола. Для этого в нашем распоряжении есть статический класс MetadataExchangeBindings с тремя статическими методами: CreateMexHttpBinding() , CreateMexNamedPipeBinding() и CreateMexTcpBinding(). Имена методов говорят сами за себя. Каждых из них возвращает созданный объект привязки Binding, который можно отдавать методу AddServiceEndpoint().

Давайте еще раз посмотрим на создание MEX-точки в коде и немного оптимизируем этот код, сделав его безопаснее. Для выполнения этого кода необходимо подключить пространство имен System.ServiceModel.Description. Вот код, который мы использовали раньше:

```
//создать объект поведения;
ServiceMetadataBehavior behavior = new ServiceMetadataBehavior();

//разрешить доступ к метаданным по протоколу Http;
behavior.HttpGetEnabled = true;

//добавить новое поведение в описание службы;
serviceHost.Description.Behaviors.Add(behavior);

//добавить оконечную mex-точку;
serviceHost.AddServiceEndpoint(
      typeof(IMetadataExchange),
      MetadataExchangeBindings.CreateMexHttpBinding(),
      "mex");
```

Однако правильнее будет сначала проверять существование поведения службы ServiceMetadataBehavior, перед добавлением новой оконечной MEX-точки. И только если такое поведение не будет найдено – создавать его:

```
      ServiceMetadataBehavior behavior;
      behavior =
serviceHost.Description.Behaviors.Find<ServiceMetadataBehavior>();
      if (behavior == null)
      {
          behavior = new ServiceMetadataBehavior();
          serviceHost.Description.Behaviors.Add(behavior);
```

```
}

//добавить оконечную mex-точку;
serviceHost.AddServiceEndpoint(
      typeof(IMetadataExchange),
      MetadataExchangeBindings.CreateMexHttpBinding(), "mex");
```

b. Пример клиента, использующего прокси-класс

Теперь, когда наша служба публикует свои метаданные, перейдем к созданию клиента, который будет общаться со службой через прокси-класс. Прежде всего надо удалить из кода клиента определение интерфейса, описывающего контракт службы. Затем начнем создавать прокси-класс.

В этом нам поможет утилита svcutil.exe, позволяющая создавать прокси-класс и конфигурационный файл клиента. С ней можно работать напрямую в командной строке или же через Visual Studio. Сначала рассмотрим второй вариант. Запустите службу и оставьте ее активной.

1. Создайте в Visual Studio новый проект клиента в виде консольного приложения с именем WCF_ClientProxy;
2. Откройте обозреватель решений проекта клиента;
3. Выделите имя пространства имен своего проекта и нажмите правую кнопку мыши;
4. Выберите команду Add Service Reference (Добавить ссылку на службу);
5. В появившемся окне в поле Address (адрес) занесите базовый адрес службы и нажмите кнопку Go (Перейти);
6. После того как в окнах Service «Служба» и Operation «Операции» отобразится наша служба и ее методы, нажмите OK;
7. Обратите внимание – в окне обозревателя решений появился раздел ServiceReferences с элементом ServiceReference1. Одновременно с этим в составе проекта появился и файл конфигурации клиента – app.config;
8. Добавьте в проекте строку `using пространство_имен_проекта_клиента.ServiceReference1;`
9. В методе main() создайте объект прокси-класса. Запомните, что имя прокси-класса всегда образуется из имени вашего класса службы, к которому в конце добавляется слово Client. Поскольку класс нашей службы называется MathService, то имя прокси-класса будет MathServiceClient. Создаем его объект:
   ```
   MathServiceClient proxy = new MathServiceClient ();
   ```
   Вы уже должны были догадаться, что определение класса MathServerClient к этому моменту реализовано автоматически в нашем проекте утилитой svcutil.exe. Но не напрямую (ведь мы не вызывали эту утилиту непосредственно), а посредством Visual Studio. MathServerClient является partial классом.
10. Теперь мы можем обращаться к методам службы через прокси-класс:

```
int result = proxy.Add(35, 57);
```

Весь код клиента должен выглядеть так:

Пример 9. Клиент, использующий прокси-класс

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ServiceModel;
using WCF_ClientProxy.ServiceReference1;


namespace WCF_ClientProxy
{
    class Program
    {
        static void Main(string[] args)
        {
            MyMathClient proxy = new MyMathClient();

            int result = proxy.Add(35, 57);
            Console.WriteLine("Сумма = {0}", result);
            Console.WriteLine("Для завершения нажмите<ENTER>.\n\n");
            Console.ReadLine();
        }
    }
}
```

Просмотрите конфигурационный файл клиента и обратите внимание, что в нем описана оконечная точка службы. Вы должны понимать, что адреса, привязка и имена контрактов в службе и клиенте должны совпадать. Запомните, что и конфигурационный файл клиента, и прокси-класс были созданы утилитой svcutil.exe. Когда мы выполняли команду «Добавить ссылку на службу», наш клиент обратился к оконечной mex-точке службы и получил от нее в ответ специальный WSDL документ, в котором содержалось необходимое описание службы.

Если бы мы запустили svcutil.exe в командной строке, результат был бы таким же. Эта утилита может выполнять много полезных действий, которые задаются ее ключами. Полное описание ключей можно просмотреть, запустив утилиту таким образом: svcutil /?. Как правило, входной информацией для утилиты являются метаданные службы, которые служба возвращает в WSDL-формате при обращении к mex-точке. На основании этой информации svcutil.exe создает конфигурационные файлы, код прокси-класса и многое другое. Например, чтобы создать прокси-класс нашей службы для клиента, мы могли бы ввести такую команду:

svcutil http://localhost/MathService /config:app.config /out:MyProxy.cs

После этого надо добавить в состав проекта созданный файл MyProxy.cs, выбрав в обозревателе решений команду «Добавить», а затем – «Существующий элемент».

В первом варианте нашего клиента мы связывались с оконечной точкой службы, используя класс ChannelFactory<>. Аналогично тому, как среда выполнения службы WCF автоматически создается вызовом ServiceHost.Open(), среда выполнения для клиента создается классом ChannelFactory<>. Конструктору ChannelFactory<> надо передать в качестве параметров привязку и адрес оконечной точки службы, а контракт указывается как generic-тип. Канал связи с клиентом создавался вызовом метода ChannelFactory<IMyMath>.CreateChannel().

Во втором варианте нашего клиента заботу о создании среды выполнения и канала берет на себя прокси-класс. Создание самого прокси-класса происходит для нас автоматически.

i. использование прокси-класса – смотрим внимательнее

Вспомните, каким образом мы с вами использовали в коде прокси-класс. В нашем примере это выглядело так:

```
MyMathClient proxy = new MyMathClient();

int result = proxy.Add(35, 57);
```

Как видите, объект proxy создавался конструктором по умолчанию. Теперь надо обратить ваше внимание на тот факт, что такой подход будет работать не всегда. Иногда в случае создания объекта прокси-класса с помощью конструктора по умолчанию, приложение будет выбрасывать исключение. Давайте разберемся, когда это может происходить, и научимся избегать таких случаев.

Конструктор по умолчанию можно использовать для создания объекта прокси-класса только в том случае, когда для контракта, который использует этот прокси-класс, создана только одна оконечная точка и эта оконечная точка описана в конфигурационном файле. Вспомните, как в нашей службе были описаны оконечные точки:

```
<services>
      <service behaviorConfiguration="NewBehavior"
            name="Module1Service.MathService">
            <endpoint address="http://localhost/MathService/ep1"
                  binding="basicHttpBinding"
              bindingConfiguration="" contract="Module1Service.IMath" />
            <endpoint binding="mexHttpBinding" bindingConfiguration="" name="mex"
              contract="IMetadataExchange" />
            <host>
              <baseAddresses>
                  <add baseAddress="http://localhost/MathService" />
              </baseAddresses>
            </host>
      </service>
</services>
```

Вы можете возразить, что оконечных точек в этом конфигурационном файле две, а не одна. Но в данном случае, речь не идет о mex-точках. А вторая оконечная точка здесь именно mex-точка. Нас сейчас интересует функциональная оконечная точка с именем ep1. В нашем случае имя точки указано вместе с адресом. Еще это имя может быть указано с помощью атрибута name:

```
<endpoint name="ep1"
      address="http://localhost/MathService"
      binding="basicHttpBinding"
      bindingConfiguration="" contract="Module1Service.IMath" />
```

Дело в том, что у контракта службы может быть описано несколько оконечных точек (помимо mex-точки). Именно в этом случае нельзя применять конструктор по умолчанию, для создания объекта прокси-класса, использующего такой контракт. Для такой ситуации надо использовать перегруженный конструктор прокси-класса, которому необходимо передать либо имя необходимой оконечной точки, указанное как значение атрибута name секции endpoint:

```
MyMathClient proxy = new MyMathClient("ep1");
int result = proxy.Add(35, 57);
```

Либо же, если же вы не используете в конфигурационном файле в секции endpoint атрибут name, для создания объекта прокси-класса вам надо использовать другой перегруженный конструктор, которому передаются адрес оконечной точки и объект привязки:

```
MyMathClient proxy = new MyMathClient("http://localhost/MathService",
      new BasicHttpBinding());
```

или же то же самое, но другими словами:

```
EndpointAddress httpAdr1 = new EndpointAddress
("http://localhost/MathService");
BasicHttpBinding HttpBinding = new BasicHttpBinding ();
MathServiceClient proxy = new MathServiceClient(HttpBinding, httpAdr1);
```

Следует добавить, что после использования объекта прокси-класса его надо закрывать с помощью, конечно же, метода Close():

```
proxy.Close();
```

Закрытие объекта прокси-класса, когда он уже не нужен, полезно не только потому, что освобождаются ресурсы, занятые этим объектом. Этим самым вы также уменьшаете количество открытых подключений на клиентском компьютере. А это количество не безгранично. Если вы не полагаетесь на то, что не будете забывать вызывать метод Close(), возможно вам будет приятно и полезно узнать, что все прокси-классы WCF наследуют вот такому generic-классу:

```
public abstract class ClientBase<T> : ICommunicationObject,IDisposable
```

Тем из вас, кто еще не понял, что в этом приятного, советую обратить внимание на интерфейс IDisposable и вспомнить, что это значит для нас, программистов. А значит это то, что мы можем красиво работать с объектом прокси-класса, не вызывая явно метод Close(). Вспомните вторую ипостась инструкции using:

```
using(MathServiceClient proxy = new MathServiceClient())
{
      int result = proxy.Add(35, 57);
```

```
            . . .
        }
```

Теперь можете быть уверены, что ваш объект прокси-класса будет обязательно закрыт, когда вы перешагнете скобку ”}”. И даже в том случае, если у вас возникнет исключительная ситуация внутри блока using, объект будет закрыт.

Очень часто клиенты обращаются к службе в синхронном режиме. А это значит, что послав службе запрос, клиент блокируется до того момента, пока не придет ответ от службы. Вы понимаете, чем опасна такая ситуация. Если ответ от службы задержится где-то в пути, или же, тем более, если запрос клиента застрянет еще на пути к службе, то клиент будет ждать очень долго. Чтобы избежать ситуации, когда клиент может быть заблокирован на продолжительное время, клиенту отводится определенный таймаут, в течение которого он будет ждать ответ от службы. Если ответ не придет до истечения этого периода, клиент просто перестанет ждать и возникнет исключительная ситуация TimeoutException. Таким образом мы избавляем себя от бесконечного зависания приложения.

Этот таймаут является свойством привязки и по умолчанию равен одной минуте. Однако это значение можно изменять. Изменять значение таймуата можно как в конфигурационном файле, так и в коде. Рассмотрим оба варианта. Чтобы указать значение таймаута в конфигурационном файле, надо создать секцию <bindings> для используемого типа привязки и в элементе <binding> использовать атрибут sendTimeout:

```
<bindings>
  <wsHttpBinding>
    <binding name = "GreaterTimeout" sendTimeout = "00:02:00"/>
  </wsHttpBinding>
</bindings>
```

Такой фрагмент в конфигурационном файле клиента установит время ожидания клиентом ответа от службы в 2 минуты, вместо одной минуты по умолчанию. Но это изменение коснется только запросов, использующих привязку wsHttpBinding.

Чтобы изменить значение таймаута в коде, надо просто установить желаемое значение для свойства sendTimeout используемого объекта привязки. Например, таким образом:

```
WSHttpBinding wsBinding = new WSHttpBinding();
wsBinding.SendTimeout = TimeSpan.FromMinutes(2);
```

И дальше использовать эту привязку по назначению.

c. Поведения службы

В предыдущем разделе, публикуя метаданные службы, мы обращались к поведению. Поведения – это набор классов и интерфейсов, которые позволяют дополнять и уточнять поведение службы или клиента. Например, если мы хотим, чтобы служба

раскрывала свои метаданные, мы должны использовать поведение, определяемое классом ServiceMetadataBehavior. Для уточнения поведения служб предназначен тип IServiceBehavior. Для управления поведением канала со стороны клиента используется тип IChannelBehavior.

В разделе, где описывалось создание mex-точки в коде, упоминалась такая строка:

```
serviceHost.Description.Behaviors.Add(behavior);
```

У класса ServiceHost есть свойство Description, представляющее собой объект класса ServiceDescription. Этот класс содержит полное описание службы и используется для создания метаданных, каналов и конфигураций службы. Среди свойств класса ServiceDescription есть две коллекции: Behaviors, содержащая все поведения службы, и Endpoints, содержащая набор оконечных точек службы. Свойства этого класса можно заполнять и вручную, но обычно это делается автоматически при вызове утилиты svcutil.exe.

На клиентской стороне существует аналогичный класс – ChannelDescription. Среди свойств этого класса есть коллекция IChannelBehaviors, описывающая поведения клиентского канала, и свойство ServiceEndpoint, описывающее оконечную точку, с которой связан канал. Обратите внимание, что ServiceEndpoint – это не коллекция.

Новые термины и понятия

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

прокси-класс – специальный объект, создаваемый в клиенте и, являющийся образом контракта службы;
Мех-точка – специальная оконечная точка службы, раскрывающая внешнему миру контракт службы;
Поведение – объект, предназначенный для добавления службе или клиенту какой-либо функциональности;

2. Контракт данных

Сейчас вы знаете, что в классе службы существуют специальные методы, называемые операциями. Однако операции – не единственные «обитатели» класса службы. Давайте поговорим вот о чем. Когда клиент вызывает какую-либо операцию, он ожидает получить определенный результат. Наша служба отправляет клиенту целочисленный результат. Но ведь могут быть ситуации, когда от какой-либо операции потребуется получить значение пользовательского типа. Данные, пересылаемые между клиентом и службой, должны быть сериализуемыми в XML-формат. Как сериализовать встроенные типы данных система знает сама, а вот если мы попросим ее переслать данные какого-то типа посложнее, который мы сами сконструировали, то мы должны будем выполнить еще немного дополнительной работы.

Подумайте над тем, как можно сделать нашу «математическую» службу более функциональной. Сейчас служба умеет только складывать два переданных ей

значения и возвращать сумму. Давайте научим ее выполнять четыре основные арифметические операции и возвращать нам результат в некотором классе, способном хранить сразу четыре значения. Рассмотрим следующий способ реализации нашего замысла. Опишем в службе вот такой класс:

```
public class MathResult
{
        public double sum;
        public double subtr;
        public double div;
        public double mult;
}
```

Объект такого класса будет хранить в своих членах результаты выполнения четырех арифметических операций и будет использоваться как возвращаемое значение в таком методе:

```
public MathResult Total(int x, int y)
{
        MathResult mr = new MathResult();
        mr.sum = x + y;
        mr.subtr = x - y;
        if (y != 0) mr.div = x / y;
        mr.mult = x * y;
        return mr;
}
```

Создадим новую службу – по-прежнему в виде консольного приложения. Она будет очень похожа на предыдущую службу, но вместо метода Add() будет предоставлять клиенту доступ к методу Total(). А тип возвращаемого значения метода Total() описывается классом `MathResult`. В этом проекте необходимо добавить ссылку на пространство имен System.Runtime.Serialization. Код службы должен выглядеть таким образом:

Пример 10. Служба, требующая контракт данных

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ServiceModel;
using System.Runtime.Serialization;

namespace WCF_MyServerDatContract
{
    [DataContract]
    public class MathResult
    {
        [DataMember]
        public double sum;
        [DataMember]
        public double subtr;
        [DataMember]
        public double div;
        [DataMember]
        public double mult;
```

```
    }

    [ServiceContract]
    public interface IMyMath
    {
        [OperationContract]
        MathResult Total(int x, int y);
    }

    public class MyMath : IMyMath
    {
        public MathResult Total(int x, int y)
        {
            MathResult mr = new MathResult();
            mr.sum = x + y;
            mr.subtr = x - y;
            if (y != 0) mr.div = x / y;
            mr.mult = x * y;
            return mr;
        }
    }

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

Создайте самостоятельно для этой службы конфигурационный файл, раскрывающий mex-точку. Он должен выглядеть, например, так:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
    <system.serviceModel>
        <behaviors>
            <serviceBehaviors>
                <behavior name="NewBehavior0">
                    <serviceMetadata httpGetEnabled="true" httpsGetEnabled="false" />
                </behavior>
            </serviceBehaviors>
        </behaviors>
        <services>
            <service behaviorConfiguration="NewBehavior0" name="WCF_MyServerDatContract.MyMath">
                <endpoint address="http://localhost/MyMath/ep1" binding="basicHttpBinding"
                    bindingConfiguration="" contract="WCF_MyServerDatContract.IMyMath" />
                <endpoint binding="mexHttpBinding" bindingConfiguration="" name="mex"
                    contract="IMetadataExchange" />
                <host>
                    <baseAddresses>
                        <add baseAddress="http://localhost/MyMath" />
```

```
                </baseAddresses>
            </host>
        </service>
    </services>
</system.serviceModel>
</configuration>
```

Небольшие изменения надо сделать и в клиенте, ведь теперь мы вызываем не метод Add(), а метод Total(). Закомментируйте в коде клиента строки:

```
int result = proxy.Add(35, 57);
Console.WriteLine("Сумма = {0}", result);
```

Вместо них напишите следующие:

```
MathResult mr = proxy.Total(35, 38);
Console.WriteLine("Результат: {0} {1} {2} {3}", mr.sum, mr.subtr, mr.div,
      mr.mult);
```

Проверьте наше приложение в действии. Скорее всего, вы получили сообщение об ошибке. Дело в том, что мы не учли одну особенность поведения WCF-приложений. Данные, которыми обмениваются служба и клиент, описываются с помощью XML Schema Definitions (XSD). В нашем случае служба должна передать клиенту тип `MathResult`, определенный нами в .NET. Мы должны каким-либо образом указать WCF-приложению, как преобразовать `MathResult` в XML-формат, а затем обратно. Другими словами, мы должны создать соответствующее XSLT-преобразование. Вспомните соответствующую тему из курса XML. Испугались? ☺ Не бойтесь, делать «руками» это не придется. Все, что нам надо будет сделать, это указать некоторые атрибуты в определении класса `MathResult`, чтобы класс выглядел следующим образом:

```
[DataContract]
public class MathResult
{
    [DataMember]
    public double sum;
    [DataMember]
    public double subtr;
    [DataMember]
    public double div;
    [DataMember]
    public double mult;
}
```

Атрибуты [DataContract] и [DataMember] определены в пространстве имен System.Runtime.Serialization. Поэтому не забудьте добавить к проекту ссылку на это пространство имен и затем указать объявление `using System.Runtime.Serialization`. С этими атрибутами определение нашего класса `MathResult` превратилось в новый вид контракта – контракт данных. Контракт данных – это составная часть контракта службы, описывающая данные, которыми могут обмениваться клиент и служба. Дело в том, что методы служб передают копии объектов. Поэтому типы данных, используемые для параметров и возвращаемых значений методов служб, должны быть сериализуемыми.

При выполнении службы класс DataContractSerializer, входящий в состав WCF, сериализирует в XML-формате все объекты, помеченные атрибутами [DataContract] и [DataMember]. Запомните: если вы хотите пересылать между службой и клиентом данные пользовательского типа, вы должны создать в службе контракт данных, описав в нем этот тип. Контракт данных показывает WCF, как преобразовывать ваш тип данных. Сами преобразования выполняются с помощью указанных атрибутов. Атрибут [DataContract] говорит о том, что пользовательский тип, возле которого этот атрибут задан, должен быть представлен в формате XSD в WSDL-описании службы, которое создает утилита svcutil.exe. Без этого атрибута класс не будет включен в описание службы и не сможет передаваться клиенту. Атрибутом [DataMember] надо помечать те члены класса, которые должны быть указаны в XSD-схеме, при этом не важно, являются ли такие члены класса public или private.
Вы должны понимать, что «пользовательским типом» являются не только классы, а также и структуры. Поэтому объекты структур тоже можно пересылать между службами и клиентами, не забыв указать перед ними атрибуты [DataContract] и [DataMember].

Необходимо сделать одно замечание. Дело в том, что вы могли и не получить сообщения об ошибке при запуске нашего приложения без атрибутов [DataContract] и [DataMember]. Все могло бы замечательно работать и без этих атрибутов. Дело в том, что Visual Studio автоматически добавляет атрибуты [DataContract] и [DataMember] к описанию классов, участвующих в пересылке между службой и клиентами. Честно говоря, логика разработчиков Visual Studio в данном случае непонятна. Ведь не добавляет же студия атрибуты [ServiceContract], например, и другие? Но вы должны уметь пользоваться контрактом данных и возьмите себе за привычку прписывать его явно самостоятельно, не полагаясь в этом вопросе на Visual Studio. К тому же, приложения WCF можно создавать с применением и других инструментов, которые, в отличие от Visual Studio, эти атрибуты проставлять не будут.

а. Контракт данных – смотрим внимательнее

Если мы хотим чтобы служба и клиент могли обмениваться данными пользовательского типа, мы должны обозначить этот тип атрибутом [DataContract], а те поля типа, которые должны пересылаться, надо обозначить атрибутом [DataMember]. При отправке данных службе эти данные перед пересылкой сериализуются. При получения запроса, служба десериализует полученные данные и обработает их. Затем служба сериализует результат обработки и отправит клиенту назад. Вы уже догадались, что получив данные от службы, клиент их десериализует. Таким образом, необходимое требование к пересылаемым данным – возможность их сериализации. Пока запомните это.

Контракты данных могут включаться друг в друга. Например, таким образом:

```
[DataContract]
```

```
class Address
{
    [DataMember]
    public string Country;
    [DataMember]
    public string City;
    [DataMember]
    public string Street;
    [DataMember]
    public string House;
}
[DataContract]
struct Student
{
    [DataMember]
    public string Name;
    [DataMember]
    public double Rating;
    [DataMember]
    public Address Address;
}
```

Поговорим о том, каким образом можно пересылать между клиентами и службами данные некоторых конкретных типов. Я думаю, вы все уже привыкли к применению перечислений и находите этот тип данных весьма полезным в некоторых ситуациях. Перечисления по умолчанию сериализуемы, поэтому, если вы хотите пересылать данные типа перечисления, вам не надо использовать атрибут [DataContract]. Например, такое использование перечисления OrderState вполне допустимо:

```
enum OrderState
{
      New,
      Proved,
      Canceled,
      Completed,
      Archived
}
[DataContract]
struct Order
{
      [DataMember]
      public OrderState State;
      [DataMember]
      public string Number;
      [DataMember]
      public DateTime OpenDate;
}
```

Однако существуют ситуации, когда применение атрибута [DataContract] с перечислениями необходимо. Такая ситуация имеет место, когда вы хотите передать не все значения перечисления, а лишь некоторые из них. Например, если вам надо передать только значения OrderState.Proved, OrderState.Canceled и OrderState.Completed, а два других скрыть, вы должны поступить таким образом. Перед перечислением указать атрибут [DataContract], а перед теми значениями перечисления, которые надо передать – указать атрибут [EnumMember]:

```
[DataContract]
enum OrderState
{
	New,
	[EnumMember]
	Proved,
	[EnumMember]
	Canceled,
	[EnumMember]
	Completed,
	Archived
}
```

В этом случае клиент получит перечисление OrderState только с тремя значениями: OrderState.Proved, OrderState.Canceled и OrderState.Completed.

Коллекции также могут пересылаться между службой и клиентами. Но при этом каждая коллекция на стороне клиента будет преобразовываться в массив элементов соответствующего вида. Например, такая реализация службы вполне допустима:

```
[DataContract]
struct Student
{
	[DataMember]
	public string Name;
	[DataMember]
	public double Rating;
}

[ServiceContract]
interface IStudents
{
	[OperationContract]
	void AddStudent(Student student);
	[OperationContract]	//метод возвращает коллекцию типа Student
	List<Student> GetStudents();
}

class Students : IStudents
{
	List<Student> all_students = new List<Student>();
	public void AddStudent(Student student)
	{
		all_students.Add(student);
	}
	public List<Student> GetStudents ()
	{
		return all_students;
	}
}
```

Но на стороне клиента, возвращаемое значение метода GetStudents() будет представлять собой Student[], и соответственно вызывать этот метод надо будет таким образом:

```
Student[] students = proxy.GetStudents();
```

Если вместо встроенных коллекций .NET таких как List, ArrayList, SortedList или других, вы решите использовать свою собственную коллекцию, вы должны помнить, что она также будет преобразована в массив. Например, для такого контракта:

```
[ServiceContract]
interface ISomething
{
      [OperationContract]     //метод возвращает пользовательскую коллекцию
      MyOwnList<double> GetAList();
}
```

метод GetAList() надо будет вызывать на стороне клиента таким образом:

```
double[] list = proxy.GetAList();
```

Все что требуется в этом случае от созданной вами коллекции MyOwnList, это чтобы она была сериализуемой чтобы она НЕ использовалась с атрибутом [DataContract]. Т.е. ваша коллекция должна быть определена как-нибудь так:

```
[Serializable]
public class MyOwnList<double> : IEnumerable<double>
{. . .}
```

Понятно, что иметь дело с коллекцией намного приятнее, чем с обычным массивом. Поэтому WCF предлагает нам с вами некоторый механизм, позволяющий получить на стороне клиента не просто массив, а коллекцию. Правда, этот механизм работает только для встроенных в .NET коллекций. Рассмотрим этот механизм.

Помните, когда мы в Visual Studio выполняли команду «Добавить ссылку на службу», на экране появлялось одноименное окно. В нижней части этого окна есть кнопка Дополнительно, которую мы нажимали, при создании асинхронного клиента. После нажатия этой кнопки появляется второе окно – Настройки ссылок на службы. В разделе этого окна «Тип данных» есть комбобокс «Тип коллекции». Этот комбобокс позволяет нам указать, как на стороне клиента должна представляться коллекция, обнаруженная в метаданных службы. Вы можете выбрать в этом комбобоксе требуемое вам значение, и, если преобразование окажется возможным, вы получите на стороне клиента не массив, а выбранную вами коллекцию.

Теперь вы обратили внимание на комбобокс размещенный чуть ниже и подписанный «Тип коллекции для словаря». Это аналогичная опция для преобразования в коллекцию коллекции Dictionary, если таковая будет обнаружена в метаданных службы. Вы помните, что данные в Dictionary связаны попарно, поэтому представлять такие данные в виде массива не всегда приемлемо.

b. Перегрузка операций

Прошло совсем немного времени с момента знакомства с технологией WCF, а мы уже создаем достаточно функциональные приложения. Безусловно, наши служба и клиент пока очень просты, но по сравнению с самой первой их версией, прогресс значителен. Подумаем, как бы сделать нашу математическую службу еще более функциональной. Очень хочется, чтобы у вас сейчас возник вопрос: «А можно ли сделать так, чтобы служба умела складывать как целые числа, так и вещественные?».

Давайте немного порассуждаем об этом. Вы должны помнить, что C# допускает перегрузку методов. Исходя из этого, возникает соблазн создать контракт службы таким образом:

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

Однако при таком подходе к делу нас будет ожидать неприятность в виде исключительной ситуации InvalidOperationException. Дело в том, что WCF не допускает перегрузки методов, так как каждая операция идентифицируется по имени, и это имя должно быть уникальным. Выход из данной ситуации существует. Правда, он требует небольших изменений, как на стороне службы, так и на стороне клиента. На стороне службы эти изменения сводятся к использованию свойства Name атрибута [OperationContract]. Вы должны реализовать контракт службы, используя свойства Name для изменения имен обоих перегруженных методов. Например, таким образом:

```
[ServiceContract]
interface IMyMath
{
	[OperationContract(Name = "IntAdd")]
	int Add(int a, int b);
	[OperationContract(Name = "DoubleAdd")]
	double Add(double a, double b);
}
```

В этом случае вы получите возможность вызывать на стороне клиента либо метод IntAdd() либо метод DoubleAdd() и получать ожидаемые результаты. Однако это не очень красивый способ. Было бы намного лучше, если бы клиент мог пользоваться одним перегруженным методом Add(), передавая ему, либо целочисленные либо вещественные значения.

Чтобы достичь такого результата придется немного поработать «руками» и внести некоторые изменения в код на стороне клиента. Но изменения придется вносить не в код, который мы сами написали, а в код, который для нас был создан автоматически. В любом случае это будет полезным. Вы увидите код, который создается для нас автоматически, и сможете оценить объем рутинной работы, от которой нас освобождает Visual Studio. Как правило, нет никакой необходимости изменять этот код вручную. Но для реализации нашей задумки – перегрузки операций WCF-службы, без этого не обойтись. Просмотр этого кода должен будет помочь вам понять, каким образом выполняются те действия, которые мы запрограммировали в нашей службе и в нашем клиенте.

Нам надо будет изменить код контракта службы, который клиент создал на своей стороне. Чтобы получить доступ к этому коду, надо выполнить следующие действия в приложении клиента в Visual Studio:

1. В обозревателе решений раскрыть группу Service References, кликнув по значку «+»;
2. Затем кликнуть по вложенному элементу ServiceReference1;
3. На экране должно открыться окно обозревателя объектов;
4. Найти в окне обозревателя объектов элемент с именем пространства имен вашего проекта и раскрыть его;
5. Внутри этого элемента раскрыть элемент с именем ПРОСТРАНСТВО_ИМЕН_ПРОЕКТА.ServiceReference1;
6. В этом элементе сделать двойной клик на элементе с именем контракта вашей службы, например IMath;
7. На экране должен отобразиться код файла Reference.sc;
8. В этом коде найдите описание наших перегруженных методов, и добавьте к атрибуту OperationContract каждого метода свойство Name с теми значениями, которые вы использовали в службе;
9. У меня все это выглядело таким образом. Добавленные свойства Name я выделил жирным шрифтом:

```
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "3.0.0.0")]
[System.ServiceModel.ServiceContractAttribute(ConfigurationName="ServiceReference1.IMath")]
public interface IMath {

        [System.ServiceModel.OperationContractAttribute(Action =
        "http://tempuri.org/IMath/IntAdd", ReplyAction =
        "http://tempuri.org/IMath/IntAddResponse", Name = "IntAdd")]
            int Add(int a, int b);

        [System.ServiceModel.OperationContractAttribute(Action =
        "http://tempuri.org/IMath/DoubleAdd", ReplyAction =
        "http://tempuri.org/IMath/DoubleAddResponse", Name = "DoubleAdd")]
        double Add(double a, double b);
}
```

10. Найти в этом же файле немного ниже реализацию наших двух операций и изменить ее следующим образом:

```
public int Add(int a, int b) {
        return base.Channel.Add(a, b);
}

public double Add(double a, double b) {
        return base.Channel.Add(a, b);
}
```

11. На этом сама перегрузка завершена.

Все что осталось сделать – это воспользоваться перегруженными операциями службы в клиенте. Например, таким образом:

```
double resultd = proxy.Add(35.2, 57.45);
Console.WriteLine("Сумма = {0}", resultd);
```

```
int resulti = proxy.Add(35, 157);
Console.WriteLine("Сумма = {0}", resulti);
```

Нельзя сказать, что перегрузка операций службы жизненно необходима, но иногда она может быть очень полезной и красивой с точки зрения реализации службы.

3. Асинхронный клиент

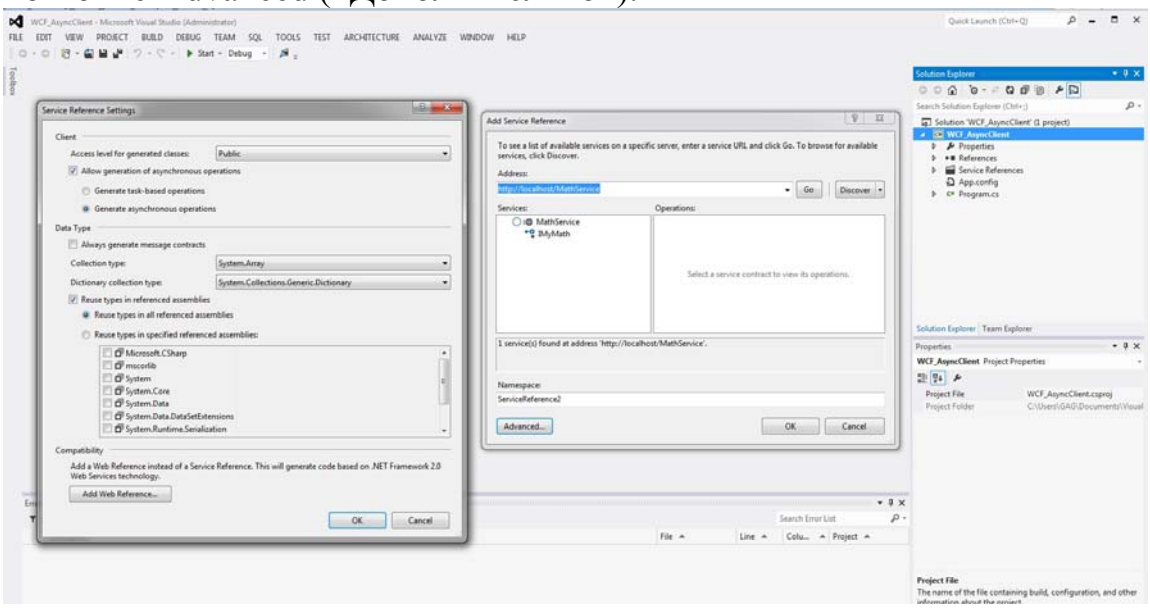
Давайте вкратце повторим то, что мы изучили на предыдущем уроке. Когда при создании клиента мы выполняем команду «Добавить ссылку на службу» (Add Service Reference), вызывается утилита svcutil.exe. Эта утилита ищет по заданному нами базовому адресу оконечную mex-точку. Если mex-точка найдена, то на стороне клиента создается прокси-класс, соответствующий классу службы. Этот прокси-класс позволяет клиенту обращаться к методам службы, как к своим локальным методам. Когда в клиенте вызывается какой-либо метод службы, прокси-класс создает и отправляет службе сообщение в формате SOAP. В этом сообщении содержится имя вызываемого метода и переданные ему параметры. После отправки сообщения прокси-класс ждет ответ от службы, и когда ответ приходит, он извлекает из него полученный результат и преобразовывает его к требуемому типу .NET.
Вы должны понимать, что общение клиента со службой происходит в синхронном режиме. Другими словами, клиент блокируется до того момента времени, когда придет ответ от службы. Иногда такое поведение клиента недопустимо.
Мы можем реализовать асинхронное общение клиента со службой, при котором клиент не будет блокироваться до момента получения ответа от службы. Для этого надо поступить следующим образом. При выполнении команды Add Service Reference («Добавить ссылку на службу») надо нажать кнопку Advanced («Дополнительно») в нижней части окна. Затем в следующем диалоговом окне надо отметить поле Generate Asynchronous Operations («Генерировать асинхронные операции»). Все остальное утилита svcutil.exe и класс IAsyncResult выполнят автоматически. В прокси-классе для каждого метода службы, помеченного атрибутом [OperationContract], будет создана пара методов – BeginИмяМетода() и EndИмяМетода().

В случае асинхронного обращения к службе, клиент вызовет метод BeginИмяМетода(), и этот метод начнет выполняться в отдельном потоке. Клиент в это время сможет продолжать свою работу в текущем потоке без какой-либо блокировки. Когда метод BeginИмяМетода() завершит свою работу, он «сообщит» об этом клиенту через один из своих параметров. Дело в том, что при создании метода BeginИмяМетода() к его «родным» параметрам будут добавлены еще два. Первый добавленный параметр – делегат типа AsyncCallback, ссылающийся на локальный метод с одним аргументом типа AsyncResult. Именно этот локальный метод и будет вызван, когда завершится выполнение метода BeginИмяМетода(). Вызов этого callback метода будет сигналом того, что BeginИмяМетода() завершил свою работу. Второй добавленный параметр – это объект, который будет передан локальному методу, на который ссылается делегат AsyncCallback, в свойстве параметра этого локального

метода AsyncResult.AsyncState. Получение результата от службы происходит при вызове метода EndИмяМетода(), который надо вызывать после завершения работы BeginИмяМетода().

Например, для нашего клиента в этом случае будут созданы два метода – BeginTotal() и EndTotal(). Обратите внимание, что все эти изменения затрагивают только клиента. Служба ничего не знает о том, как ее вызывают – синхронно или асинхронно.

Давайте создадим для нашей службы асинхронного клиента. Как вы помните, в службе ничего изменять не надо. Надо просто запустить нашу службу и оставить ее активной. Теперь начнем новый проект. По-прежнему наш клиент будет консольным приложением. Добавим в клиенте ссылку на службу, не забыв при этом отметить поле Generate Asynchronous Operations («Генерировать асинхронные операции») в диалоговом окне Advanced («Дополнительно»).



Затем реализуем следующий код:

Пример 11. Клиент, вызывающий службу асинхронно

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ServiceModel;
using WCF_AsyncClient.ServiceReference1;

namespace WCF_AsyncClient
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            MyMathClient proxy = new MyMathClient();  //создаем объект прокси-класса
            IAsyncResult arAdd;   //готовим возвращаемое значение метода BeginTotal()

            //вызываем метод BeginTotal()
            //обратите внимание на 3-й и 4-й параметры:
            //GetSumCallback – адрес метода, который будет вызван по завершении
            //асинхронного вызова BeginTotal();
            //proxy – объект, передаваемый методу GetSumCallback()
            //через свойство AsyncState его параметра. Нам этот объект будет
            //нужем в методе GetSumCallback() для вызова EndTotal()

            arAdd = proxy.BeginTotal(100, 50, GetSumCallback, proxy);

            Console.WriteLine("Для завершения нажмите<ENTER>\n\n");
            Console.ReadLine();
        }
        //локальный метод, который будет вызван по завершении
        //асинхронного вызова BeginTotal();
        static void GetSumCallback(IAsyncResult ar)
        {
            MathResult mr = ((MyMathClient)ar.AsyncState).EndTotal(ar);
            Console.WriteLine("Результат: {0} {1} {2} {3}", mr.sum, mr.subtr, mr.div,
            mr.mult);
        }
    }

}
```

Теперь запустите нашего клиента. Посмотрите внимательно на его вывод. Сначала вы увидите сообщение «Для завершения нажмите<ENTER>», а уже после этого – результат работы метода Total(). Это говорит о том, что клиент не ждал завершения инициированного им обращения к службе, а продолжил свою работу и выполнил команду Console.WriteLine(«Для завершения нажмите<ENTER>\n\n»). Затем клиент оставался активным, так как ожидал он нас нажатия Enter. В это время служба прислала ответ, т.е. метод BeginTotal() завершил свою работу, и был вызван метод GetSumCallback(), который и вывел на экран результат работы метода Total(). Другими словами, наш клиент общается со службой в асинхронном режиме и не блокируется.

4. Другие способы хостинга службы

a. Библиотека службы WCF

Каким образом мы хостили наши службы до сих пор? Правильно – в консольных приложениях. Поскольку класс Program консольного приложения содержит метод main(), значит, он запускается как отдельный процесс. Чтобы разместить в этом процессе экземпляр нашей службы, мы создавали экземпляр класса ServiceHost и передавали ему в качестве параметра класс нашей службы. Все что оставалось сделать после этого – вызвать метод Open() класса ServiceHost, и наша служба была готова принимать обращения клиентов.

Я думаю, сейчас мы можем смело перейти к другим способам реализации служб, поскольку вы уже четко представляете, что такое служба, и специфика реализации

службы не будет отвлекать ваше внимание. Очень часто службу WCF создают в виде dll-библиотеки, которую затем можно хостить в разных процессах. Рассмотрим создание такой службы.

1. Запустите Visual Studio и создайте новый проект типа «Библиотека службы WCF» с именем WCF_Service_dll;
2. Откройте обозреватель решений и внимательно изучите структуру проекта. Вы увидите два файла – IService1.cs и Service1.cs.
3. Откройте код файла IService1.cs. Он предназначен для описания контрактов службы. Там уже размещены заготовки контракта службы и контракта данных. Нам эти заготовки не нужны – удалите их. Однако обратите внимание на примечание об имени службы!
4. Добавьте в файл IService1.cs наши контракты службы и данных:

```
[DataContract]
public class MathResult
{
    [DataMember]
    public double sum;
    [DataMember]
    public double subtr;
    [DataMember]
    public double div;
    [DataMember]
    public double mult;
}

[ServiceContract]
public interface IMyMath
{
    [OperationContract]
    MathResult Total(int x, int y);
}
```

   Поскольку мы изменили имя интерфейса (вспомните примечание!), мы должны отразить это изменение в файле App.config. Обратите внимание, что конфигурационный файл службы создается автоматически;
5. Откройте файл App.config и измените в элементе endpoint в атрибуте contract имя интерфейса IService1 на IMyMath. Затем в элементе service в атрибуте name замените Service1 на MyMath. Наконец, в элементе baseAddresses во вложенном элементе add в атрибуте baseAddress тоже замените Service1 на MyMath;
6. Откройте код файла Service1.cs и вместо указанных там заготовок занесите определение класса нашей службы:

```
public class MyMath : IMyMath
{
    public MathResult Total(int x, int y)
    {
        MathResult mr = new MathResult();
        mr.sum = x + y;
        mr.subtr = x - y;
        if (y != 0) mr.div = x / y;
        mr.mult = x * y;
        return mr;
```

```
        }
    }
```

7. Откомпилируйте свой проект и попробуйте его запустить.

Вы увидите, что Visual Studio запустит тестовый хостинг и, в случае успеха, развернет нашу службу и запустит тестовый клиент. Если тестовый клиент запустился и вы видите в нем методы нашей службы, значит dll создана правильно.

b. Хостинг WCF службы в Windows службе

Теперь мы имеем в своем распоряжении WCF-службу, созданную в виде dll. Чтобы служба работала, ее надо разместить в каком-либо процессе. Рассмотрим пример размещения нашей dll в службе Windows. Вы должны еще помнить, что мы говорили о службах в первом уроке. Службы – это специфические приложения, которые не имеют пользовательского интерфейса и не могут активироваться, как обычные приложения. Службы управляются с помощью специальной оболочки Windows. Если вы введете команды Win+”R”, а затем services.msc и нажмете «OK», то эта оболочка активируется, и вы увидите перечень служб в вашей системе. Вы увидите имя каждой службы, ее описание, текущий статус, тип запуска и создателя службы. Если вы выделите какую-либо службу в этом списке, по слева активируются команды, которые можно применить к выбранной службе. Все, что мы можем сделать с любой службой в этой оболочке, это запустить службу, остановить службу и перезапустить службу. Выбор не большой, но вы должны понимать, что службы не являются пользовательскими приложениями. Каждая служба умеет делать что-то свое. Когда служба запускается, она начинает делать свою работу. Как правило, незаметно для пользователя. Если мы разместим в приложении типа служба Windows нашу WCF-службу, то мы можем поручить операционной системе заботу о запуске и поддержке в рабочем состоянии нашей службы. Давай посмотрим, как это можно сделать.

Мы не будем создавать службу Windows для нашей WCF-службы «с нуля». Мы воспользуемся для этой цели специальным шаблоном. В этом шаблоне вы найдете два метода OnStart() и OnStop(). В этих методах разработчики должны размещать код, который служба Windows будет начинать выполнять при запуске (метод OnStart()) и при остановке (метод OnStop()). Мы разместим в этих методах код выполняющий активацию и деактивацию среды выполнения нашей WCF-службы. Вот как это выглядит:

1. Запустите Visual Studio, выберите тип проекта Visual C# - Windows и выберите шаблон Windows Service (Служба Windows);
2. Переименуйте создаваемый проект на WindowsServiceHostDll и нажмите OK;
3. Перейдите в обозреватель решения;
4. Щелкните правой кнопкой мыши на пустом пространстве конструктора и выберите Add Installer (Добавить установщик);

5. В обозревателе решения появится новый файл ProjectInstaller.cs, который тоже откроется в конструкторе;
6. В конструкторе щелкните правой кнопкой мыши на объекте serviceProcessInstaller1 и выберите Properties (Свойства);
7. Для свойства Account выберите значение LocalSystem;
8. Щелкните правой кнопкой мыши на файле Service1.cs и выберите View Code (Перейти к коду);
9. Теперь к нашей Windows-службе надо добавить ссылку на пространство имен System.ServiceModel и строку кода using System.ServiceModel;
10. Теперь надо разместить в Windows-службе нашу WCF-службу, созданную в виде dll библиотеки. Для этого надо снова выполнить команду Project – Add Reference (Проект-Добавить ссылку), затем перейти на закладку Browse (Обзор), выбрать в появившемся окне наш dll файл и сказать OK. Имя пространства имен, в котором мы создали dll, появится в списке ссылок в обозревателе решений;
11. Добавьте строку кода using пространство_имен_dll (пространство_имен_dll – имя пространства имен, в котором мы создали dll);
12. В файле Service1.cs в классе Service1 объявите статический internal член
```
internal static ServiceHost myServiceHost = null;
```
13. Измените метод OnStart(), чтобы он выглядел таким образом:
```
protected override void OnStart(string[] args)
{
    if (myServiceHost != null)
    {
        myServiceHost.Close();
    }

    myServiceHost = new ServiceHost(typeof(MyMath));
    myServiceHost.Open();

}
```
14. Измените метод OnStop(), чтобы он выглядел таким образом:
```
protected override void OnStop()
{
    if (myServiceHost != null)
    {
        myServiceHost.Close();
        myServiceHost = null;
    }

}
```
15. Добавьте к решению файл App.config, как мы делали это раньше. Вы должны получить такой конфигурационный файл:
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
    <system.serviceModel>
        <behaviors>
            <serviceBehaviors>
```

```
                <behavior name="NewBehavior0">
                    <serviceMetadata />
                </behavior>
            </serviceBehaviors>
        </behaviors>
        <services>
            <service behaviorConfiguration="NewBehavior0" name="WCF_Service_dll.MyMath">
                <endpoint address="http://localhost//WindowsServiceHostDll/MyMath/ep1"
                    binding="basicHttpBinding" bindingConfiguration=""
contract="WCF_Service_dll.IMyMath" />
                <endpoint binding="mexHttpBinding" bindingConfiguration="" name="mex"
                    contract="IMetadataExchange" />
                <host>
                    <baseAddresses>
                        <add baseAddress="http://localhost//WindowsServiceHostDll/MyMath" />
                    </baseAddresses>
                </host>
            </service>
        </services>
    </system.serviceModel>
</configuration>
```

16. Перестройте решение и убедитесь, что в папке bin появился файл WindowsServiceHostDll.exe;
17. Теперь, используя утилиту InstallUtil.exe, установите в системе нашу Windows-службу, в которой хостится наша WCF-служба. Мы делали это в начале нашего курса. Для этого в режиме командной строки надо выполнить команду:

```
InstallUtil.exe WindowsServiceHostDll.exe
```

18. Теперь перейдите к управлению службами, например, набрав в командной строке services.msc;
19. В списке служб найдите нашу службу с именем Service1 (это имя по умолчанию, заданное в свойстве ServiceName объекта ServiceInstaller1);
20. Щелкните на нашей службе правой кнопкой мыши и выберите команду «Пуск». Статус службы должен измениться, приняв значение «Работает»;

Теперь наша служба активна и ждет обращений клиентов. Настройте клиента для работы с нашей службой (можете взять клиента из нашего примера и заменить в нем адрес) и проверьте ее работу.

Для удаления службы надо выполнить команду: InstallUtil.exe /u WindowsServiceHostDll.exe.

Это не единственный способ разместить WCF-службу в Windows-службе. Мы могли поступить по-другому. Вместо действий, указанных в пункте 11, мы могли добавить к проекту Windows-службы новый проект, типа «служба WCF». В обозревателе решений появились бы два файла: IService1.cs и Service1.cs. В файле IService1.cs можно было бы указать контракты нашей службы, а в файле Service1.cs – класс службы, как мы делали при создании службы в виде dll. Результат был бы таким же.

c. Хостинг WCF-службы в IIS

Вы, конечно же, знаете термин «сервер» и часто его употребляете в разговорах. При этом вы четко понимаете, что вы имеете в виду под «сервером». Но уверены ли вы, что ваши собеседники понимают этот термин так же, как вы того хотите? Дело в том, что сервером можно называть и мощный специализированный компьютер, и некоторые виды программных продуктов. Например, чтобы мы могли иметь доступ к удаленным Интернет-ресурсам, эти ресурсы должны располагаться на специальном компьютере-сервере, на котором должны быть установлены специальные программы – веб-серверы. Одной из таких программ веб-серверов является Apache, предназначенный в основном для Unix-подобных операционных систем. Еще один популярный сервер - IIS (Internet Information Services), разработанный корпорацией MicroSoft. Но IIS это не только веб-сервер, он содержит еще сервероа и для некоторых других служб Интернета. IIS прекрасно приспособлен для хостинга WCF-служб.

Давайте рассмотрим, как разместить WCF-службу в Internet Information Services. Необходимо выполнить следующие действия:

1. создать веб-приложение, указав в поле «Расположение» не файловую систему, а HTTP;
2. задать имя виртуального каталога, например http://localhost/MyService;
3. добавить ссылку на проект, реализующий службу, которую вы хотите хостить в IIS;
4. в обозревателе решения удалить файл default.aspx;
5. добавить в обозревателе решения новый текстовый файл и назвать его service.svc;
6. открыть файл service.svc и вставить в него следующий код:
   <%@ ServiceHost Service="HostService.MyService" %>

вставить в файл web.config, в раздел <configuration> секцию <system.serviceModel>:

```
<system.serviceModel>
   <services>
     <service behaviorConfiguration=" MyServiceBehaviour"
     name="HostService.MyService ">
       <endpoint address="" binding="wsHttpBinding"
       contract=" MyService.Contract.IMyService ">
       </endpoint>
     </service>
   </services>
   <behaviors>
     <serviceBehaviors>
       <behavior name=" MyServiceBehaviour">
         <serviceMetadata httpGetEnabled="True" httpGetUrl=""/>
       </behavior>
     </serviceBehaviors>
   </behaviors>
</system.serviceModel>
```

Перестройте приложение. Теперь ваша служба размещена в IIS. В svc-файле содержится ссылка на службу, при этом сама служба (в виде dll) может находиться в виртуальном каталога IIS.

Однако существует возможность разместить службу непосредственно в svc-файле. Тогда этот файл будет выглядеть, например, так:

<%@ ServiceHost Language=c# Service=" HostService.MyService " %>

```
using System;
using System.ServiceModel;

namespace HostService
{
    [ServiceContract]
    public interface IMyService
    {
        [OperationContract]
        void Method1();
        [OperationContract]
        double Method2();
    }

    [ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
    public class MyService : IMyService
    {
        public void Method1()
        {
                //реализация метода
        }

        public double Method2()
        {
                //реализация метода
        }
    }
}
```

d. Хостинг WCF-службы в WAS

Обратите внимание на тот факт, что хостинг службы в IIS накладывает на службу одно существенное ограничение. Общаться с такой службой можно будет только по протоколу HTTP. Кроме этого не стоит забывать, что IIS – прежде всего веб-сервер, а не механизм для хостинга. Учитывая это, Microsoft разработала специальный инструмент для хостинга – Windows Activation Service (WAS). Это системная служба, включенная в состав Windows Vista, Windows Server 2008 и Windows 7 (и следующих версий). WAS может хостить как службы, так и веб-сайты. Службы, размещенные в WAS могут использовать все протоколы, доступные WCF, а не только HTTP, как в случае с хостингом в IIS. Чтобы разместить WCF-службу в WAS, необходимо создать .svc-файл.

5. Односторонние операции

Давайте сейчас сосредоточимся на том, каким образом общаются клиент и служба. Во всех наших примерах клиент отправлял службе сообщение с именем метода службы (операции) и параметрами для этого метода. Затем, в случае синхронного обращения к службе, клиент блокировался до момента получения от службы результата. Или же, в случае асинхронного обращения к службе, клиент продолжал свою работу без блокировки, в какой-то момент получая ответ от службы. В обоих этих случаях речь идет об общении типа «отправить запрос – получить ответ». Но бывают случаи, когда клиенту необходимо лишь отправить службе запрос, а ответ службы его не интересует. Интуитивно понятно, что если служба не должна отправлять клиенту ответ, то реализация такой службы будет более простой. WCF позволяет службе реализовать такое поведение. Это выполняется на уровне операций службы.

Чтобы указать, что какой-то из методов службы, включенных в контракт, не должен возвращать клиенту ответ, надо в атрибуте [OperationContract] такого метода добавить модификатор IsOneWay=true. Понятно, что такой метод должен указывать свой тип возвращаемого значения как void. Если вы укажете для такого метода тип возвращаемого значения отличный от void вы получите исключительную ситуацию InvalidOperationException. Стоит отметить, что все привязки встроенные в WCF, поддерживают односторонние операции.

Обратите внимание – когда вы используете в службе обычный, не односторонний метод с типом возвращаемого значения void, клиент на самом деле ожидает ответ от службы. Этим ответом будет пустое сообщение. Однако, в случае возникновения ошибки, в этом пустом сообщении будет содержаться описание ошибки, что поможет разобраться с ошибочной ситуацией.

Давайте рассмотрим односторонний способ общения на примере. Вы уже, наверное, устали от нашей «математической» службы. Пусть она тоже немного отдохнет, а мы тем временем создадим другую пару «служба-клиент».

Создадим службу в виде dll-библиотеки. Эта служба будет содержать в своем контракте два метода: один односторонний, а второй – обычный. Посмотрим, чем отличается работа этих методов.

Пример 12. Клиент и служба с односторонними методами

1. Запустите Visual Studio и создайте новый проект типа «Библиотека службы WCF» с именем WCF_OneWay_Reply;
2. Откройте код файла IService1.cs. Удалите размещенные там заготовки. Еще раз обратите внимание на примечание об имени службы!
3. Добавьте в файл IService1.cs такой контракт службы:

```
[ServiceContract]
```

```
public interface IReply
{
    [OperationContract(IsOneWay = true)]
    void FastReply();
    [OperationContract]
    void SlowReply();
}
```

Поскольку мы изменили имя интерфейса (вспомните примечание), мы должны отразить это изменение в файле App.config;

4. Откройте файл App.config и измените в элементе endpoint в атрибуте contract имя интерфейса IService1 на IReply. Затем в элементе service в атрибуте name замените Service1 на Reply. Наконец, в элементе baseAddresses во вложенном элементе add в атрибуте baseAddress тоже замените Service1 на Reply;
5. Откройте код файла Service1.cs и вместо указанных там заготовок занесите определение класса нашей службы:

```
public class Reply: IReply
{
    public void FastReply()
    {
        Thread.Sleep(5000);
    }
    public void SlowReply()
    {
        Thread.Sleep(5000);
    }
}
```

Оба могучих метода нашего контракта делают одно и то же – приостанавливают работу приложения на 5 секунд.

6. Откомпилируйте свой проект и попробуйте его запустить.

Если у вас запустится «Тестовый клиент WCF» и вы увидите в нем методы службы, значит все сделано правильно.
Мы не будем сейчас хостить нашу dll-библиотеку ни в каком процессе, просто потому, что для тестирования этой службы вполне достаточно ее размещения в «Тестовом клиенте WCF». В ходе тестирования мы вполне сможем проанализировать отличия в поведении методов FastReply() и SlowReply(), что и является целью создания этой службы. А вот дома вы самостоятельно разместите эту службу в службе Windows.

Итак, оставьте «Тестовый клиент WCF» активным и перейдите к созданию клиента. Клиент наш снова будет консольным приложением. Не будем подробно останавливаться на создании клиента – вы уже должны были освоить этот процесс. В результате работы должен получиться такой код:

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

```
using System.ServiceModel;
using WCF_ClientOneWay.ServiceReference1;

namespace WCF_ClientOneWay
{
    class client
    {
        static void Main(string[] args)
        {
            ReplyClient proxy = new ReplyClient();
            //proxy.FastReply();
            proxy.SlowReply();

            Console.WriteLine("Введите числовое значение ");
            int value = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Вы ввели {0}", value);
            proxy.Close();
        }
    }
}
```

Напоминаем, что для клиента необходимо добавить ссылку на пространство имен System.ServiceModel, а также необходимо выполнить команду «Добавить ссылку на службу», в ходе выполнения которой и будет создан прокси-класс для нашего клиента.

Посмотрите внимательно на то, что делает наш клиент. Он отправляет службе запрос, после чего просит нас ввести какое-либо числовое значение, отображает введенное значение на экране и завершает работу. Сначала запустите клиента, вызвав в нем метод службы FastReply(), а затем снова запустите клиента, вызвав метод SlowReply().

Я думаю, этот пример достаточно показателен. После вызова FastReply(), который помечен как односторонний метод, клиент сразу переходит к выполнению следующей строки и просит ввести числовое значение. Не ожидая 5 секунд до завершения работы метода FastReply()! При вызове SlowReply() клиент вынужден ждать как минимум 5 секунд, прежде чем попросит нас ввести числовое значение. Другими словами, в этом случае клиент ожидает завершения метода SlowReply() и лишь затем продолжает работу. Вы понимаете, что при вызове FastReply() никакой асинхронности нет. Просто клиент не ждет завершения метода службы, потому что не рассчитывает что-либо получить от этого метода.

6. Дуплексные операции

Продолжим разговор о том, каким образом может осуществляться общение службы и клиента. Мы уже рассмотрели способ общения, который можно условно назвать «запрос-ответ», затем рассмотрели способ с использованием односторонних операций, который можно условно назвать «запрос-ответ не нужен». Но существуют и другие варианты общения. Давайте представим случай, когда службе необходимо отправить клиенту сообщение независимо от запроса самого клиента. Например, в случае наступления какого-либо события. Или когда на несколько запросов клиента служба

должна отправить лишь одно сообщение. Или, наконец, если время, необходимое службе для подготовки сообщения клиенту, представляет собой достаточно продолжительный период.

В этих случаях WCF предоставляет так называемые дуплексные контракты. Такие контракты позволяют как клиенту, так и службе отправлять сообщения, не ожидая запроса от противной стороны. Поскольку в этом случае сообщения должны иметь возможность ходить в обоих направлениях, обе стороны должны содержать оконечные точки. По дуплексному каналу можно вести общение как типа «запрос-ответ», так и использовать односторонние операции. Однако надо учитывать один нюанс. Есть каналы, которые не поддерживают двустороннюю передачу сообщений. Природа канала определяется привязкой, используемой при установлении связи клиента со службой. Например, протокол TCP поддерживает двустороннюю связь, а протокол HTTP не поддерживает. В таком случае WCF создает дополнительный второй канал от службы к клиенту. Для реализации дуплексного контракта надо выбирать те из встроенных привязок WCF, которые в названии содержат слово dual, например wsDualHttpBinding.

Дуплексный контракт должен описываться как на стороне службы, так и на стороне клиента. Контракт на стороне службы описывает методы, которые может вызывать клиент. Кроме того, контракт службы должен определить контракт обратного вызова (callback) в свойстве CallbackContract своего атрибута [ServiceContract]. При этом контракт обратного вызова не помечается атрибутом [ServiceContract]. Он и так будет включен в метаданные службы. А вот операции контракта обратного вызова помечать атрибутами [OperationContract] необходимо.

Этот контракт обратного вызова определяет методы, которые служба может вызывать на стороне клиента. Вы помните, что в случае дуплексного контракта клиент должен иметь свою оконечную точку, на которую служба может присылать сообщения? Кроме этого, в классе службы, как правило, определяется член типа контракта обратного вызова, т.е. типа интерфейса, в котором определен callback контракт. Этот член используется для доступа к каналу обратного вызова, от службы к клиенту.

На клиента в случае службы с дуплексным контрактом ложится выполнение двух задач. Клиент должен хостить объект типа контракта обратного вызова и предоставить службе оконечную точку.

Для выполнения первой из этих задач клиент должен реализовать класс, который наследует интерфейсу, определяющему на стороне службы контракт обратного вызова (callback). В этом классе должны быть определены методы, получающие сообщения от службы. Затем клиент должен создать объект этого класса и построить для него среду выполнения с помощью класса InstanceContext. Обратите внимание, что на

стороне клиента имя контракта обратного вызова образуется добавлением к имени контракта службы слова Callback. Нечто подобное происходит на стороне клиента с именем прокси-класса, которое образуется из имени класса службы добавлением слова Client.

Как мы сказали, для создания среды выполнения (хостинга) для объекта типа контракта обратного вызова используется объект класса InstanceContext. Другими словами, клиент хостит объект типа контракта обратного вызова с помощью объекта класса InstanceContext. При создании объекта InstanceContext его конструктору надо передать объект класса, наследующего интерфейс обратного вызова. Также объект InstanceContext обрабатывает сообщения, приходящие от службы.

Все эти действия можно схематично описать следующим кодом:

```
//класс, наследующий интерфейсу контракта обратного вызова
//этот интерфейс описан на стороне службы
class MyCallback : IContractCallback
{
      public void ClientMethod()
      {...}
}

//создание объекта класса, наследующего интерфейсу
//контракта обратного вызова
IContractCallback callback = new MyCallback();
//создание среды выполнения для объекта callback
InstanceContext context = new InstanceContext(callback);
//создание прокси для клиента
IContractCallback proxy = new IContractCallback(context);
//запуск какого-то метода на стороне службы
proxy.ServiceMethod();
```

В силу специфики задач, стоящих перед клиентом, реализующим дуплексный контракт, прокси-класс для такого клиента отличается от обычных прокси-классов, с которыми мы имели дело до сих пор. В данном случае прокси-класс наследует специальному generic классу DuplexClientBase<T>. Обратите внимание на тот факт, что в качестве generic параметра для этого класса можно указать абсолютно произвольный тип. Никакая проверка на предмет того, соответствует ли указанный тип какому-либо контракту обратного вызова на этапе компиляции не выполняется. Если вы укажете тип, не соответствующий ожидаемому контракту, то даже на этапе выполнения объект прокси будет создан. Ошибка возникнет только в тот момент, когда вы попробуете вызвать через созданный объект прокси какой-либо метод и будет сгенерировано исключение InvalidOperationException.

Пример 13. Служба и клиент с дуплексным контрактом

Это описание реализации дуплексного контракта может показаться вам сложным. Но не расстраивайтесь. Пример, который мы сейчас разберем, должен расставить все по своим местам. Мы реализуем службу, которая будет получать от клиента запрос и

отвечать на него следующим образом: после получения запроса служба будет ожидать окончания текущей минуты, а с наступлением следующей минуты начнет отправлять клиенту number сообщений с периодичностью period миллисекунд, где number и period – значения, полученные от клиента. В каждом сообщении служба будет передавать клиенту текущее время.

1. Запустите Visual Studio и создайте новый проект типа «Библиотека службы WCF» с именем WCF_DuplexSvc;
2. Откройте код файла IService1.cs. Удалите размещенные там заготовки. И снова обратите внимание на примечание об имени службы!
3. Добавьте в файл IService1.cs такой контракт службы:

```
/*это контракт службы, метод ReturnTime() может вызываться клиентом, при вызове
клиент передает службе period – периодичность, с которой клиент хочет получать
сообщения службы и number – количество этих сообщений. Чтобы не блокировать
клиента, метод объявлен как односторонний. Обратите внимание на атрибут, который
указывает, что контракт имеет callback составляющую*/

[ServiceContract(CallbackContract = typeof(IClientCallback))]
public interface IDuplexSvc
{
      [OperationContract(IsOneWay = true)]
      void ReturnTime(int period, int number);
}
/*это вторая часть контракта службы – контракт обратного вызова, метод
ReceiveTime() описан на стороне клиента и может вызываться службой, при вызове
служба передает клиенту информацию в string str*/

public interface IClientCallback
{
      [OperationContract(IsOneWay = true)]
      void ReceiveTime(string str);
}
```

4. Поскольку мы изменили имя интерфейса (вспомните примечание), мы должны отразить это изменение в файле App.config. Откройте файл App.config и измените в элементе endpoint в атрибуте contract имя интерфейса IService1 на IDuplexSvc. Затем в элементе service в атрибуте name замените Service1 на DuplexSvc. Наконец, в элементе baseAddresses во вложенном элементе add в атрибуте baseAddress тоже замените Service1 на DuplexSvc;
5. При этом не забудьте выбрать привязку, поддерживающую дуплексные операции, например wsDualHttpBinding;
6. Откройте код файла Service1.cs и вместо указанных там заготовок занесите такой код:

```
      public class DuplexSvc: IDuplexSvc
      {
/*метод ReturnTime() каждый раз запускается в новом потоке, для запуска потока
используется делегат ParameterizedThreadStart, способный передать один параметр типа
object, но поскольку нам надо передать в поток два параметра типа int, мы упаковываем
их в List<int> и в таком виде передаем методу потока SendTimeToCLient() */
            public void ReturnTime(int period, int number)
```

```
                {
                        DataValues src = new DataValues();
                        src.callback =
                        OperationContext.Current.GetCallbackChannel<IClientCallback>();
                        Thread t =
                              new Thread(new
                              ParameterizedThreadStart(src.SendTimeToCLient));
                        t.IsBackground = true;
                        List<int> parameters = new List<int>();
                        parameters.Add(period);
                        parameters.Add(number);
                        t.Start(parameters);
                }
        }

/*метод SendTimeToCLient() будет вызван после обращения клиента к нашей службе. Он
подождет до начала следующей минуты и начнет отправлять клиенту сообщения с
периодичностью period миллисекунд, всего таких сообщений он отправит number. Каждое из
этих сообщений доставляется клиенту методом ReceiveTime() */

        public class DataValues
        {
              //член типа контракта обратного вызова
              public IClientCallback callback = null;

              public void SendTimeToCLient(object data)
              {
                    //по условию задачи надо подождать до начала следующей минуты
                    int s = 60 - DateTime.Now.Second;
                    Thread.Sleep(s * 1000);
                    DateTime start = DateTime.Now;
                    //достать из object data наши два параметра типа int,
                    List<int> parameters = (List<int>)data;
                    int period = parameters[0];
                    int number = parameters[1];
                    //каждое сообщение клиенту готовится в цикле
                    for (int i = 0; i < number; i++)
                    {
                          try
                          {
                                //задержка между сообщениями period секунд
                                Thread.Sleep(period * 1000);
                                TimeSpan result = DateTime.Now - start;
                                TimeSpan r = result.Add(new TimeSpan(0, 0, s));

    callback.ReceiveTime(DateTime.Now.ToLongTimeString().ToString() +
                                " время работы со службой - " +
                                r.Minutes.ToString() + ":" +
                                r.Seconds.ToString());
                          }
                          catch (Exception ex)
                          {
                          }
                    }
              }
        }
```

Откомпилируйте свой проект и попробуйте его запустить. Если все сделано правильно, запустится «Тестовый клиент WCF». Для проверки работы службы, вам надо будет разместить ее в службе Windows. Сделаете это дома.
Перейдите к созданию клиента, реализующего дуплексный контракт. Клиент будет создан как консольное приложение.

Создайте проект со следующим кодом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using WCF_DuplexClient.ServiceReference1;

namespace WCF_DuplexClient
{
    public class CallbackHandler : IDuplexSvcCallback
    {
        static InstanceContext site = new InstanceContext(new CallbackHandler());
        public static DuplexSvcClient proxy = new DuplexSvcClient(site);

        public void ReceiveTime(string str)
        {
            Console.WriteLine("Получено сообщение :\n" + str);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            CallbackHandler.proxy.ReturnTime(2, 5);
            Console.ReadKey();
        }
    }
}
```

При создании проекта добавьте ссылку на пространство имен System.ServiceModel, а также добавьте ссылку на службу, указав адрес нашей дуплексной службы. Этот адрес можете взять из файла службы App.config, он может быть таким:
http://localhost:8731/Design_Time_Addresses/DuplexServer/DuplexSvc

После запуска службы (смотрите домашнее задание 1), запустите созданное клиентское приложение и взгляните на результат. Будьте достаточно терпеливы, чтобы дождаться начала следующей минуты ☺.

Домашнее задание

1. Разместить в службе Windows нашу WCF службу с дуплексным контрактом из примера 13 и протестировать ее с клиентом.

2. Создать WCF-службу и клиента, которые будут работать следующим правилам:

1. Служба получает от клиента string **path,** содержащий путь к каталогу на компьютере, где хостится служба;
2. Служба отправляет клиенту содержимое паки, расположенной по пути **path**;
3. Служба должна опубликовать в конфигурационном файле свои метаданные;
4. Общение службы и клиента должно выполняться по **tcp** протоколу.
5. Клиент не должен блокироваться после отправки сообщения службе, но при этом не должен использовать асинхронные способы доступа к службе;

WCF-служба должна быть реализована в виде dll-библиотеки. Клиент должен быть консольным приложением.