

Projektowanie aplikacji ASP.NET  
Wykład 12/15  
Windows Communication Foundation

Wiktor Zychła 2018/2019

---

Spis treści

2	Wprowadzenie .....	2
3	Kontrakt + adres + wiązanie (Contract + Address + Binding) .....	3
3.1	ASMX WebServices .....	3
3.2	WSDL – Web Service Description Language .....	3
3.3	WCF w klasycznym potoku wytwarzania .....	5
3.4	WCF w „odwróconym” potoku wytwarzania .....	6
4	Aktywacja WCF bez elementów deklaracyjnych .....	8
5	Wytwarzanie kodu klienckiego dla WCF .....	9
5.1	Użycie klasy ChannelFactory .....	9
5.2	Użycie klasy ClientBase .....	9
6	Rozszerzenia WCF .....	11
7	Hostowanie WCF poza serwerem aplikacyjnym .....	15

## 2 Wprowadzenie

Podsystem Windows Communication Foundation ma w założeniach eliminować ograniczenia omówionego na poprzednim wykładzie prostego podsystemu ASMX WebServices. Te ograniczenia to:

- Ścisłe związanie WebServices ze środowiskiem IIS – brak możliwości hostowania usługi poza aplikacją typu Web. O ile w przypadku silnika renderowania stron nie jest to tak poważny problem, o tyle chciałoby się móc hostować usługę aplikacyjną na przykład w aplikacji konsolowej
- Brak możliwości rozszerzania potoku wywołania – na przykład modyfikowania wchodzącego/wychodzącego strumienia XML
- Brak wsparcia dla innych wiązań niż http[s], na przykład dla komunikacji serializowanej binarnie po TCP
- Brak wsparcia dla rozwijanych interoperacyjnych rozszerzeń dla usług aplikacyjnych, tzw. [WS-\\*](#)

Te (i inne) powody doprowadziły do przepisania implementacji stosu usług aplikacyjnych do osobnego podsystemu – WCF.

Z punktu widzenia rozwijania aplikacji WCF ma, oprócz eliminacji w/w ograniczeń klasycznego WebServices, następujące cechy:

- Usługa zrealizowana z wykorzystaniem podstawowego typu hosta (tzw. basicHttpBinding) jest wstecznie kompatybilna z usługą typu WebService – klient i serwer obu usług mogą być stosowane zamiennie. Dzięki temu istnieje możliwość wymiany stosu technologicznego serwera z WebServices na WCF bez zmiany wygenerowanych klienckich proxy
- Implementacja oparta o WCF powinna w większości przypadków działać szybciej niż odpowiadająca jej usługa typu WebService

### 3 Kontrakt + adres + wiązanie (Contract + Address + Binding)

O usłudze internetowej można myśleć jak o trójce (kontrakt, adres, wiązanie) gdzie:

- Kontrakt – to specyfikacja wejścia / wyjścia, rozumiana jako lista udostępnianych metod, gdzie każda z metod ma nazwę, zbiór parametrów wywołania oraz typ wartości zwracanej
- Adres – adres pod którym usługa jest dostępna w sieci, sposób wyrażenia adresu może być zależny od protokołu
- Wiązanie – protokół komunikacyjny/transportowy oraz sposób przekazywania argumentów wywołania

#### 3.1 ASMX WebServices

W przypadku ASMX WebServices, kontrakt usługi wyrażony jest w implementacji w postaci kodu poszczególnych metod. Na przykład dla usługi

```
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
public class WebService1 : System.Web.Services.WebService
{
    [WebMethod]
    public string DoWork(string work)
    {
        return "work done " + work;
    }
}
```

można mówić o następującej charakterystyce:

- Kontrakt – usługa ma jedną metodę, **DoWork**, która ma sygnaturę określoną w implementacji
- Adres – usługa ma adres relatywny w stosunku do aplikacji w której jest udostępniona, adresem punktu dostępowego jest adres zasobu \*.asmx. Ostatecznie adres to na przykład <http://localhost:12345/Service1.aspx>
- Wiązanie – usługa typu WebService jest dostępna za pomocą protokołu HTTPs, gdzie klient tworzy żądanie POST z dokumentem SOAP w określonej postaci i dostaje odpowiedź również w postaci SOAP.

#### 3.2 WSDL – Web Service Description Language

Interoperacyjny charakter wiązania usług typu SOAP pozwala dodatkowo wykorzystać protokół WSDL do pozyskania [metadanych](#) usługi. Przez metadane rozumie się tu formalny opis wszystkich metod wraz z adresami i wiązaniami, wyrażony w postaci XML.

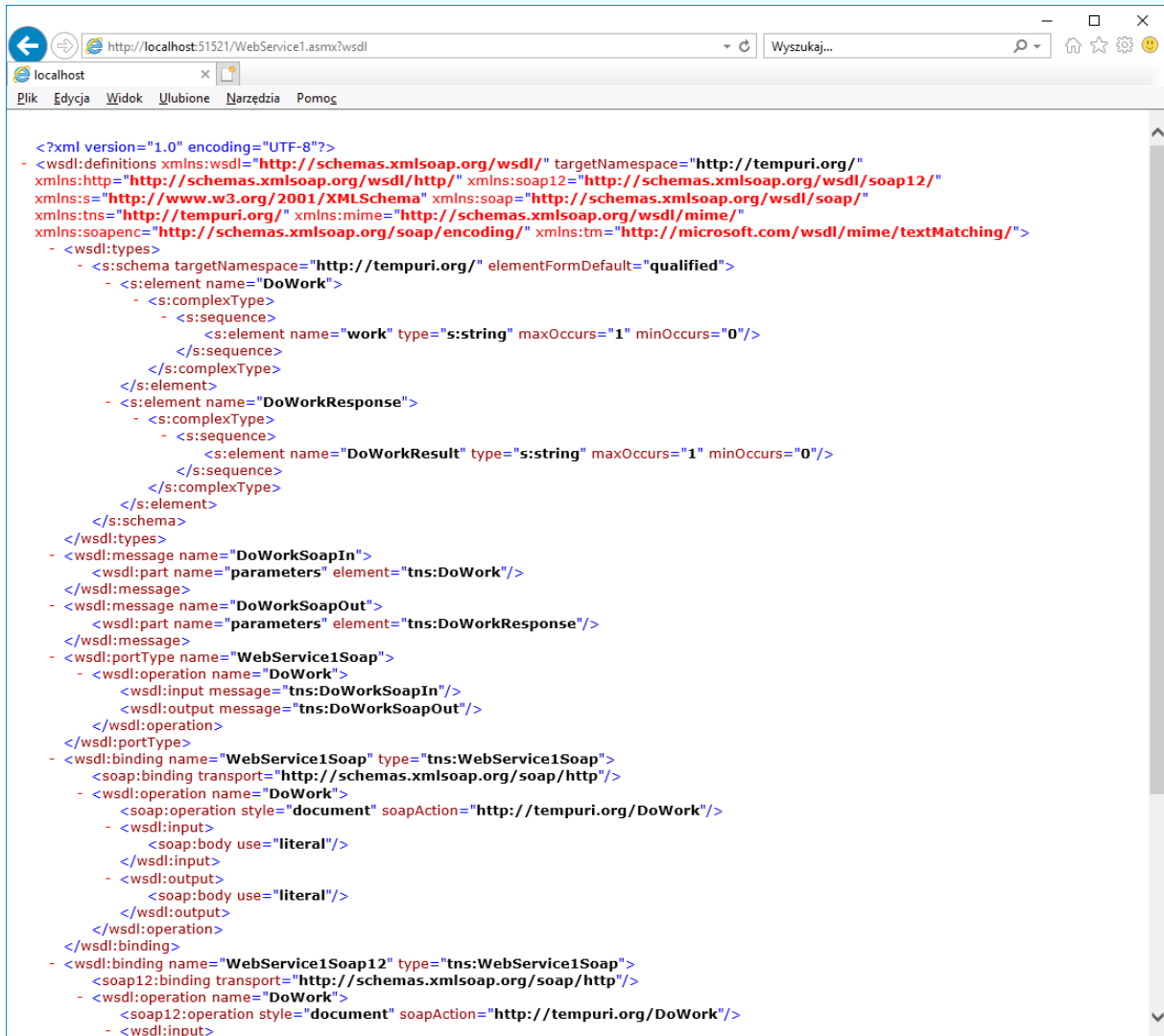
Ta postać jest rozumiana przez wiele technologii klienckich, istnieją również narzędzia dla architektów (np. Altova) w których istnieje możliwość projektowania usług natywnie na poziomie WSDL bez potrzeby tworzenia kodu.

W przypadku ASMX WebServices, dokument WSDL opisujący usługę jest domyślnie dostępny pod adresem

<http://adres.uslugi/service.asmx?wsdl>

gdzie sufiks ?wsdl należy połączyć z żądaniem typu GET.

Przykładowo dla usługi



```
<?xml version="1.0" encoding="UTF-8"?>
- <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://tempuri.org/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://tempuri.org/" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/">
  - <wsdl:types>
    - <s:schema targetNamespace="http://tempuri.org/" elementFormDefault="qualified">
      - <s:element name="DoWork">
        - <s:complexType>
          - <s:sequence>
            <s:element name="work" type="s:string" maxOccurs="1" minOccurs="0"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      - <s:element name="DoWorkResponse">
        - <s:complexType>
          - <s:sequence>
            <s:element name="DoWorkResult" type="s:string" maxOccurs="1" minOccurs="0"/>
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>
  - <wsdl:message name="DoWorkSoapIn">
    <wsdl:part name="parameters" element="tns:DoWork"/>
  </wsdl:message>
  - <wsdl:message name="DoWorkSoapOut">
    <wsdl:part name="parameters" element="tns:DoWorkResponse"/>
  </wsdl:message>
  - <wsdl:portType name="WebService1Soap">
    - <wsdl:operation name="DoWork">
      <wsdl:input message="tns:DoWorkSoapIn"/>
      <wsdl:output message="tns:DoWorkSoapOut"/>
    </wsdl:operation>
  </wsdl:portType>
  - <wsdl:binding name="WebService1Soap" type="tns:WebService1Soap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
    - <wsdl:operation name="DoWork">
      <soap:operation style="document" soapAction="http://tempuri.org/DoWork"/>
      - <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      - <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  - <wsdl:binding name="WebService1Soap12" type="tns:WebService1Soap">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
    - <wsdl:operation name="DoWork">
      <soap12:operation style="document" soapAction="http://tempuri.org/DoWork"/>
      - <wsdl:input>
```

Jak pokazano na poprzednim wykładzie, taki dokument metadanych może posłużyć do automatycznego wygenerowania kodu proxy dla klienta.

O usłudze ASMX WebServices powiemy więc żargonowo, że WSDL jest *artefaktem pochodnym* – potok implementacji usługi wygląda bowiem tak:

*kod C# implementacji usługi → pozyskanie WSDL z automatycznie wygenerowanych metadanych → wygenerowanie proxy klienta za pomocą wsdl.exe z metadanych WSDL*

### 3.3 WCF w klasycznym potoku wytwarzania

WCF pozwala na oparcie implementacji na zmodyfikowanym potoku wytwarzania, który pozwala na lepsze reużycie kodu.

Najpierw jednak wspomnijmy o możliwości zachowania dotychczasowego potoku.

W tym podejściu, usługa jest wytworzona przez wybranie w Visual Studio funkcji dodania nowej usługi. W przeciwieństwie do usługi typu ASMX WebService, generator dodaje nie tylko implementację usługi ale również interfejs który ona implementuje. W takim (domyślnym) przypadku, atrybuty po których środowisko uruchomieniowe rozpoznaje usługę (**ServiceContract** i **OperationContract**) są umieszczone na interfejsie.

```
public class Service2 : IService2
{
    public string DoWork(string work)
    {
        return "hello from WCF " + work;
    }
}

[ServiceContract]
public interface IService2
{
    [OperationContract]
    string DoWork( string work);
}
```

Bez poznania alternatywnego potoku (o czym za chwilę) można ulec pokusie wielu poradników, które sugerują możliwość zredukowania pary klasa – interfejs do jednego artefaktu – samej klasy:

```
[ServiceContract]
public class Service2
{
    [OperationContract]
    public string DoWork(string work)
    {
        return "hello from WCF " + work;
    }
}
```

W obu przypadkach – z dodatkowym interfejsem lub bez – usługa działa tak samo jak usługa ASMX WebService i w identyczny sposób pozwala pozyskać metadane WSDL:

```

<?xml version="1.0" encoding="UTF-8"?>
- <wsdl:definitions xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:wsa10="http://www.w3.org/2005/08/addressing" xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:msc="http://schemas.microsoft.com/ws/2005/12/wsdl/contract" xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
  xmlns:wsap="http://schemas.xmlsoap.org/ws/2004/08/addressing/policy" xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing" xmlns:tns="http://tempuri.org/"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://tempuri.org/" name="Service2">
  - <wsdl:types>
    - <xsd:schema targetNamespace="http://tempuri.org/Imports">
      <xsd:import namespace="http://tempuri.org/" schemaLocation="http://localhost:51521/Service2.svc?xsd=xsd0"/>
      <xsd:import namespace="http://schemas.microsoft.com/2003/10/Serialization/"
        schemaLocation="http://localhost:51521/Service2.svc?xsd=xsd1"/>
    </xsd:schema>
  </wsdl:types>
  - <wsdl:message name="Service2_DoWork_InputMessage">
    <wsdl:part name="parameters" element="tns:DoWork"/>
  </wsdl:message>
  - <wsdl:message name="Service2_DoWork_OutputMessage">
    <wsdl:part name="parameters" element="tns:DoWorkResponse"/>
  </wsdl:message>
  - <wsdl:portType name="Service2">
    - <wsdl:operation name="DoWork">
      <wsdl:input message="tns:Service2_DoWork_InputMessage" wsaw:Action="http://tempuri.org/Service2/DoWork"/>
      <wsdl:output message="tns:Service2_DoWork_OutputMessage" wsaw:Action="http://tempuri.org/Service2/DoWorkResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  - <wsdl:binding name="BasicHttpBinding_Service2" type="tns:Service2">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
    - <wsdl:operation name="DoWork">
      <soap:operation style="document" soapAction="http://tempuri.org/Service2/DoWork"/>
      - <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      - <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  - <wsdl:service name="Service2">
    - <wsdl:port name="BasicHttpBinding_Service2" binding="tns:BasicHttpBinding_Service2">
      <soap:address location="http://localhost:51521/Service2.svc"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>

```

### 3.4 WCF w „odwróconym” potoku wytwarzania

WCF pozwala jednak na odwrócenie potoku wytwarzania – w tym odwróconym potoku pierwszym wytwarzanym artefaktem nie jest implementacja ale właśnie interfejs, który jest **współdzielony między klientem a serwerem** (w postaci skompilowanej, \*.dll).

Potok w tym przypadku wygląda tak:

*interfejs C# opisujący kontrakt usługi*  
 (serwer) → *implementacja usługi* → (opcjonalne) *pozyskanie WSDL z automatycznie wygenerowanych metadanych*  
 (klient) → *wygenerowanie proxy klienta bez WSDL, bezpośrednio z interfejsu*

Różnice między klasycznym a odwróconym potokiem dotyczą głównie sposobu użycia usługi przez klienta – klient nie musi generować kodu proxy z WSDL, ale może pozyskać proxy bezpośrednio z interfejsu (oczywiście pod warunkiem że klient też jest napisany w .NET).

W tym odwróconym potoku definicję interfejsu:

```
[ServiceContract]
public interface IService2
{
    [OperationContract]
    string DoWork( string work);
}
```

należy wydzielić do osobnej biblioteki kodu i udostępnić klientowi. W rozdziale o wytwarzaniu kodu klienckiego omówione zostaną sposoby wykorzystania interfejsu do generowania proxy dla klienta.

## 4 Aktywacja WCF bez elementów deklaratywnych

WCF pozwala na aktywację usługi w kodzie bez elementów deklaratywnych (bez pliku \*.svc). Jest to możliwe dzięki znanej już technice routingu, w którym wykorzystany jest handler usług, a adres aktywacji może być dowolny.

Czytelnikowi sugeruje się zapoznanie z materiałem ilustrującym tę technikę:

[How to programmatically configure SSL for WCF](#)

[Dynamic routing and easy upgrade from ASMX to WCF](#)



## 5 Wytwarzanie kodu klienckiego dla WCF

Wytworzenie kodu proxy dla WCF możliwe jest w klasycznym potoku. Do wygenerowania klienta z metadanych WCF można użyć znanego już **wsdl.exe** (przeznaczonego pierwotnie dla usług ASMX WebServices) lub dedykowanego WCF **svcutil.exe**.

W odwróconym potoku można jednak wygenerować proxy bezpośrednio z interfejsu:

### 5.1 Użycie klasy ChannelFactory

Zaskakująco prostą metodą wygenerowania proxy dla klienta, przy założeniu dostępności interfejsu, jest użycie [ChannelFactory](#)

```
var address = new EndpointAddress("http://localhost:12345/Service2.svc");
var binding = new BasicHttpBinding();

var factory = new ChannelFactory<IService2>(binding);
var proxy = factory.CreateChannel(address);

string result = proxy.DoWork("test");
```

Ta metoda jest wyjątkowo zaskakująca, ponieważ fabryka wykonuje bardzo złożoną pracę:

- Konstruktor fabryki (tu: **CreateChannel<IService2>**) używa refleksji do inspekcji interfejsu, który jest tu argumentem generycznym. Kod proxy tworzony jest automatycznie dzięki mechanizmom dynamicznego tworzenia kodu i jest cacheowany do przyszłych użyć
- Metoda **CreateChannel** tworzy wystąpienie proxy dla zadanego interfejsem kontraktu, z wygenerowanego automatycznie kodu

Stąd, w tym podejściu, klasy proxy w ogóle nigdzie „nie widać”!

### 5.2 Użycie klasy ClientBase

Innym sposobem utworzenia proxy z interfejsu jest użycie **ClientBase** – jest to klasa której dziedziczenie daje dostęp do proxy:

```
public class Service2Proxy : ClientBase<IService2>, IService2
{
    public Service2Proxy( EndpointAddress address, BasicHttpBinding binding )
        : base( binding, address )
    {
    }

    public string DoWork(string work)
    {
        return this.Channel.DoWork(work);
    }
}
```

Zasada tworzenia klasy dziedziczącej z ClientBase jest następująca

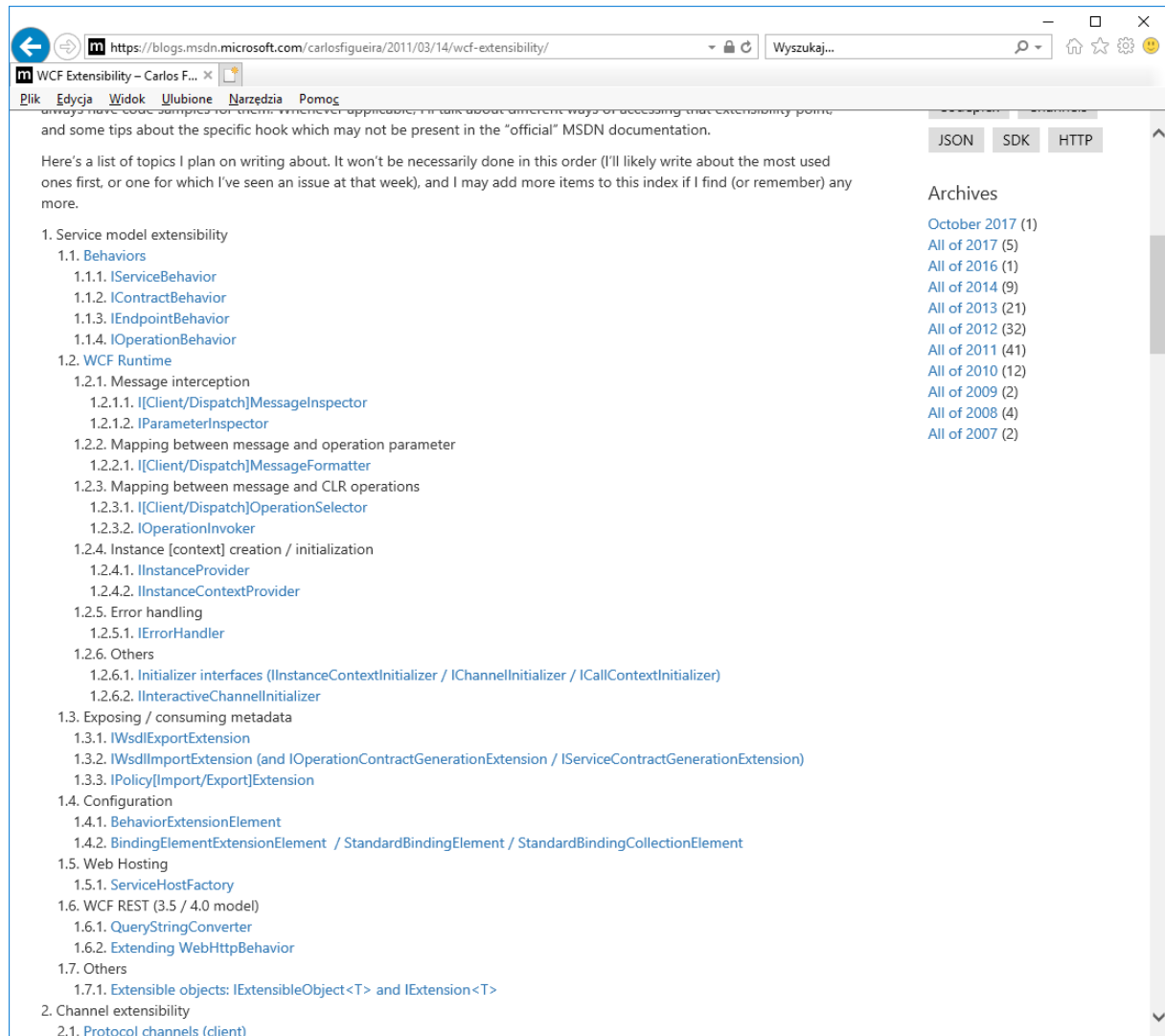
- należy w niej dziedziczyć **ClientBase<I>** gdzie I jest typem interfejsu (to daje dostęp do składowej **Channel** typu I
- należy w niej implementować interfejs I i w jego metodach *delegować* wywołania metod do **Channel**

Po co klient, mając *łatwe* generowanie proxy przez **ChannelFactory** miałby chcieć tworzyć implementację dziedziczącą z **ClientBase**? O tym w kolejnym podrozdziale.

## 6 Rozszerzenia WCF

Stos WCF może być rozszerzony w tak wielu miejscach, że potrzebna jest specjalna mapa rozszerzalności, dostępna pod adresem

<https://blogs.msdn.microsoft.com/carlosfigueira/2011/03/14/wcf-extensibility/>



The screenshot shows a web browser window with the URL <https://blogs.msdn.microsoft.com/carlosfigueira/2011/03/14/wcf-extensibility/>. The page content includes a list of topics planned for writing, categorized into two main sections: Service model extensibility and Channel extensibility. The Service model extensibility section is further divided into Behaviors, WCF Runtime, Exposing / consuming metadata, Configuration, Web Hosting, WCF REST, and Others. The Channel extensibility section includes Protocol channels (client).

- 1. Service model extensibility
  - 1.1. Behaviors
    - 1.1.1. IServiceBehavior
    - 1.1.2. IContractBehavior
    - 1.1.3. IEndpointBehavior
    - 1.1.4. IOperationBehavior
  - 1.2. WCF Runtime
    - 1.2.1. Message interception
      - 1.2.1.1. IClient/Dispatch]MessageInspector
      - 1.2.1.2. IParameterInspector
    - 1.2.2. Mapping between message and operation parameter
      - 1.2.2.1. IClient/Dispatch]MessageFormatter
    - 1.2.3. Mapping between message and CLR operations
      - 1.2.3.1. IClient/Dispatch]OperationSelector
      - 1.2.3.2. IOperationInvoker
    - 1.2.4. Instance [context] creation / initialization
      - 1.2.4.1. IInstanceProvider
      - 1.2.4.2. IInstanceContextProvider
    - 1.2.5. Error handling
      - 1.2.5.1. IErrorHandler
    - 1.2.6. Others
      - 1.2.6.1. Initializer interfaces (IInstanceContextInitializer / IChannelInitializer / ICallContextInitializer)
      - 1.2.6.2. IInteractiveChannelInitializer
  - 1.3. Exposing / consuming metadata
    - 1.3.1. IWsdExportExtension
    - 1.3.2. IWsdImportExtension (and IOperationContractGenerationExtension / IServiceContractGenerationExtension)
    - 1.3.3. IPolicy[Import/Export]Extension
  - 1.4. Configuration
    - 1.4.1. BehaviorExtensionElement
    - 1.4.2. BindingElementExtensionElement / StandardBindingElement / StandardBindingCollectionElement
  - 1.5. Web Hosting
    - 1.5.1. ServiceHostFactory
  - 1.6. WCF REST (3.5 / 4.0 model)
    - 1.6.1. QueryStringConverter
    - 1.6.2. Extending WebHttpBehavior
  - 1.7. Others
    - 1.7.1. Extensible objects: IExtensibleObject<T> and IExtension<T>
- 2. Channel extensibility
  - 2.1. Protocol channels (client)

Rozszerzenia możliwe są zarówno na kliencie jak i na serwerze, przy czym rozszerzenia na kliencie w większości przypadków **wymagają** proxy dziedziczącego z **ClientBase**.

Dla współdzielonego interfejsu

```
[ServiceContract]
public interface TheInterface
{
    [OperationContract]
    string DoWork( string Work );
}
```

możliwe są przykładowe następujące rozszerzenia klienta dodające *inspekcję* wychodzących i wchodzących komunikatów:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.Text;
using System.Threading.Tasks;
using Interface;

namespace Client
{
    class Program
    {
        static void Main( string[] args )
        {
            ChannelFactoryDemo();
            ClientBaseDemo();

            Console.ReadLine();
        }

        /// <summary>
        /// Do uzyskania proxy wykorzystany jest ChannelFactory
        /// </summary>
        private static void ChannelFactoryDemo()
        {
            var factory = new ChannelFactory<TheInterface>( new BasicHttpBinding
            ( ) );

            var address = new EndpointAddress( "http://localhost:12345/TheService" );

            var client = factory.CreateChannel( address );
            Console.WriteLine( client.DoWork( " foo " ) );
            ( client as IDisposable ).Dispose();
        }

        /// <summary>
        /// Do uzyskania proxy wykorzystany jest ClientBase (tu: napisany ręcznie)
        /// Plus: można korzystać z rozszerzeń
        /// </summary>
        private static void ClientBaseDemo()
        {
            var address = new EndpointAddress( "http://localhost:12345/TheService" );

            using ( var client = new TheInterfaceProxy( new BasicHttpBinding(),
            address ) )
            {
                client.Endpoint.EndpointBehaviors.Add( new InspectorBehavior() );

                Console.WriteLine( client.DoWork( " bar " ) );
            }
        }
    }
}

```

```

    }

    public class TheInterfaceProxy : ClientBase<TheInterface>, TheInterface
    {
        public TheInterfaceProxy( Binding binding, EndpointAddress address ) : base( binding, address ) { }

        #region TheInterface Members

        public string DoWork( string Work )
        {
            return this.Channel.DoWork( Work );
        }

        #endregion
    }

    /// <summary>
    /// Podstawowa infrastruktura rozszerzania
    /// </summary>
    class InspectorBehavior : IEndpointBehavior
    {
        #region IEndpointBehavior Members

        public void AddBindingParameters( ServiceEndpoint endpoint, System.ServiceModel.Channels.BindingParameterCollection bindingParameters )
        {
        }

        public void ApplyClientBehavior( ServiceEndpoint endpoint, System.ServiceModel.Dispatcher.ClientRuntime clientRuntime )
        {
            clientRuntime.ClientMessageInspectors.Add( new DispatchInspector() );
        };

        public void ApplyDispatchBehavior( ServiceEndpoint endpoint, System.ServiceModel.Dispatcher.EndpointDispatcher endpointDispatcher )
        {
        }

        public void Validate( ServiceEndpoint endpoint )
        {
        }

        #endregion
    }

    /// <summary>
    /// Przykładowe rozszerzenie - inspektor wiadomości
    /// </summary>
    class DispatchInspector : IClientMessageInspector
    {
        #region IClientMessageInspector Members

        public void AfterReceiveReply( ref Message reply, object correlationState )
        {
            MessageBuffer buffer = reply.CreateBufferedCopy( Int32.MaxValue );

```

```
        reply = buffer.CreateMessage();
        Console.WriteLine( "Receiving:\n{0}", buffer.CreateMessage().ToString() );
    }

    public object BeforeSendRequest( ref Message request, IClientChannel channel )
    {
        MessageBuffer buffer = request.CreateBufferedCopy( Int32.MaxValue );
        request = buffer.CreateMessage();
        Console.WriteLine( "Sending:\n{0}", buffer.CreateMessage().ToString() );

        return null;
    }

    #endregion
}
}
```

## 7 Hostowanie WCF poza serwerem aplikacyjnym

Rozszerzenia serwera są symetryczne do rozszerzeń klienta. Zademonstrowane zostaną na przykładzie, w którym usługa WCF jest hostowana w aplikacji konsolowej. Jest to możliwe dzięki użyciu hosta [ServiceHost](#)

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.IO;
using System.Linq;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.Text;
using System.Threading.Tasks;
using System.Xml;
using Interface;

namespace Service
{
    class Program
    {
        static void Main( string[] args )
        {
            using ( ServiceHost host = new ServiceHost( typeof( TheImplementatio
n ), new Uri( "http://localhost:12345/TheService" ) ) )
            {
                var binding = new BasicHttpBinding();
                var address = new Uri( "http://localhost:12345/TheService" );

                var endpoint = host.AddServiceEndpoint( typeof( TheInterface ),
binding, address );
                //endpoint.EndpointBehaviors.Add( new InspectorBehavior() );

                // opcjonalnie - WSDL
                host.Description.Behaviors.Add( new ServiceMetadataBehavior() {
HttpGetEnabled = true } );
                host.Description.Behaviors.Remove( typeof( ServiceDebugBehavior
) );
                host.Description.Behaviors.Add( new ServiceDebugBehavior() { Inc
ludeExceptionDetailInFaults = true } );

                host.Open();

                Console.WriteLine( "Host listens" );
                Console.ReadLine();
            }
        }
    }

    /// <summary>
    /// Podstawowa infrastruktura rozszerzania
    /// </summary>
    class InspectorBehavior : IEndpointBehavior
    {
```

```

        #region IEndpointBehavior Members

        public void AddBindingParameters( ServiceEndpoint endpoint, System.ServiceModel.Channels.BindingParameterCollection bindingParameters )
        {
        }

        public void ApplyClientBehavior( ServiceEndpoint endpoint, System.ServiceModel.Dispatcher.ClientRuntime clientRuntime )
        {
        }

        public void ApplyDispatchBehavior( ServiceEndpoint endpoint, System.ServiceModel.Dispatcher.EndpointDispatcher endpointDispatcher )
        {
            endpointDispatcher.DispatchRuntime.MessageInspectors.Add( new DispatchInspector() );
        }

        public void Validate( ServiceEndpoint endpoint )
        {
        }

        #endregion
    }

    [AttributeUsage( AttributeTargets.Class )]
    public class InspectorServiceBehaviorAttribute : Attribute, IServiceBehavior
    {
        public void AddBindingParameters( ServiceDescription serviceDescription, ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints, BindingParameterCollection bindingParameters )
        {
        }

        public void ApplyDispatchBehavior( ServiceDescription serviceDescription, ServiceHostBase serviceHostBase )
        {
            for ( int i = 0; i < serviceHostBase.ChannelDispatchers.Count; i++ )
            {
                ChannelDispatcher channelDispatcher = serviceHostBase.ChannelDispatchers[i] as ChannelDispatcher;
                if ( channelDispatcher != null )
                {
                    foreach ( EndpointDispatcher endpointDispatcher in channelDispatcher.Endpoints )
                    {
                        endpointDispatcher.DispatchRuntime.MessageInspectors.Add( new DispatchInspector() );
                    }
                }
            }
        }

        public void Validate( ServiceDescription serviceDescription, ServiceHostBase serviceHostBase )
        {
        }
    }

```



```

/// <summary>
/// Przykładowe rozszerzenie - inspektor wiadomości
/// </summary>
class DispatchInspector : IDispatchMessageInspector
{
    #region IDispatchMessageInspector Members

    public object AfterReceiveRequest(
        ref System.ServiceModel.Channels.Message request,
        IClientChannel channel, InstanceContext instanceContext )
    {
        MessageBuffer buffer = request.CreateBufferedCopy( Int32.MaxValue );
        request = buffer.CreateMessage();
        Console.WriteLine( "Received:\n{0}", buffer.CreateMessage().ToString
() );

        //return null;
        // */

        MemoryStream ms = new MemoryStream();
        XmlWriter writer = XmlWriter.Create( ms );
        request.WriteMessage( writer ); // the message was consumed here
        writer.Flush();
        ms.Position = 0;
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.Load( ms );
        ms.Dispose();

        //Now recreating the message
        //ms = new MemoryStream();
        //xmlDoc.Save( ms );
        //ms.Position = 0;

        XmlNamespaceManager bodyMgr = new XmlNamespaceManager(xmlDoc.NameTab
le);
        bodyMgr.AddNamespace( "s", "http://schemas.xmlsoap.org/soap/envelope
/" );

        XmlNode bodyNode = xmlDoc.SelectSingleNode( "//s:Envelope/s:Body", b
odyMgr );
        this.ChangeNode( bodyNode );

        //StreamReader sr = new StreamReader( ms );
        //string body = sr.ReadToEnd();

        //ms.Position = 0;
        //XmlReader reader = XmlReader.Create( ms );
        //Message newMessage = Message.CreateMessage( reader, int.MaxValue,
request.Version );
        //newMessage.Properties.CopyProperties( request.Properties );

        //Console.WriteLine( "body\r\n" + body );

        Message newMessage = Message.CreateMessage( request.Version, request
.Headers.Action, bodyNode.FirstChild );

        request = newMessage;
//         request = newMessage;

```

```

    /*
    using ( MemoryStream msin = new MemoryStream() )
    using ( MemoryStream msout = new MemoryStream() )
    {
        XmlWriter writer = XmlWriter.Create( msin );
        request.WriteMessage( writer ); // the message was consumed here
        writer.Flush();

        msin.Position = 0;
        XmlDocument xmlDoc = new XmlDocument();
        //xmlDoc.PreserveWhitespace = false;
        xmlDoc.Load( msin );

        //Now recreating the message
        xmlDoc.Save( msout );
        msout.Position = 0;
        XmlReader reader = XmlReader.Create( msout );
        Message newMessage = Message.CreateMessage( reader, int.MaxValue
, request.Version );
        newMessage.Properties.CopyProperties( request.Properties );

        request = newMessage;
    }
    */

    return null;
}

private void ChangeMessage( XmlDocument doc )
{
    if ( doc == null ) return;

    ChangeNode( doc.DocumentElement );
}

private void ChangeNode( XmlNode node )
{
    if ( node == null ) return;
    if ( node.NodeType == XmlNodeType.Text ) node.InnerText = node.InnerText.Trim();

    foreach ( XmlNode childNode in node.ChildNodes )
        ChangeNode( childNode );
}

public void BeforeSendReply( ref System.ServiceModel.Channels.Message reply, object correlationState )
{
    MessageBuffer buffer = reply.CreateBufferedCopy( Int32.MaxValue );
    reply = buffer.CreateMessage();
    Console.WriteLine( "Sending:\n{0}", buffer.CreateMessage().ToString(
) );
}

#endregion
}
}

```