

# Projektowanie aplikacji ASP.NET

## Wykład 10/15

### Usługi SOAP - WCF

Wiktor Zychla 2024/2025

---

#### Spis treści

1	Wprowadzenie .....	2
2	Kontrakt + adres + wiązanie (Contract + Address + Binding) .....	3
2.1	ASMX WebServices.....	3
2.2	WSDL – Web Service Description Language.....	3
2.3	WCF w klasycznym potoku wytwarzania.....	5
2.4	WCF w „odwróconym” potoku wytwarzania .....	6
3	Aktywacja WCF bez elementów deklaracyjnych .....	8
4	Wytwarzanie kodu klienckiego dla WCF .....	9
4.1	Użycie klasy ChannelFactory .....	9
4.2	Użycie klasy ClientBase.....	9
5	Hostowanie WCF poza serwerem aplikacyjnym .....	11
6	Rozszerzenia WCF.....	15
7	Autentykacja / autoryzacja.....	21
7.1	Autentykacja za pomocą ciastka .....	21
7.2	Autentykacja za pomocą nagłówka .....	22
8	Usługi WCF w .NET.Core.....	27
8.1	Współdzielony kontrakt .....	27
8.2	Serwer .....	27
8.3	Klient.....	28
9	Przykład z życia .....	30

# 1 Wprowadzenie

Usługi [SOAP](#) to zbiorcze określenie takiego sposobu budowania komunikacji między klientem a serwerem, w którym

- Protokołem komunikacyjnym jest http
- Językiem formatowania żądań i odpowiedzi jest XML (w dialekcie SOAP)
- Zapytania klienta są zawsze przesyłane jako POST
- Opis usługi (metadane) może być wyspecyfikowany za pomocą standardu [WSDL](#)
- Istnieją standaryzacje rozszerzające (zbiorczo nazwane [WS-\\*](#)) dodające wsparcie m.in. dla mechanizmów zabezpieczeń czy transakcji

Tego typu usługi są chętniej wybierane w komunikacji między serwerami, w przeciwieństwie do usług typu REST, które są chętniej wybierane do komunikacji klient (przeglądarka) – serwer.

Podsystem **Windows Communication Foundation** ma w założeniach eliminować ograniczenia podsystemu **ASMX WebServices**, dostępnego w .NET.Framework od wersji 1.0.

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

## 2 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

### 2.1 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.

### 2.2 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.

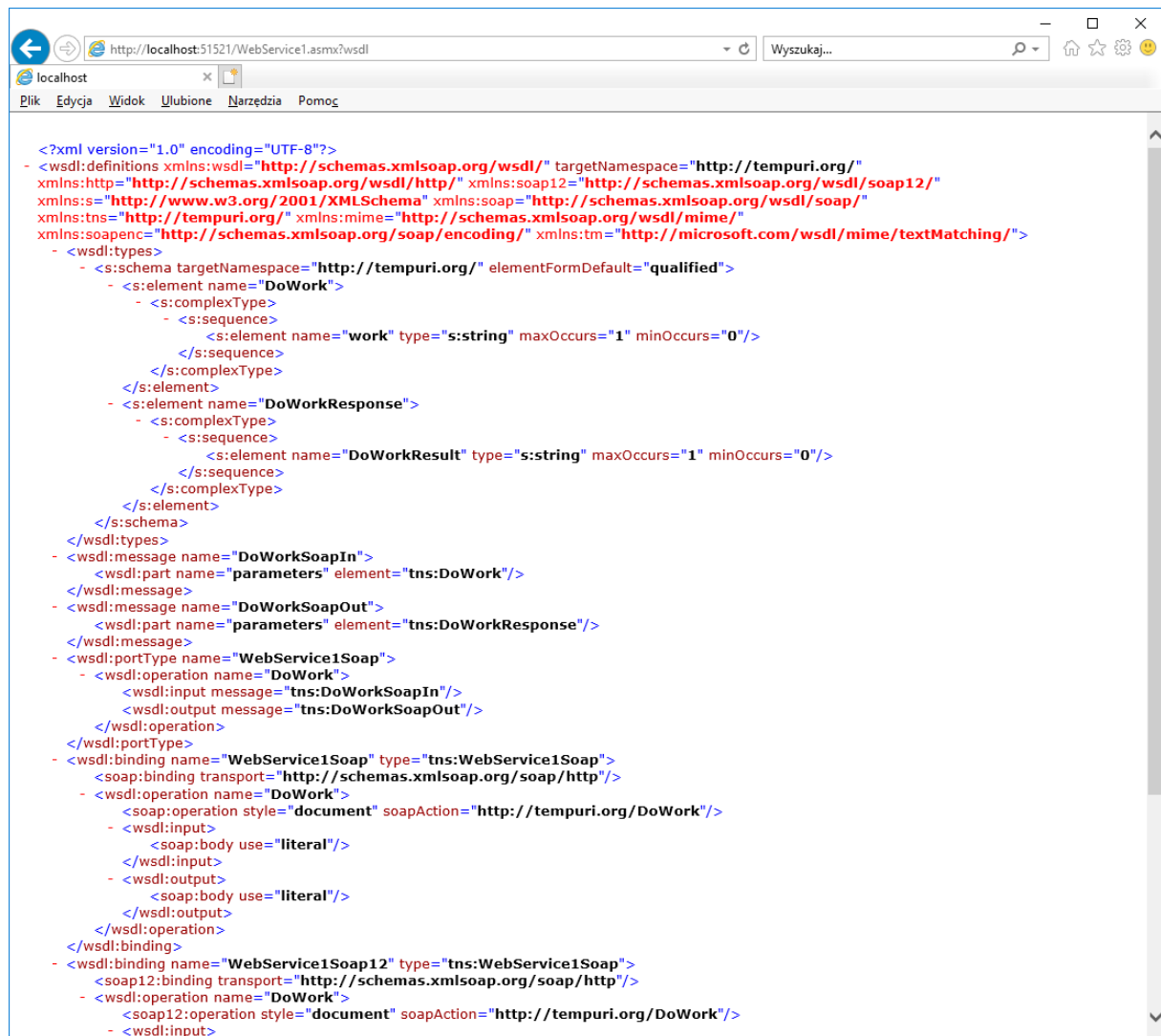
Ta postać jest rozumiana przez wiele technologii klienckich, istnieją również narzędzia dla architektów (np. Altova) w których istnieje możliwość projektowania usług natywnie na poziomie WSDL bez potrzeby tworzenia kodu.

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<sup>1</sup> implementacji usługi wygląda bowiem tak:

- (1) kod C# implementacji usługi →
- (2) pozyskanie WSDL z automatycznie wygenerowanych metadanych →
- (3) wygenerowanie proxy klienta za pomocą **wsdl.exe** z metadanych WSDL

<sup>1</sup> Przez *potok* rozumiemy tu sekwencję czynności wykonywaną przez dewelopera w celu uzyskania działającej implementacji

## 2.3 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.

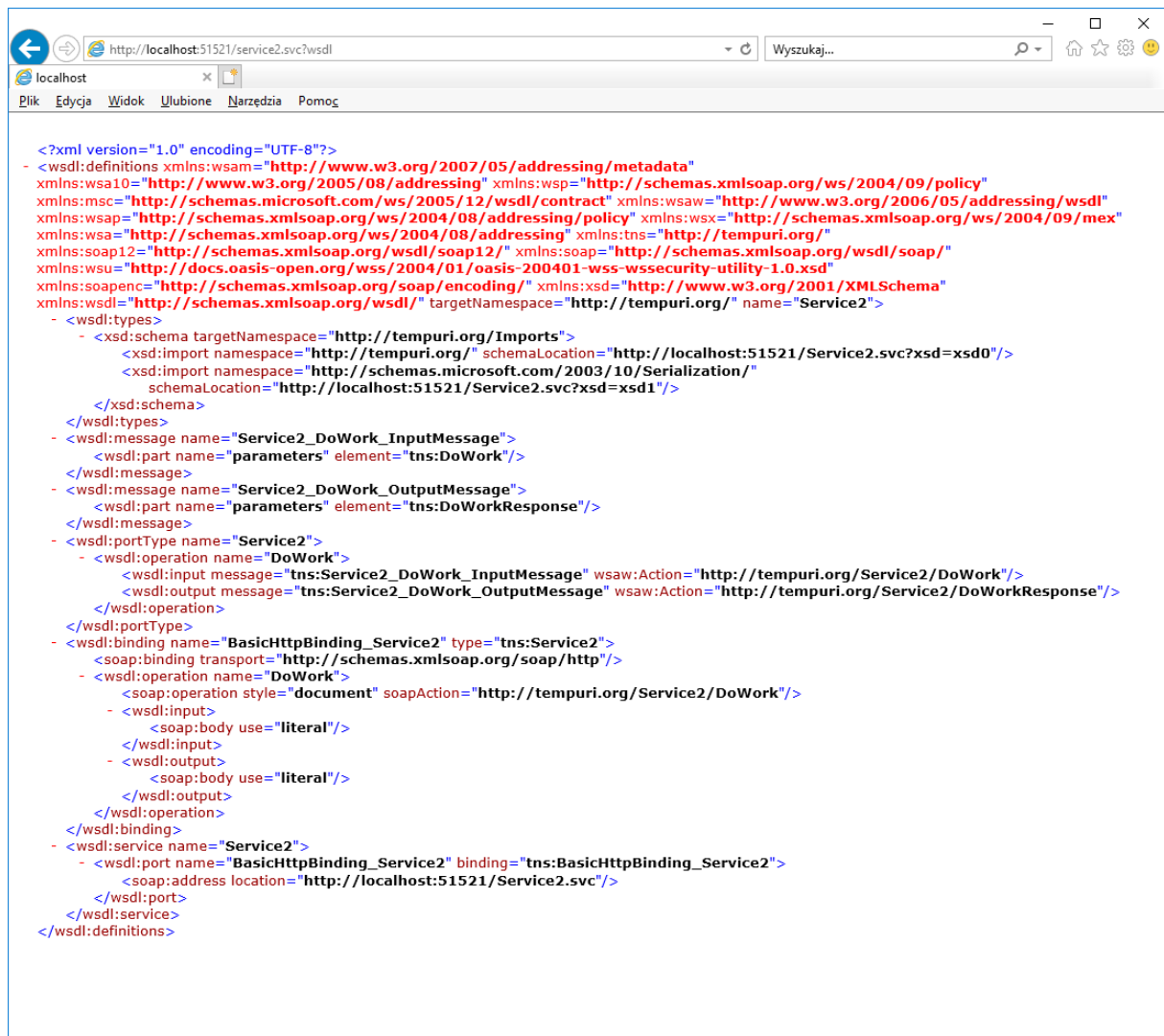
```
public class Service2 : IService2
{
    public string DoWork(string work)
    {
        return "hello from WCF " + work;
    }
}

[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>
```

## 2.4 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:

*(wspólne) 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.

### 3 Aktywacja WCF bez elementów deklaracyjnych

WCF pozwala na aktywację usługi w kodzie bez elementów deklaracyjnych (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](#)



## 4 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:

### 4.1 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 zaskakująco łatwa, a 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 interfejsu kontraktu, z wygenerowanego automatycznie kodu

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

### 4.2 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.

## 5 Hostowanie WCF poza serwerem aplikacyjnym

Rozszerzenia serwera są symetryczne do rozszerzeń klienta. Zadeemonstrowane 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() );
    }

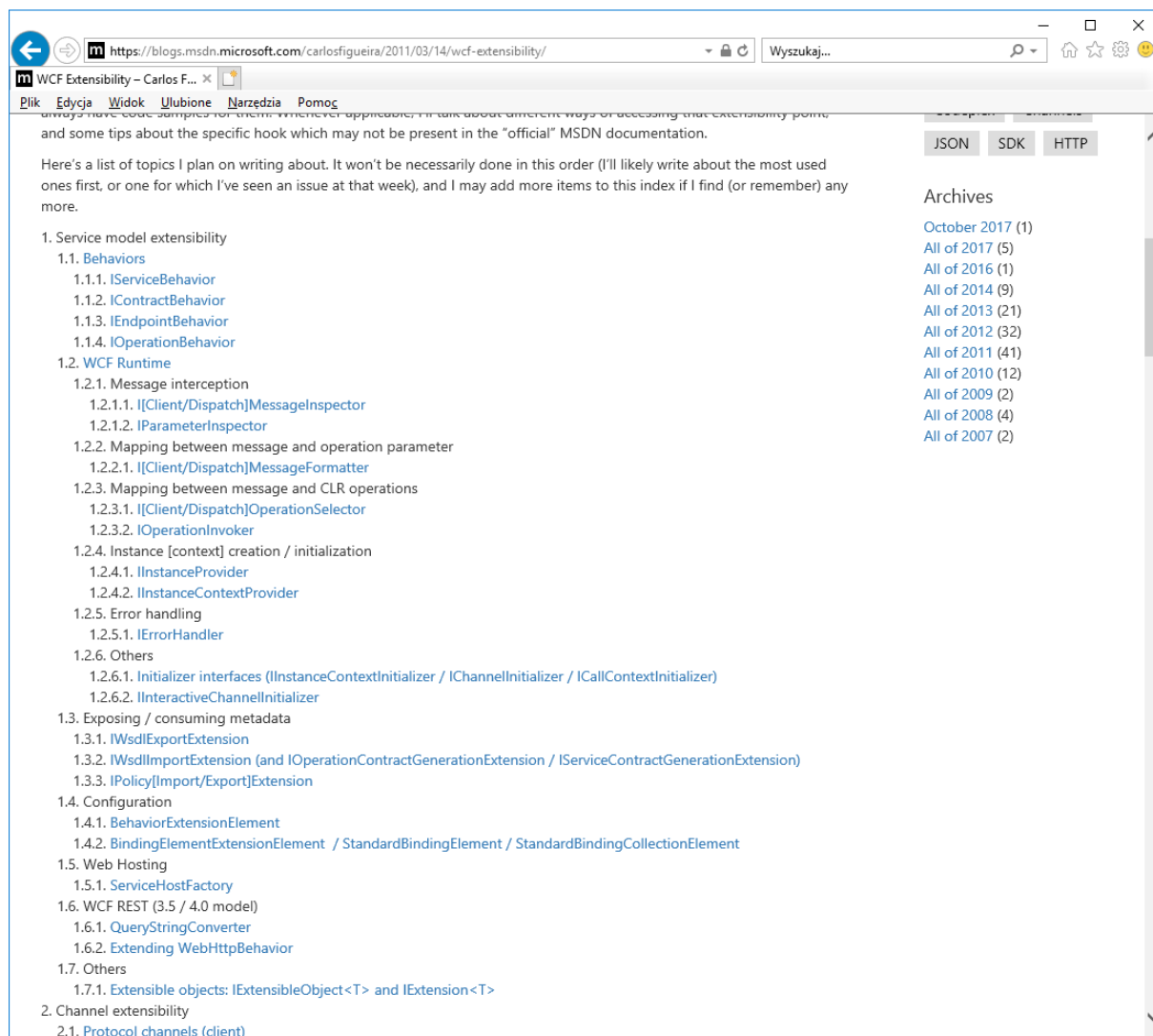
    #endregion
}
}

```

## 6 Rozszerzenia WCF

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

<https://docs.microsoft.com/en-us/archive/blogs/carlosfigueira/wcf-extensibility>



Rozszerzenia możliwe są zarówno na kliencie jak i na serwerze, przy czym rozszerzenia na kliencie w większości przypadków **wymagają** proxy dziedziczącego z **ClientBase** – w przypadku proxy wykorzystującego **ChannelFactory** takiej możliwości nie ma.

**Przykład.** Dla współdzielonego interfejsu

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

możliwe są następujące rozszerzenia klienta dodające *inspekcję* wychodzących i wchodzących komunikatów. O tych obiektach dodawanych do potoku, które pozwalają na inspekcję wiadomości czy to po stronie klienta czy po stronie serwera mówi się żargonowo **inspektorzy** (*inspectors*).

W poniższym przykładzie klienta proszę zwrócić uwagę na dodanie inspektora w linii:

**client.Endpoint.EndpointBehaviors.Add( new InspectorBehavior() );**

```
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 ) : b
ase( 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.Servi
ceModel.Channels.BindingParameterCollection bindingParameters )
        {
        }

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

        public void ApplyDispatchBehavior( ServiceEndpoint endpoint, System.Serv
iceModel.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 correlationStat
e )
        {
            MessageBuffer buffer = reply.CreateBufferedCopy( Int32.MaxValue );
            reply = buffer.CreateMessage();
            Console.WriteLine( "Receiving:\n{0}", buffer.CreateMessage().ToStrin
g() );
        }

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

            return null;
        }

        #endregion
    }
}

```

Opowiadający klientowi kod serwera również może posiadać własne inspektory:

```

[ServiceBehavior( ConcurrencyMode = ConcurrencyMode.Multiple,
InstanceContextMode = InstanceContextMode.PerCall )]
public class TheImplementation : TheInterface
{
    #region TheInterface Members

    public string DoWork( string Work )
    {
        return string.Format( "{0} from WCF!", Work );
    }

    #endregion
}

class Program
{
    static void Main( string[] args )
    {
        using ( ServiceHost host = new ServiceHost( typeof(
TheImplementation ), 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 - behavior

        // opcjonalnie - WSDL
        host.Description.Behaviors.Add( new ServiceMetadataBehavior()
{ HttpGetEnabled = 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
}

/// <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;
    }

    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
}

```

Proszę zwrócić uwagę na pewną ładną symetrię. Zarówno klient jak i serwer używają interfejsu **IEndpointBehavior**.

Klient używa metody **ApplyClientBehavior** z tego interfejsu, w której dodaje inspektora typu **IClientMessgelInspector**. Ten inspektor ma metody **BeforeSendRequest** i **AfterReceiveReply** – to ma sens, klient wysyła żądanie i dostaje odpowiedź.

Serwer używa metody **ApplyDispatchBehavior** z tego interfejsu, w której dodaje inspektora typu **IDispatchMessageInspector**. Ten inspektor ma metody **AfterReceiveRequest** i **BeforeSendReply** – to też ma sens, bo serwer dostaje żądanie i wysyła odpowiedź.

## 7 Autentykacja / autoryzacja

Poznając ASP.NET mieliśmy okazję obejrzeć kilka różnych podejść do autentykacji / autoryzacji. Większość z nich sklasyfikowalibyśmy jako *deklaratywne* tzn. stosowne reguły były albo wprost częścią konfiguracji (WebForms i sekcje autoryzacji w **web.config**) albo metadanych (MVC/WebAPI i atrybuty **[Authorize]** nad kontrolerami/metodami).

WCF ma dość [rozbudowany model uwierzytelniania](#) ponieważ implementuje standard WS-\*. Sposobów na zabezpieczenie usługi jest więc całkiem sporo i warto przeglądnąć dokumentację i [załączone przykłady](#).

Ze swojej strony chciałbym zwrócić uwagę na dwie nietrudne możliwości

### 7.1 Autentykacja za pomocą ciastka

Jeżeli klient posiada ciastko autentykacji (które normalnie po stronie serwera jest obsługiwane przez moduł autentykacji) to najprostszym sposobem sprawdzenia czy żądanie do usługi WCF ma poprawne ciastko jest kod imperatywny w konstruktorze usługi WCF. Technika którą zastosujemy polega na użyciu klasy [PrincipalPermission](#) i za jej pomocą można osiągnąć dokładnie taki sam efekt jak za pomocą np. atrybutu **Authorize** – określa się jakie role (i czy w ogóle jakieś) ma spełniać obiekt **Principal** a następnie za pomocą **Demand()** sprawdza się czy takie wymagania są spełnione. Jeśli nie – wyrzucany jest wyjątek który trzeba obsłużyć.

```
public WCFService()
{
    // kontekst zaautentykowanego użytkownika do wątku
    Thread.CurrentPrincipal = HttpContext.Current.User;

    //najpierw sprawdzamy czy klient jest zaautentykowany
    PrincipalPermission pAny = new PrincipalPermission( null, null, true
);
    try
    {
        pAny.Demand();
    }
    catch (SecurityException ex)
    {
        throw ...
    }

    //następnie czy posiada wymagane uprawnienie (rolę)
    //w przykładzie pokazano jak sprawdzić dwie role
    PrincipalPermission pAnk = new PrincipalPermission( null,
Rola.Operator, true );
    PrincipalPermission pAdm = new PrincipalPermission( null,
Rola.Administrator, true );

    IPermission pComposite = pAnk.Union( pAdm );

    try
    {
        pComposite.Demand();
    }
    catch( SecurityException ex )
    {

```

```
        throw ...  
    }  
}
```

## 7.2 Autentykacja za pomocą nagłówka

Jeśli nośnikiem transportowym jest http, to autentykacja może być oparta o istnienie nagłówka autentykującego, na przykład w standardzie [Basic Authentication](#).

Do dodania nagłówka od strony klienta użyjemy inspektora (proszę zerknąć do rozdziału opisującego rozszerzenia żeby przypomnieć sobie czym jest inspektor).

Inspektor klienta – dodaje nagłówek:

```
/// <summary>  
/// Client inspector WCF który dodaje nagłówek basic auth  
/// </summary>  
public class BasicAuthenticationClientInspector : IClientMessageInspector  
{  
    public BasicAuthenticationClientInspector(string UsernamePassword)  
    {  
        this._usernamepassword = UsernamePassword;  
    }  
  
    private const string AUTHORIZATION_HEADER = "Authorization";  
  
    private string _usernamepassword;  
  
    private string UsernamePasswordHeaderValue  
    {  
        get  
        {  
            string token =  
Convert.ToBase64String(Encoding.GetEncoding("iso-8859-  
1").GetBytes(this._usernamepassword));  
            string header = string.Format("Basic {0}", token);  
  
            return header;  
        }  
    }  
  
    public virtual object BeforeSendRequest(ref  
System.ServiceModel.Channels.Message request,  
System.ServiceModel.IClientChannel channel)  
    {  
        HttpRequestMessageProperty httpRequestMessage;  
        object httpRequestMessageObject;  
        if  
(request.Properties.TryGetValue(HttpRequestMessageProperty.Name, out  
httpRequestMessageObject))  
        {  
            httpRequestMessage = httpRequestMessageObject as  
HttpRequestMessageProperty;  
            if  
(string.IsNullOrEmpty(httpRequestMessage.Headers[AUTHORIZATION_HEADER]))  
            {  
                httpRequestMessage.Headers[AUTHORIZATION_HEADER] =  
this.UsernamePasswordHeaderValue;  
            }  
        }  
    }  
}
```

```

        }
    }
    else
    {
        httpRequestMessage = new HttpRequestMessageProperty();
        httpRequestMessage.Headers.Add(AUTHORIZATION_HEADER,
this.UsernamePasswordHeaderValue);
        request.Properties.Add(HttpRequestMessageProperty.Name,
httpRequestMessage);
    }
    return null;
}

public virtual void AfterReceiveReply(ref Message reply, object
correlationState)
{
    // intencjonalnie puste
}
}

```

Inspektor serwera odczytuje nagłówki:

```

/// <summary>
/// Server inspector WCF który odczytuje nagłówek basic auth
/// </summary>
public class BasicAuthenticationServerInspector :
IDispatchMessageInspector
{
    private const string AUTHORIZATION_HEADER = "Authorization";

    public virtual object AfterReceiveRequest(ref Message request,
IClientChannel channel, InstanceContext instanceContext)
    {
        HttpRequestMessageProperty httpRequestMessage;
        object httpRequestMessageObject;
        if
(request.Properties.TryGetValue(HttpRequestMessageProperty.Name, out
httpRequestMessageObject))
        {
            httpRequestMessage = httpRequestMessageObject as
HttpRequestMessageProperty;
            if
(!string.IsNullOrEmpty(httpRequestMessage.Headers[AUTHORIZATION_HEADER]))
            {
                BasicAuthenticationOperationContext.Current.Raw =
httpRequestMessage.Headers[AUTHORIZATION_HEADER];
            }
        }

        return null;
    }

    public virtual void BeforeSendReply(ref Message reply, object
correlationState)
    {
    }
}

/// <summary>

```

```

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

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

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

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

    public virtual void Validate(ServiceEndpoint endpoint)
    {
    }

    #endregion
}

/// <summary>
/// Fasada na "Items" w OperationContext, potrzebna na serwerze
/// </summary>
public class BasicAuthenticationOperationContext :
IExtension<OperationContext>
{
    /// <summary>
    /// Zakodowane wg. specyfikacji nagłówka
    /// </summary>
    public string Raw { get; set; }

    /// <summary>
    /// Rozkodowane wg specyfikacji nagłówka
    /// </summary>
    public string Authorization
    {
        get
        {
            try
            {
                if (!string.IsNullOrEmpty(this.Raw))
                {
                    var authHeaderVal =
AuthenticationHeaderValue.Parse(this.Raw);

                    if (authHeaderVal.Scheme.Equals("basic",
StringComparison.OrdinalIgnoreCase) &&
authHeaderVal.Parameter !=
null)
                {

```



```

        var encoding = Encoding.GetEncoding("iso-8859-1");
        var decoded = encoding.GetString(Convert.FromBase64String(authHeaderVal.Parameter));

        return decoded;
    }
    // fallback
    return string.Empty;
}
catch
{
    return string.Empty;
}
}

private BasicAuthenticationOperationContext()
{
}

public static BasicAuthenticationOperationContext Current
{
    get
    {
        var context = OperationContext.Current.Extensions.Find<BasicAuthenticationOperationContext>();
        if (context == null)
        {
            context = new BasicAuthenticationOperationContext();
            OperationContext.Current.Extensions.Add(context);
        }
        return context;
    }
}

public virtual void Attach(OperationContext owner) { }
public virtual void Detach(OperationContext owner) { }
}

```

Żeby użyć inspektora po stronie klienta, należałoby go po prostu dodać do obiektu **ClientBase**

```

using ( var client = new TheInterfaceProxy( new BasicHttpBinding(),
address ) )
{
    client.Endpoint.EndpointBehaviors.Add( new
BasicAuthenticationClientInspector("username:password") );

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

```

Żeby po stronie serwera odczytać wartość nagłówka, należy w implementacji metody odczytać wartość **BasicAuthenticationOperationContext.Current.Authorization**, np.:

```
[ServiceBehavior( ConcurrencyMode = ConcurrencyMode.Multiple,
InstanceContextMode = InstanceContextMode.PerCall )]
public class TheImplementation : TheInterface
{
    #region TheInterface Members

    public string DoWork( string Work )
    {
        string header =
BasicAuthenticationOperationContext.Current.Authorization;
        return string.Format( "{0} from WCF!", Work );
    }

    #endregion
}
```

## 8 Usługi WCF w .NET.Core

### 8.1 Współdzielony kontrakt

Współdzielony kontrakt wymaga referencji do **System.ServiceModel.Primitives**. Z tego pakietu pochodzą atrybuty dekorujące model i kontrakt usługi.

```
/// <summary>
/// Model dla żądania
/// </summary>
[DataContract]
public class CustomRequestModel
{
    [DataMember]
    public string RequestValue { get; set; }
}

/// <summary>
/// Model dla odpowiedzi
/// </summary>
[DataContract]
public class CustomResponseModel
{
    [DataMember]
    public string ResponseValue { get; set; }
}

/// <summary>
/// Interfejs usługi
/// </summary>
[ServiceContract]
public interface ITestService
{
    [OperationContract]
    CustomResponseModel DoWork( CustomRequestModel request );
}
```

### 8.2 Serwer

Serwer wymaga referencji do pakietów **CoreWCF.Primitives** i **CoreWCF.Http** oraz zaimplementowania kontraktu usługi

```
/// <summary>
/// Implementacja usługi WCF
/// </summary>
public class TestServiceImpl : ITestService
{
    public CustomResponseModel DoWork( CustomRequestModel request )
    {
        if ( request == null )
        {

```

```

        return new CustomResponseModel() { ResponseValue = "empty" };
    }
    else
    {
        return new CustomResponseModel()
        {
            ResponseValue = request.RequestValue + " from WCF"
        };
    }
}
}

```

Samo uruchomienie usługi wymaga wyłącznie użycia stosownego middleware

```

using CoreWCF;
using CoreWCF.Configuration;
using CoreWCF.Description;
using WCFContract;
using WCFDemoCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddServiceModelServices().AddServiceModelMetadata();
//builder.Services.AddSingleton<IServiceBehavior, UseRequestHeadersForMetad
ataAddressBehavior>();

var app = builder.Build();

app.UseServiceModel( builder =>
{
    builder.AddService<TestServiceImpl>( ( serviceOptions ) => { } )
        .AddServiceEndpoint<TestServiceImpl, ITestService>(
            new BasicHttpBinding(), "/TestService", endpoint =>
            {
                // tu możliwość rejestracji rozszerzeń
                //endpoint.EndpointBehaviors.Add( ... );
            } );
} );

var serviceMetadataBehavior = app.Services.GetRequiredService<ServiceMetada
taBehavior>();
serviceMetadataBehavior.HttpGetEnabled = true;

app.Run();

```

Jak widać istnieje możliwość rejestrowania rozszerzeń, kontrakty rozszerzeń są w większości przypadków zbieżne z tymi, które omówiono wyżej dla .NET.Framework.

### 8.3 Klient

Kod kliencki wymaga referencji do pakietów **System.ServiceModel.Primitives** i **System.ServiceModel.Http**. Dostępne są oba sposoby wytwarzania kodu klienckiego, zarówno przy użyciu **ChannelFactory** jak i przez dziedziczenie z **ClientBase**.

```
using System.ServiceModel;
using WCFContract;

var address = new EndpointAddress("http://localhost:5194/TestService");
var binding = new BasicHttpBinding();

var client = new ChannelFactory<ITestService>( binding, address ).CreateChannel();
var response = client.DoWork(new CustomRequestModel() { RequestValue = "foo" });

Console.WriteLine( response.ResponseValue );

Console.ReadLine();
```

## 9 Przykład z życia

Usługi systemu ePUAP związane m.in. z obsługą skrzynek podawczych czy podpisywania dokumentów przez ePUAP, są udostępnione dla integracji z zewnętrznymi systemami. Są to usługi typu SOAP.

Dla zainteresowanych – jak wygląda dokumentacja takich usług:

<https://epuap.gov.pl>

a następnie Strefa urzędnika / Pomoc / Dla integratorów / Specyfikacja WSDL