Projektowanie aplikacji ASP.NET

Wykład 12/15
ClickOnce, gRPC

Wiktor Zychla 2024/2025

Spis treści

2 ClickOnce 2
3 WebServices 5
3.1 Usługa 5
3.2 Proxy 7
3.3 Adres zwrotny dla usługi 11
4 gRPC 13
4.1 Serwer gRPC 13
4.2 Klient gRPC 15
4.3 gRPC-Web 16
4.3.1 Serwer 17
4.3.2 Klient 17

2 ClickOnce

Motywacja: aplikacja w przeglądarce ma dostęp wyłącznie do API ustandaryzowanych w ramach HTML5. W szczególności aplikacja przeglądarkowa **nie ma swobodnego dostępu do systemu plików oraz do zasobnika certyfikatów**. To uniemożliwia wykonanie w technologii przeglądarkowej takiej aplikacji, która wykona **podpisanie cyfrowe** dokumentu elektronicznego certyfikatem użytkownika – klucz prywatny certyfikatu jest dostępny wyłącznie na maszynie użytkownika i nie ma technicznie żadnego sposobu żeby dostać się do tego klucza prywatnego z poziomu przeglądarki, ani też nie ma sposobu żeby ten klucz wysłać na serwer (to drugie jest akurat oczekiwane, nie powinno być takiej możliwości).

Wzrost popularności technologii certyfikatów (m.in. w związku z możliwością instalowania certyfikatów na e-dowodach ale również z uwagi na możliwość pozyskania karty elektronicznej z certyfikatem) oznacza że istnieje potrzeba odniesienia się do tego problemu.

To oznacza, że w celu implementacji jednej z ważnych funkcjonalności oprogramowania przemysłowego – czyli podpisywania plików certyfikatami – należy użytkownikowi dostarczyć klasyczną aplikację desktopową.

Technologia ClickOnce pozwala na uruchamianie aplikacji desktopowych bezpośrednio z serwera aplikacji.

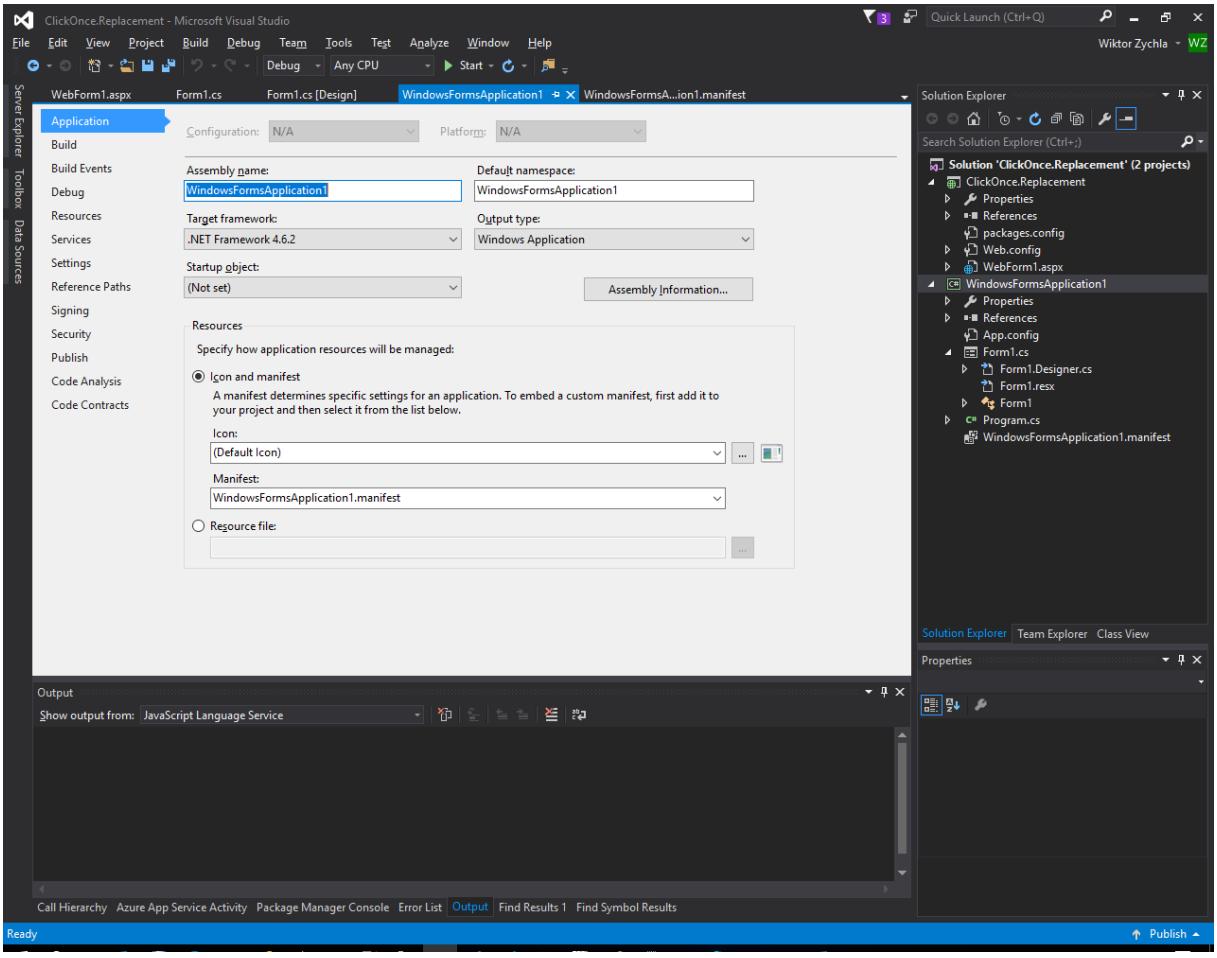
ClickOnce daje użyteczną alternatywę, z uwagi na dobrą charakterystykę:

	Klasyczna aplikacja desktopowa	Aplikacja ClickOnce
Wytwarzanie	.NET/C#	.NET/C#
Wymagania	Wymaga .NET Framework na maszynie użytkownika	Wymaga .NET Framework na maszynie użytkownika
Dostęp do API	Dostęp do całego API .NET	Dostęp do całego API .NET w tym API niedostępne dla aplikacji przeglądarkowych, np. • dostęp do systemu plików • dostęp do zasobnika certyfikatów
Instalacja	Pakiet instalacyjny, np. *.msi	Plik *.application i pliki *.deploy umieszczone na serwerze

		aplikacyjnym, dostępne z przeglądarki wspierającej ClickOnce (natywnie - Microsoft Edge, Microsoft Internet Explorer, pozostałe przeglądarki – pluginy)
Aktualizacja	Dodatkowy pakiet instalacyjny z aktualizacją	Automatyczna aktualizacja po umieszczeniu nowej wersji aplikacji na serwerze

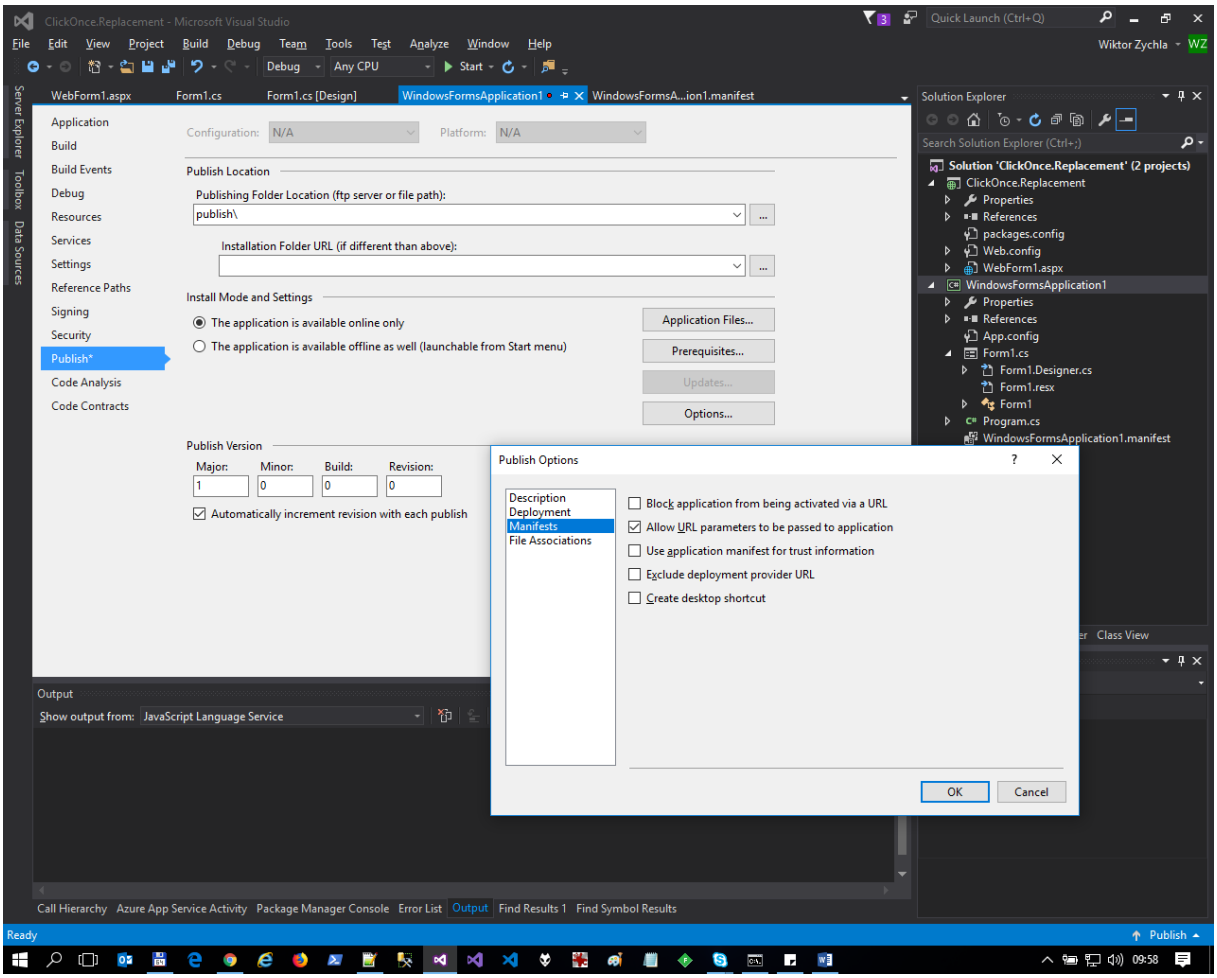
Wytworzenie aplikacji ClickOnce polega na przygotowaniu zwykłej aplikacji desktopowej i opublikowaniu jej na serwerze aplikacyjnym.

Krok 1. Na panelu właściwości projektu aplikacji desktop należy wyszukać zakładkę Publish

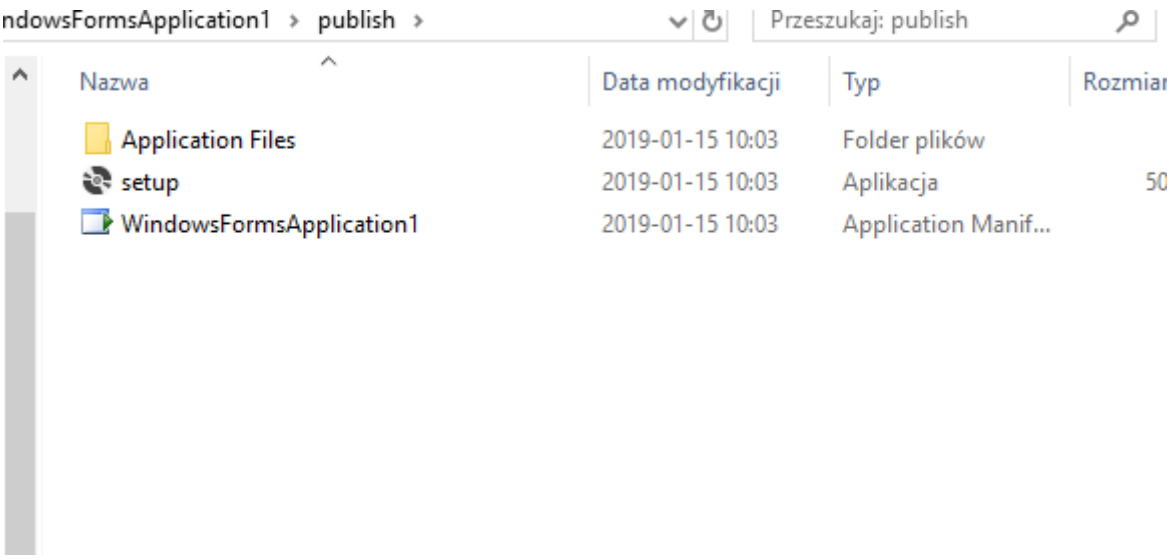


Krok 2. Na zakładce Publish należy

- zaznaczyć opcję Application is available online only
- ustawić wersję publikowanej aplikacji (uwaga, niezależna od wersjonowania samego pliku *.exe)
- w Options/Deployment upewnić się że wybrano „use *.deploy file extension”
- w Options/Manifests zaznaczyć „Allow URL parameters to be passed to application”



Krok 3. Ustawić folder docelowy i opublikować aplikację do wybranego foldera aplikacji web (Publish Now). Folderem docelowym może być podfolder foldera bieżącej aplikacji lub bezpośrednio – folder aplikacji web. W tym pierwszym przypadku zawartość podfoldera należy ręcznie przekopiować do foldera aplikacji web.



Krok 4. Aby uruchomić aplikację w przeglądarce, użytkownik aplikacji nawiguje do pliku *.application, w powyższym przykładzie byłoby to WindowsFormsApplication1.application. Odnośnik do pliku *.application może być umieszczony bezpośrednio na stronie web.

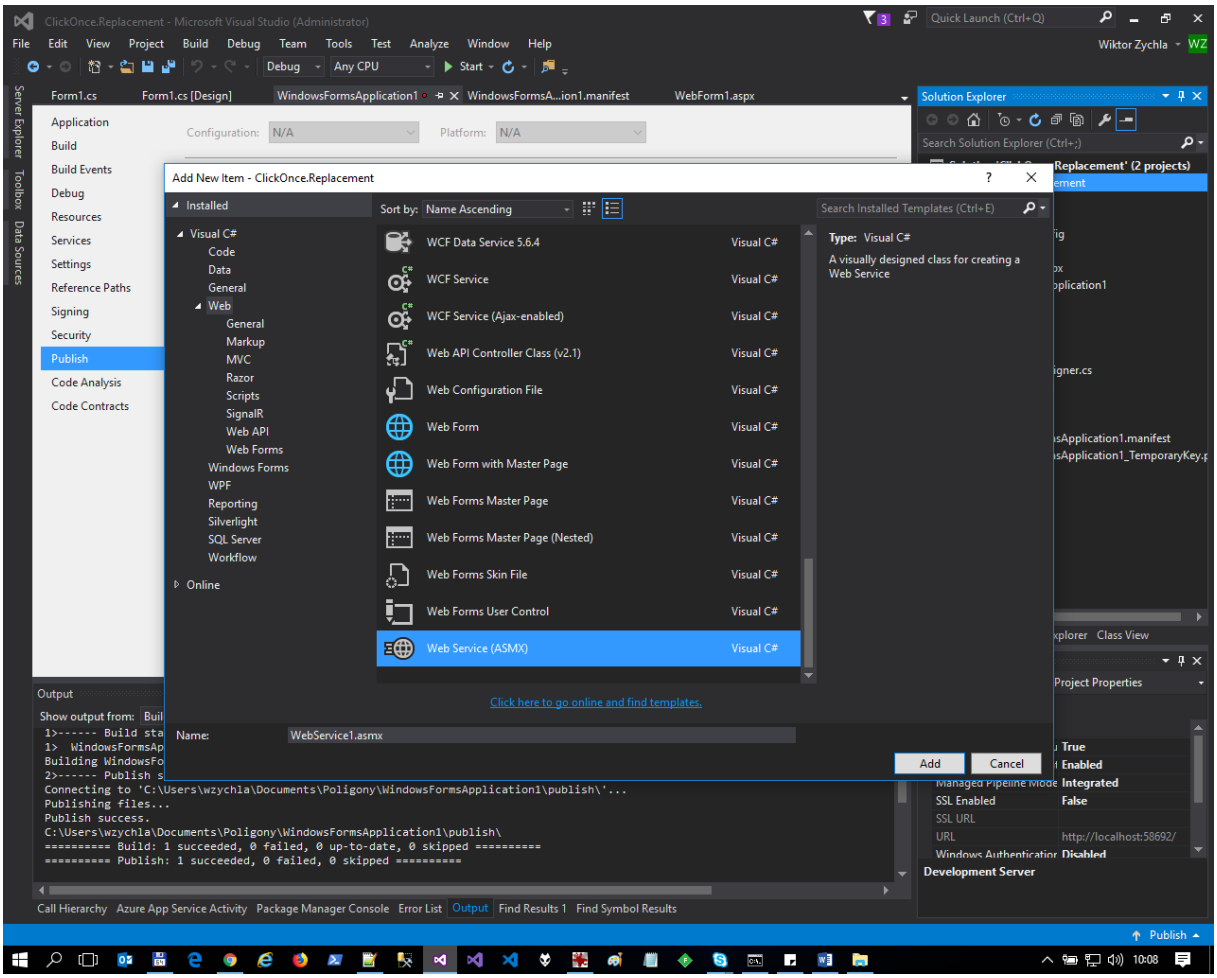
3 WebServices

3.1 Usługa

Aplikacja ClickOnce uruchamia się w środowisku użytkownika i ma nieograniczony dostęp do zasobów lokalnych ale w przeciwieństwie do aplikacji web – brakuje jej możliwości komunikacji z serwerem. Do tego celu należy dla aplikacji przygotować usługę aplikacyjną, najszybciej – usługę typu WebService.

> Uwaga! Po tym co już mówiliśmy o WCF, demonstrowanie ClickOnce przy pomocy usługi WebService „starego typu" ma wyłącznie uzasadnienie „bo tak jest najszybciej to pokazać na wykładzie". W praktycznych zastosowaniach, aplikacja ClickOnce komunikowałaby się z usługą WCF lub REST.

Krok 1. W projekcie aplikacji Web – dodaj nowy element, w gałęzi Web, komponent ASMX WebService



Krok 2. Uzupełnienie logiki aplikacji, na przykład

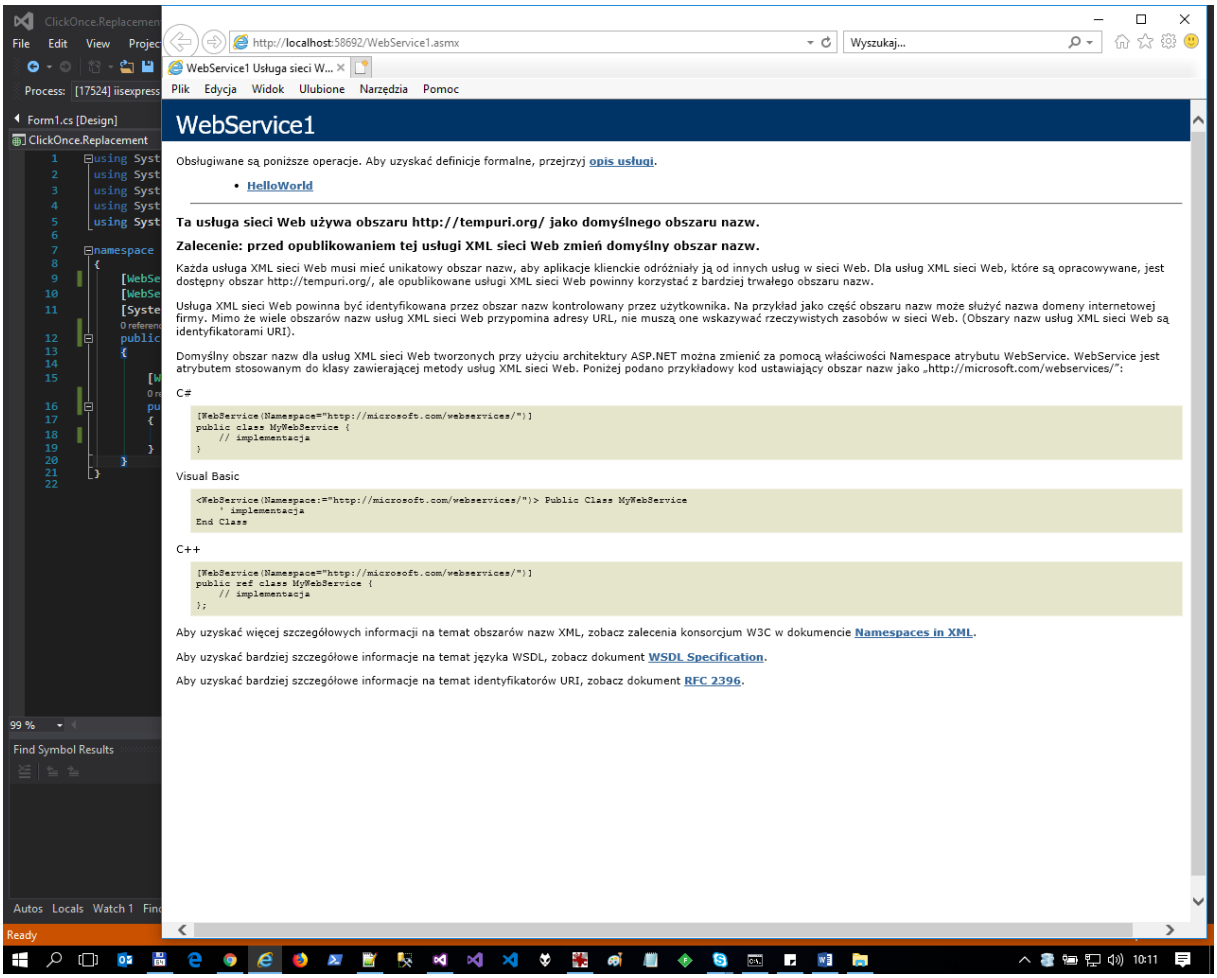
```
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
public class WebService1 : System.Web.Services.WebService
{

    [WebMethod]
    public string HelloWorld( string data )
```

```
    {
        return "Hello World " + data;
    }
}
```

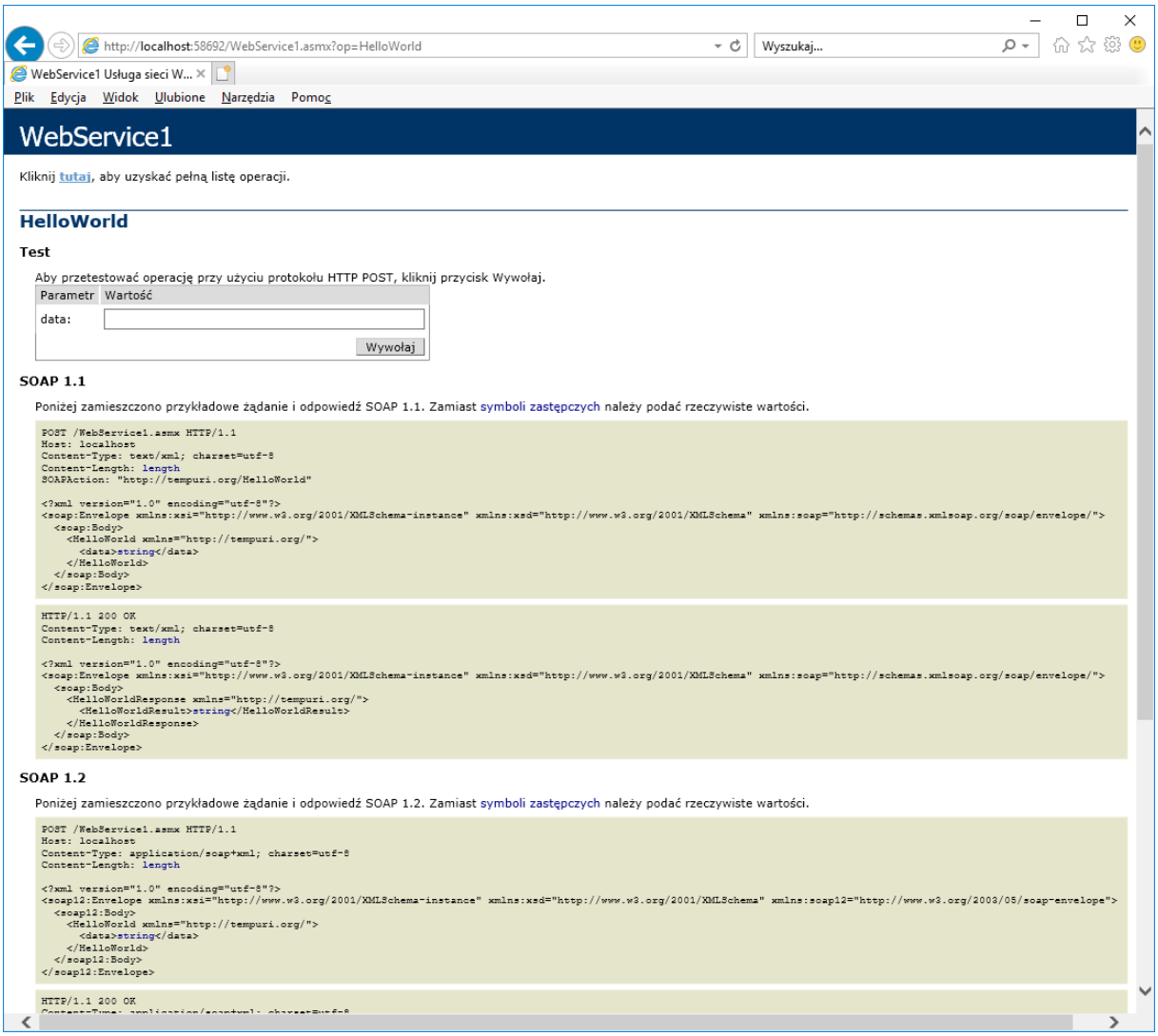
Ciało klasy dziedziczącej z **WebService** reprezentuje usługę, z którą będzie się komunikować aplikacja. Ciało metody **HelloWorld** to jedna z metod udostępnianych przez usługę.

Klient usługi komunikuje się z nią przez http/https, za pomocą dialektu SOAP, który w praktyce sprowadza się do żądania typu POST którego argumentem jest odpowiednio spreparowany XML.

Usługa zawołana w trybie GET pokazuje informację o tym jak ją wołać:



W szczególności, po wybraniu metody, można zobaczyć dokumentację standardu wywołania w dialektach SOAP 1.1, SOAP 1.2 i http POST. To oznacza, że usługa jest całkowicie interoperacyjna – klienta usługi można napisać w dowolnej technologii wspierającej gniazda TCP.

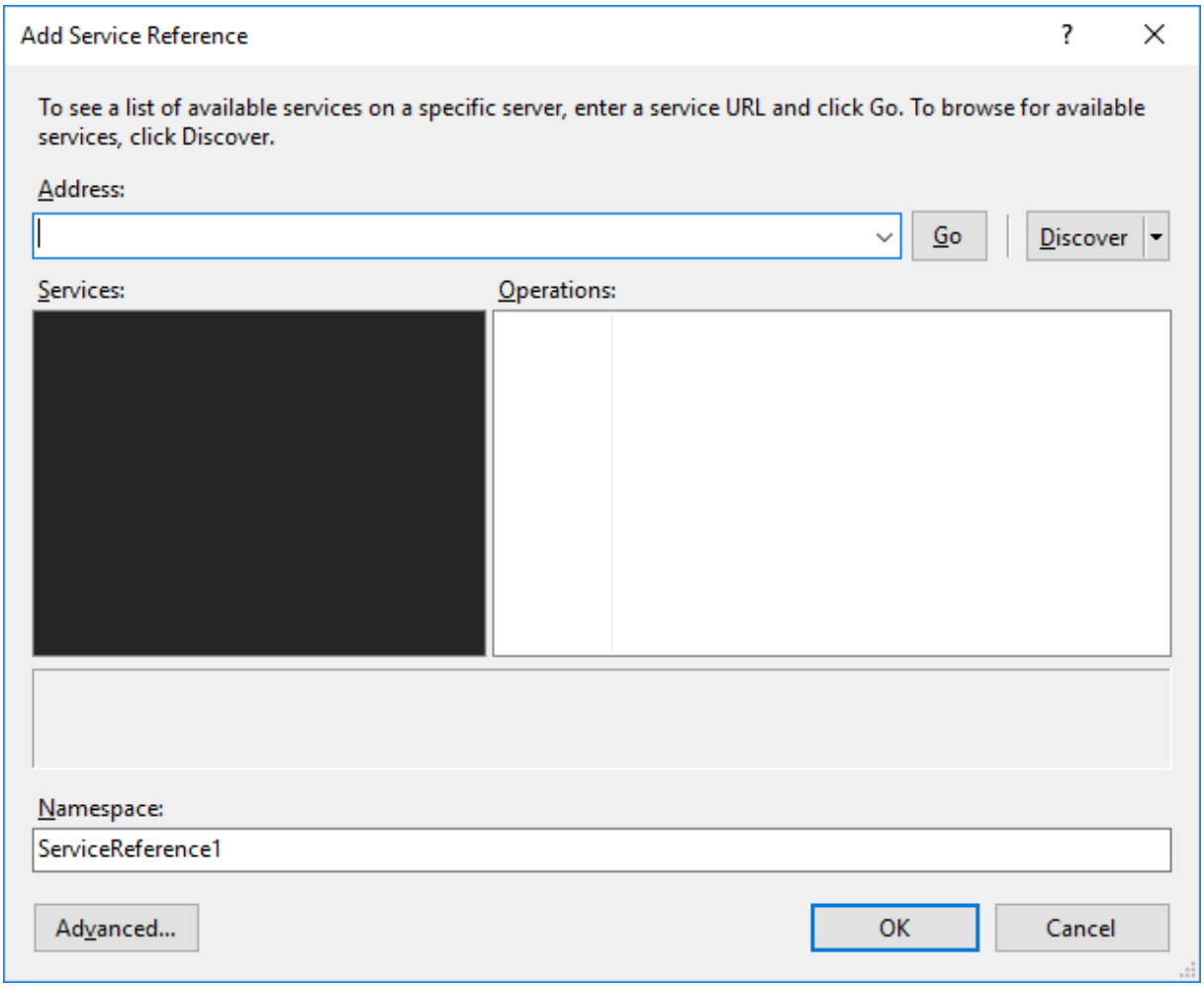


3.2 Proxy

Jeśli klientem usługi WebService jest aplikacja .NET (a tak jest w przypadku ClickOnce!), to można zautomatyzować przygotowanie kodu usługi klienckiej typu proxy. Jest to możliwe dzięki standardowi WSDL (o tym już mówiliśmy), który dokumentuje kontrakt usługi w sposób formalny.

W praktyce należy

Krok 1. W aplikacji desktop dodać referencję do usługi WebService – prawy przycisk na „references” i „Add Service Reference”. Opcjonalnie można użyć narzędzia **wsdl.exe** lub **svcutil.exe** z linii poleceń.

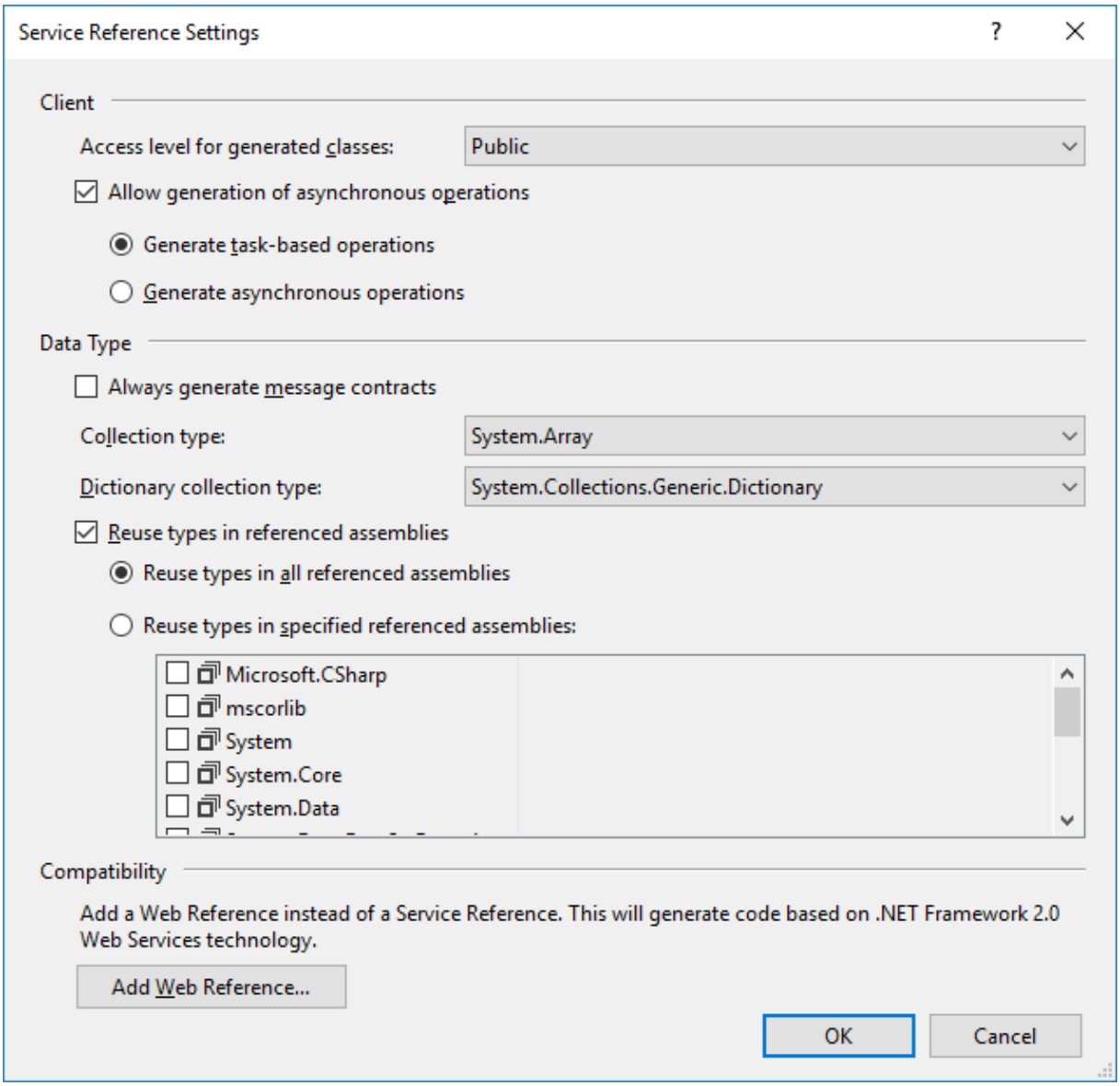


Z uwagi na wsparcie dla WCF, aktualne wersje Visual Studio pozwalają generować klasy proxy na dwa sposoby:

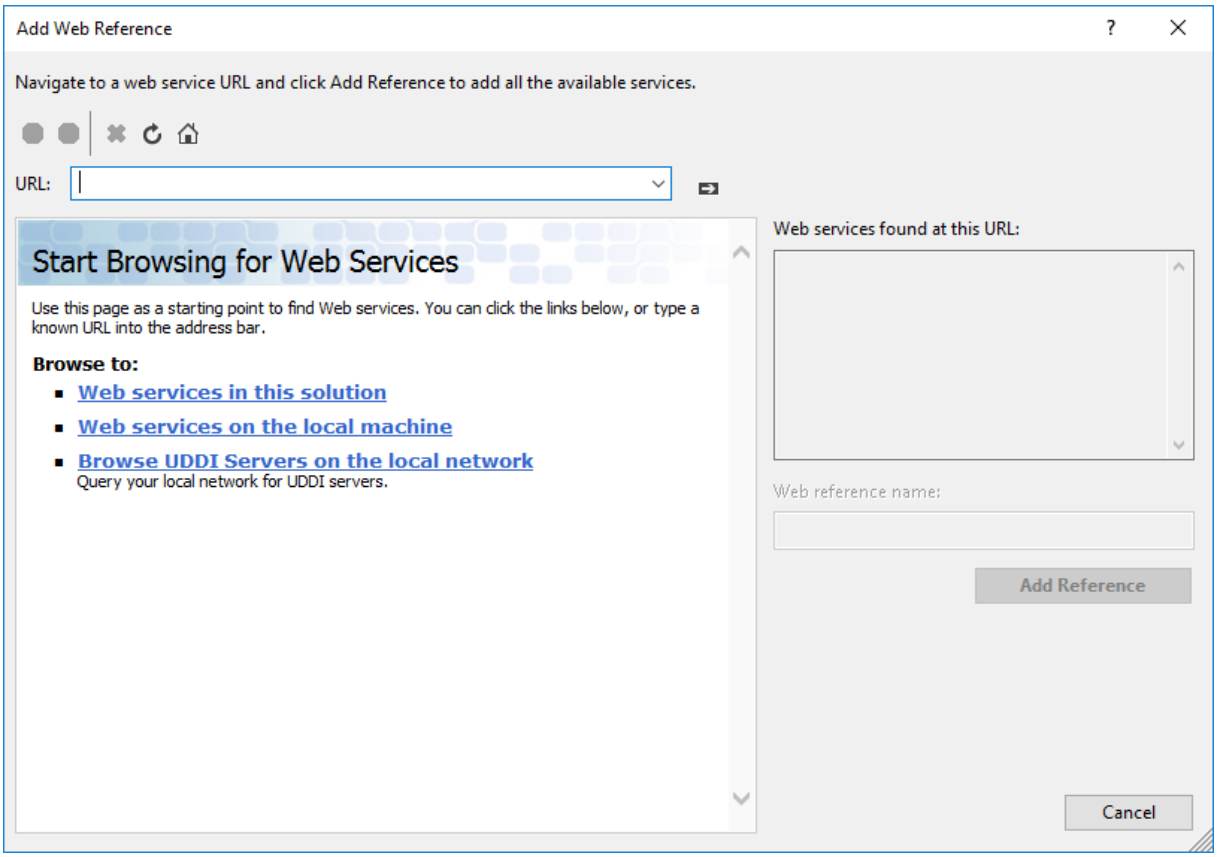
- Klasy proxy „starego” typu, przeznaczone pierwotnie do komunikacji z usługami WebServices – te klasy proxy dziedziczą z systemowej klasy **SoapHttpClientProtocol**
- Klasy proxy „nowego” typu przepznaczone do komunikacji z usługami WCF – te dziedziczą z **ClientBase**

W praktyce z uwagi na ten sam kontrakt (SOAP) klas proxy można używać zamiennie – klasa proxy do usługi WebService zadziała z usługą WCF, podobnie jak klasa proxy WCF.

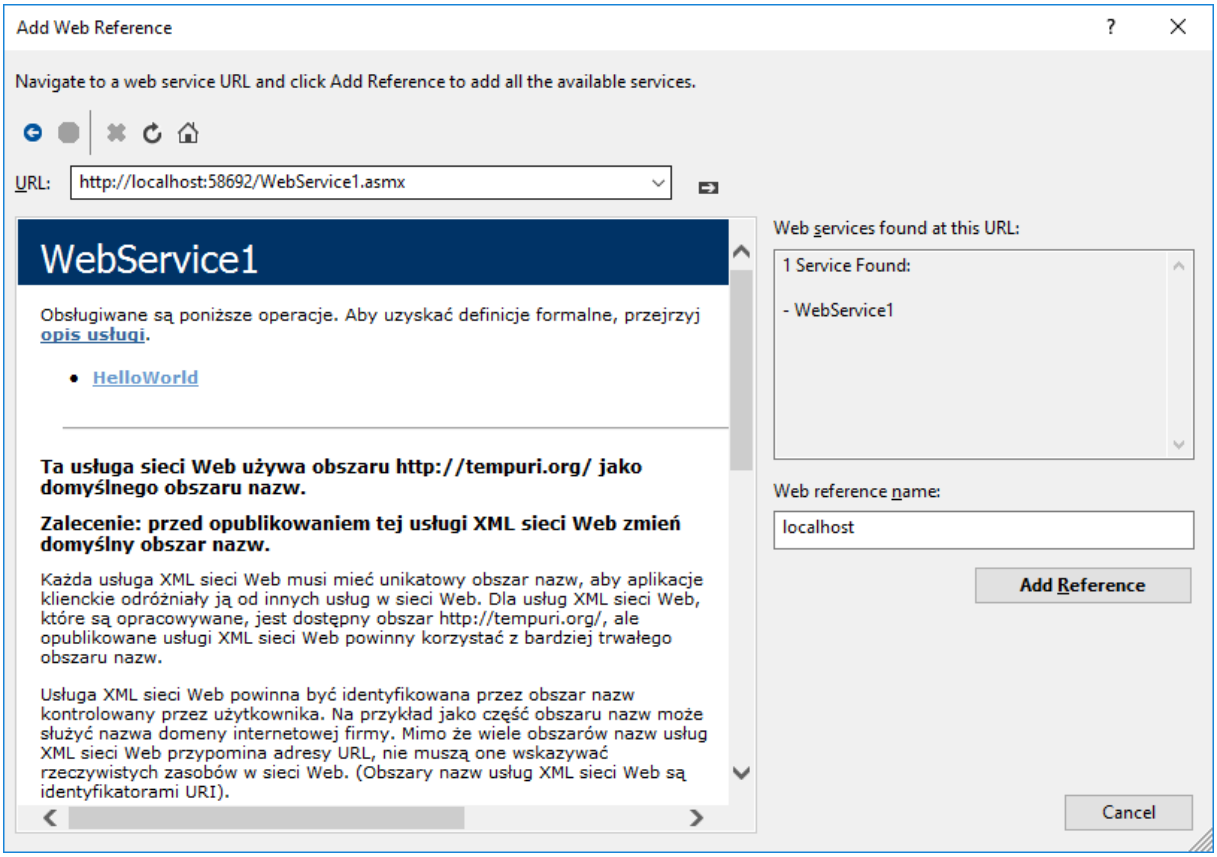
Ten kreator który widać na powyższym obrazie służy do generowania proxy „nowego” typu. Aby dostać się do generatora proxy starego typu, należy kliknąć „Advanced”



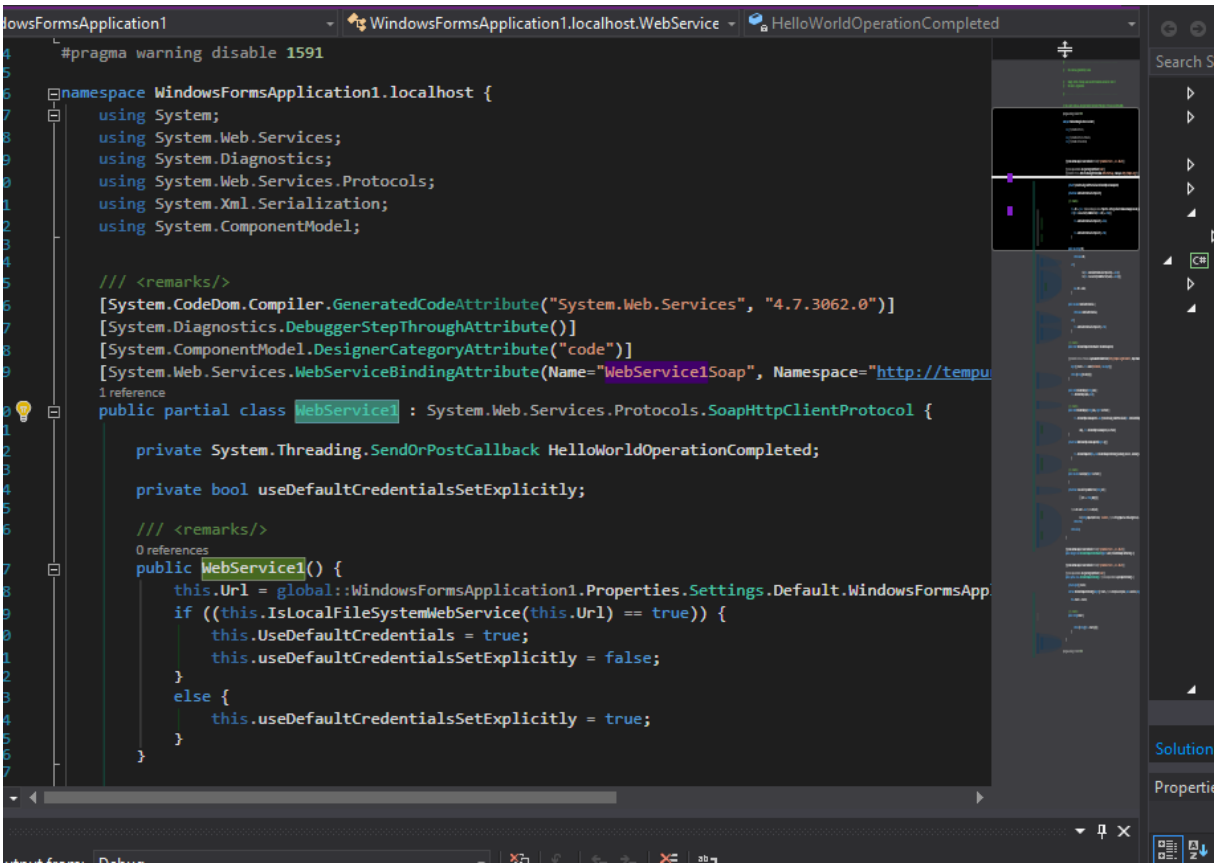
a tu „Add Web Reference”



Tu wystarczy wybrać „Web services in this solution”, wybrać usługę, wskazać dla niej alias dla proxy:

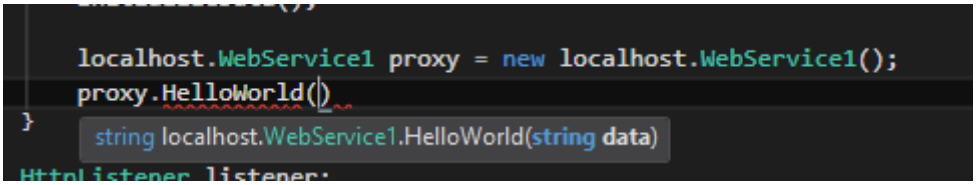


Wygenerowany kod proxy dla klienta można zobaczyć w pliku Reference.cs:



Wygenerowane proxy pozwala na wywołanie kodu usługi WebService z poziomu aplikacji desktop. Należy zwrócić uwagę na to że proxy rzeczywiście odwzorowuje sygnaturę metody mimo tego, że jej wywołanie oznacza w rzeczywistości:

- Utworzenie żądania SOAP
- Wysłanie żądania w trybie POST
- Odebranie odpowiedzi SOAP
- Sparsowanie odpowiedzi



Oba sposoby generowania, stary i nowy, są dostępne z linii poleceń:

- Stary – **wsdl.exe**
- Nowy – **svcutil.exe**

3.3 Adres zwrotny dla usługi

Wygenerowane proxy ma jedną niefortunność – adres usługi w klasie proxy jest zapisany jako adres miejsca w którym leżała usługa z której generowano proxy. Jest to zapewne localhost na jakimś porcie deweloperskim.

Po przeniesieniu aplikacji ClickOnce na serwer, usługa proxy dalej odwołuje się do localhost i to jest błąd – użytkownik na localhost **nie ma** serwera aplikacyjnego.

Powstaje więc pytanie – skąd aplikacja ClickOnce miałaby wiedzieć z jakiego adresu jest wywoływana, tak żeby **zmienić** adres docelowy dla proxy na taki, pod jakim rzeczywiście dostępna jest usługa?

Odpowiedź na to pytanie brzmi następująco: aplikacja ClickOnce może się dowiedzieć skąd została pobrana, przez **System.Deployment.Application.ApplicationDeployment.CurrentDeployment**. Typową praktyką jest odczytanie adresu pobrania aplikacji ClickOnce a następnie zbudowanie adresu usługi **relatywnie** w stosunku do adresu pobrania:

```
// aplikacja jest pobrana stąd
var deploymentUri = System.Deployment.Application.ApplicationDeployment.CurrentD
eployment.ActivationUri;

// WebService jest relatywnie w stosunku do adresu pobrania
var serviceUri    = new Uri(deploymentUri, "./WebService1.asmx");

localhost.WebService1 proxy = new localhost.WebService1();
proxy.Url                   = serviceUri.ToString();

var result                  = proxy.HelloWorld("foo");
```

4 gRPC

gRPC to framework dla wywołań zdalnych niezależny od platformy technologicznej. Powody dla których można chcieć użyć gRPC a nie „zwykłych" usług WCF/REST to m.in.

- gRPC jest domyślnie oparty o serializację Protocol Buffers która jest szybsza niż serializacja XML/JSON
- gRPC domyślnie bazuje na HTTP2 więc obsługuje streaming oraz dwukierunkową komunikację (na platformie .NET można opcjonalnie, dla zgodności, uruchomić usługi gRPC na „zwykłym" HTTP1.1)

Różne testy wydajności które można znaleźć w sieci (lub powtórzyć samodzielnie) mówią o kilkukrotnie (do 10x) większej wydajności.

Przed rozpoczęciem pracy z gRPC warto przestudiować oficjalną dokumentację ASP.NET.

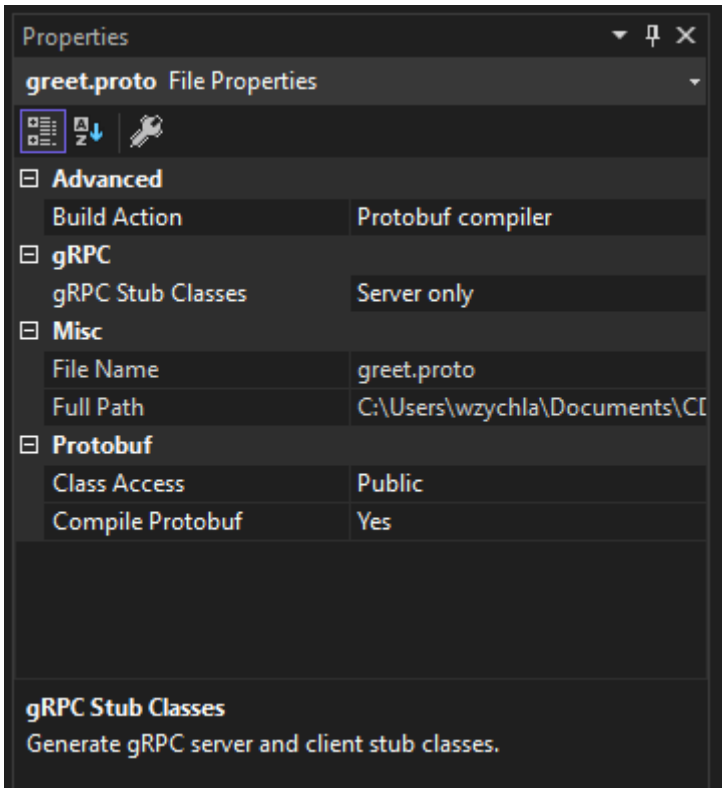
4.1 Serwer gRPC

Utworzenie serwera polega na użyciu szablonu projektu **ASPNET Core gRPC Service** który powoduje dodanie zależności do pakietu **Grpc.AspNetCore** (w chwili pisania tego materiału pakiet ma wersję 2.40).

Solucja składa się z opisu przykładowej usługi w języku proto3

```
syntax = "proto3";

option csharp_namespace = "GrpcService1";

package greet;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply);
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings.
message HelloReply {
  string message = 1;
}
```

Proszę zwrócić uwagę, że w VS ten typ artefaktu (*.proto) ma we właściwościach atrybut **Build Action** ustawiony na **Protobuf compiler**:



Kompilator protobuf to ważny element ekosystemu i w tym scenariuszu jest nie tylko wywoływany automatycznie przez plugin VS ale też w samej solucji nie widać nigdzie wprost wyniku jego pracy.

Sam kompilator protobuf, **protoc**, jest domyślnie zainstalowany w folderze pakietów nuget, na Windows jest to folder użytkownika i w nim

.nuget\packages\grpc.tools\2.40.0\tools\windows_x64\protoc.exe

Kompilator można przywoływać z linii poleceń co warto we własnym zakresie wypróbować.

W szablonie projektu w VS pojawia się już więc wyłącznie klasa implementująca samą usługę

```
public class GreeterService : Greeter.GreeterBase
{
    private readonly ILogger<GreeterService> _logger;
    public GreeterService( ILogger<GreeterService> logger )
    {
        _logger = logger;
    }

    public override Task<HelloReply> SayHello( HelloRequest request, Server
CallContext context )
    {
        return Task.FromResult( new HelloReply
        {
            Message = "Hello " + request.Name
        } );
    }
}
```

Kod aplikacji:

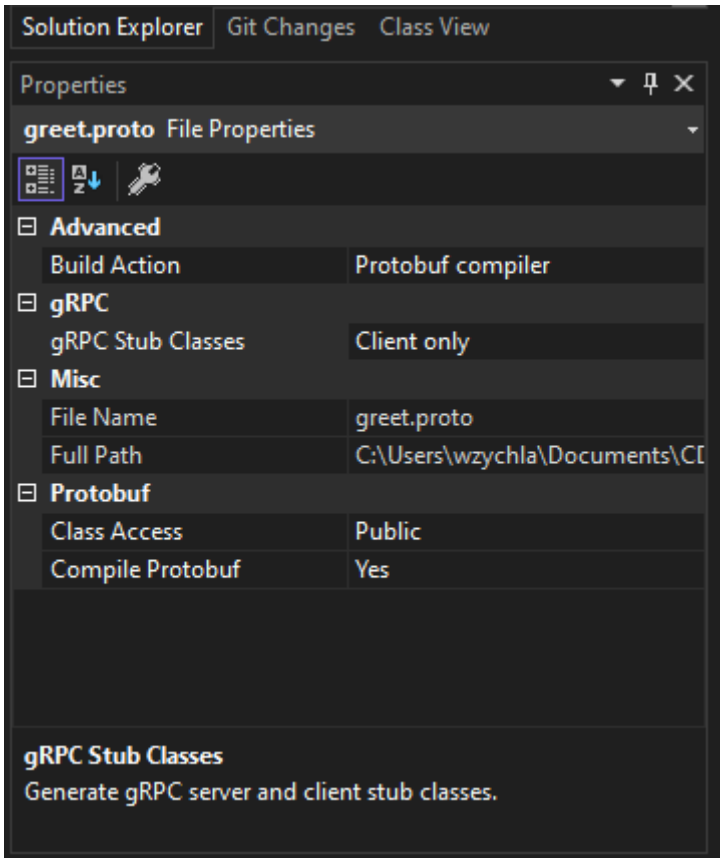
```
using GrpcService1.Services;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddGrpc();

var app = builder.Build();

// Configure the HTTP request pipeline.
app.MapGrpcService<GreeterService>();
app.MapGet( "/", () => "Communication with gRPC endpoints must be made thro
ugh a gRPC client." );

app.Run();
```

4.2 Klient gRPC

Tworzenie klienta gRPC polega na dodaniu referencji do pakietów

- Grpc.Net.Client
- Google.Protobuf
- Grpc.Tools

W projekcie aplikacji klienta należy utworzyć podfolder **Protos**, skopiować do niego opis usługi, w we właściwościach ponownie wskazać **Protobuf compiler** jako **Build action** ale tym razem należy wybrać opcję generowania kodu klienta.



Warto również zwrócić uwagę na możliwy problem niewłaściwego (niezaufanego) certyfikatu na środowisku deweloperskim.

Kod klienta

```
using Grpc.Net.Client;
using GrpcService1;

var handler = new HttpClientHandler();
handler.ServerCertificateCustomValidationCallback =
    HttpClientHandler.DangerousAcceptAnyServerCertificateValidator;

using var channel = GrpcChannel.ForAddress("https://localhost:7009",
    new GrpcChannelOptions() { HttpHandler = handler } );
var client        = new Greeter.GreeterClient(channel);
var reply         = await client.SayHelloAsync( new HelloRequest { Name = "
GreeterClient" });

Console.WriteLine( "Greeting: " + reply.Message );
Console.WriteLine( "Press any key to exit..." );

Console.ReadKey();
```

4.3 gRPC-Web

gRPC w wyżej opisanej wersji ma pewną niedogodność – serwer obsługujący protokół wymaga HTTP/2. O ile samo w sobie to nie jest uciążliwe (zarówno Kestrel uruchamiający aplikację w konsoli jak i IIS

obsługują HTTP/2) to problemem może być co innego – na przykład to że nie wszystkie debuggery HTTP obsługują HTTP/2.

Na przykład – Fiddler Classic.

Stąd próba uruchomienia kodu klient-serwer w scenariuszu z włączonym w tle Fiddlerem skończy się wyjątkiem ze strony klienta, mówiącym o tym że do połączenia wymagane jest HTTP/2 a klient widzi połączenie debuggera na HTTP/1.1.

Inna niedogodność domyślnej konfiguracji to brak możliwości wywołania usługi z klienta w przeglądarce (Javascript czy Blazor) – klienci przeglądarkowi nie obsługują HTTP/2.

Tym niedogodnościom naprzeciw wychodzi gRPC-Web. W przypadku ASP.NET Core wymagane są pewne drobne zmiany w już istniejącym kodzie.

4.3.1 Serwer

Serwer wymaga dodatkowego pakietu, **Grpc.AspNetCore.Web**. Po instalacji należy dodać middleware do potoku:

```
app.UseGrpcWeb( new GrpcWebOptions() { DefaultEnabled = true } );
```

Należy również upewnić się że serwer będzie obsługiwał przychodzące połączenia HTTP/1.1:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1AndHttp2"
    }
  }
}
```

(domyślne ustawienie **Protocols** to **Http2**)

4.3.2 Klient

Klient wymaga dodatkowego pakietu, **Grpc.Net.Client.Web.** Po instalacji należy zmodyfikować kod klienta

```
...

using var channel = GrpcChannel.ForAddress("https://localhost:7009",
```

```
    new GrpcChannelOptions() { HttpHandler = new GrpcWebHandler( handler )
} );

. . .
```

Proszę zwrócić uwagę że modyfikacji podlega wyłącznie konfiguracja handlera HTTP.