Ćwiczenie 6-A

Usługi typu REST

Autor: Mariusz Fraś

1 Cele ćwiczenia

Celem ćwiczenia jest:

1. Opanowanie techniki realizacji serwisu hostowanego przez serwer www IIS.
2. Opanowanie techniki realizacji serwisów webowych typu REST.

Zadanie polega na:

- Zrealizowanie usługi typu REST hostowanej przez serwer WWW (w tym przypadku użyte będzie uproszczone oprogramowanie, **IIS Express** jako host usługi.).
  - usługa będzie obsługiwać żądania HTTP (metody GET, POST, …) odwołujących się do zasobów z użyciem konkretnych URL-i.
  - w ćwiczeniu usługa operuje na danych w formatach XML i JSON.
- Zrealizowanie klienta konsolowego umożliwiającego wysłanie żądania odwołującego się do konkretnego zasobu.

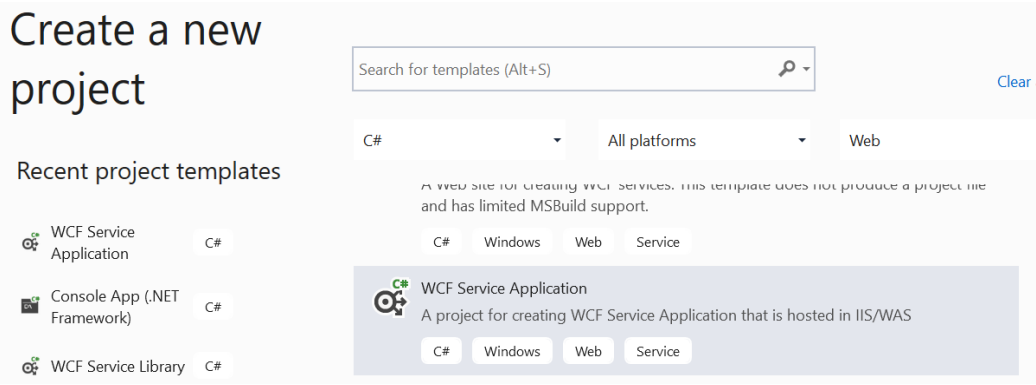
2 Zadanie – część I

Usługa typu REST hostowana przez serwer www.

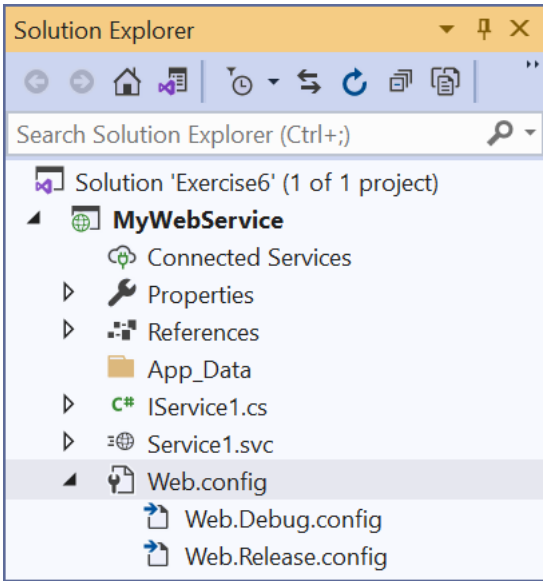
1. Definiowanie usługi WCF w stylu REST

Utwórz aplikację hostowaną w serwerze IIS.

- Utwórz nową solucję i projekt aplikacji typu **WCF Service Application** nadając jej własną nazwę – opcja: **Add… → New Project → *Odpowiednia opcja***.



- Przejrzyj zawartość projektu. Zwróć uwagę na następujący element:
  - **Web.config** – konfiguracja niezbędna do załadowania kontraktu poprzez host serwisu WCF (tu: serwis www).
  - **Service1.svc** – węzeł z kodem implementacji usługi.



- Przejrzyj plik Web.config. Zwróć uwagę na brak węzła **`<Service>`** - jest on opcjonalny.
- W pliku **IService1.cs** i zdefiniuj interfejs (kontrakt) usługi **IRestService** oraz kontrakt danych (w tym celu można sfaktoryzować istniejący kod).

- Zdefiniuj kontrakt danych (*DataContract*) – klasę opisującą jakiś zestaw danych, zawierający co najmniej 3 pola, w tym identyfikator (np. ID elementu, nazwa elementu, jakaś wartość elementu, itp.):

```
[DataContract]
  public class contract_type {
    [DataMember(Order = 0)]
    public int X1 { get; set; }
     [DataMember(Order = 1)]
    public string X2 { get; set; }
     [DataMember(Order = 2)]
    public double X3 { get; set; }
  }
```

  gdzie ***contract_type*** to twoja nazwa klasy kontraktu (np. Item Book, Car, itp.), ***X1***, ***X2,...*** to nazwy pól kontraktu danych (nadane w definicji kontraktu własne nazwy).
  Dla tworzonego później konstruktora, pola muszą być **public**.

- Zdefiniuj kontrakt serwisu **IRestService** z metodami:
  - metoda **getAllXml**
    - zwraca listę elementów (zdefiniowanych przez kontrakt)
    - ma atrybut typu **WebGet** – realizuje pobieranie danych przez klienta metodą **GET**,
    - nie ma specyfikacji formatu danych – używa domyślnego **xml**.
  - metoda **getByIdXml**
    - zwraca jeden element o identyfikatorze **Id**
    - ma atrybut typu **WebGet** oraz
    - specyfikuje format zwracanych danych **xml**,
  - metoda **addXml**
    - dodaje element o identyfikatorze **Id**
    - ma atrybut typu **WebInvoke** i specyfikowaną metodę **POST**,
    - specyfikuje przekazywanie danych w formacie **xml**,
    - zwraca string opisujący wynik operacji.
  - metoda **deleteXml**
    - usuwa element o podanym identyfikatorze metodą **DELETE**.

```
public interface IRestService
{
  [OperationContract]
  [WebGet(UriTemplate = "/Xxx")]
   List<contract_type> getAllXml();
  [OperationContract]
  [WebGet(UriTemplate = "/Xxx/{id}",
          ResponseFormat = WebMessageFormat.Xml)]
   typ_kontraktu getByIdXml(string Id);
  [OperationContract]
  [WebInvoke(UriTemplate = "/Xxx",
             Method = "POST",
             RequestFormat = WebMessageFormat.Xml)]
   string addXml(contract_type item);
```

```
    [OperationContract]
    [WebInvoke(UriTemplate = "/Xxx{id}", Method = "DELETE"]
     string deleteXml(string Id);
}
```

gdzie: ***contract_type*** – to typ (klasa) kontraktu danych,
Xxx – to własna nazwa folderu (zasobnika danych) dla zasobów – np. *items*, *books* dla książek, *cars* dla samochodów, itp. Ponadto:

- w metodach specyfikowany jest szablon **Uri** – relatywna (do adresu bazowego usługi) ścieżka URL do zasobu (jako string).
- w ścieżce możliwa jest specyfikacja zmiennej części stringu (różna w różnych wywołaniach), która jest podawana w nawiasach {…}.

- W pliku **Service1.svc.cs** wpisz kod klasy **MyRestService** implementującej interfejs **IRestService**:
  W klasie ma być:
  - Specyfikacja jednej instancji serwisu dla wszystkich wywołań:

    ```
    [ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
    ```

  - Zadeklarowana lista elementów – globalna zmienna dla klasy:

    ```
    private static List<contract_type> Yyy;
    ```

    gdzie: ***Yyy*** to nazwa listy, ***contract_type*** to typ kontraktu danych.
  - Można dodać konstruktor, a w nim dodać do listy trzy elementy:

    ```
    Yyy = new List<contract_type>() {
      new contract_type {x1=…,x2=…,x3=…},
      new contract_type {x1=…,x2=…,x3=…},
      new contract_type {x1=…,x2=…,x3=…},
    }
    ```

    gdzie ***x1***, ***x2*** i ***x3*** to nazwy pól kontraktu danych (nadane w definicji kontraktu własne nazwy). W miejsce kropek wpisz odpowiednie wartości. Np. {Id=123, Price=9,2,…}. (o **x1** patrz *Uwagi* poniżej)
  - Metoda `getAll()` , która zwraca wszystkie elementy listy ***Yyy***.

    ```
    public List<contract_type> getAllXml() {
      return Yyy;
    }
    ```

  - Metoda `getByIdXml(string Id)` , która zwraca jeden element typu *contract_type* o identyfikatorze **Id**.

    ```
    public contract_type getByIdXml(string Id)
    {
      int intId = int.Parse(Id);
      int idx = Yyy.FindIndex(b => b.Id == intId);
      if (idx == -1)
        throw new WebFaultException<string>("404: Not Found",
                                            HttpStatusCode.NotFound);
      return Yyy.ElementAt(idx);
    }
    ```

Uwagi:

- parametry trzeba przekształcać na odpowiedni typ,
- tu założono, że pierwsze pole kontraktu danych **x1** to unikalny identyfikator – zamiast **x1** użyj własnej nazwy (tu użyto **Id**).
- instrukcja **`Find(…)`** wyszukuje dla danej listy element spełniający podany w nawiasach warunek (w takiej postaci jak w przykładzie) – b oznacza odwołanie do elementu listy (można użyć dowolnego oznaczenia zamiast b) – tu szukamy elementu listy ***Yyy*** dla którego pierwsze pole ma wartość identyfikatora **Id**..
- tu: **Id** równe **-1** oznacza, że nie znaleziono elementu,

Ustawianie i wysyłanie odpowiedzi z odpowiednimi nagłówkami.

- skonfigurowanie nagłówków odpowiedzi (w tym kodu odpowiedzi) jest możliwe za pomocą obiektu reprezentującego odpowiedź, uzyskanego z kontekstu operacji:

```
OutgoingWebResponseContext response;   //global variable
response = WebOperationContext.Current.OutgoingResponse;
response.StatusCode = HttpStatusCode.some_code;
```

gdzie ***some_code*** to ustawiany jakiś kod HTTP.
Zmienna **response** powinna być globalna w klasie.

- w przypadku wystąpienia błędu, który nie pozwala przekazać danych można użyć metody **`WebFaultException<T>(…)`**, która powoduje zwrot odpowiedniego kodu błędu HTTP oraz krótki opis błędu (podawanych w parametrach).

- Metoda `addXml(contract_type item)`, która dodaje do listy element o identyfikatorze **Id** przekazywany w zmiennej **item**. (*Uwaga: tu zakładamy, że serwis przypisuje nowe ID*).
  W metodzie należy (podobnie jak dla `getByIdXml(…)`):
  - sprawdzić czy **item** nie jest null,
  - sprawdzić, czy nie istnieje już element o takim samym ID – jeśli nie to dodać dane, jeśli tak to zwrócić odpowiedni kod błędu,

```
public string addXml(contract_type item) {
  if (element == null)
    throw new WebFaultException<string>("400: BadRequest",
                                        HttpStatusCode.BadRequest);
[ dobierz nowy indeks (identyfikator) - newIdx ]
  int idx = Yyy.FindIndex(b => b.Id == newIdx);
  if (idx == -1) {
    item.Id = newIdx;
    Yyy.Add(item);
    return "Added item with ID=" + item.Id;
  } else
    throw new WebFaultException<string>("409: Conflict",
                                        HttpStatusCode.Conflict);
}
```

<table>
<tr>
<td></td>
<td>

Metoda <code>string deleteXml(string Id)</code> , która usuwa element o identyfikatorze Id.
Metoda jest niemal identyczna jak <code>getByIdXml(string Id)</code> tylko:
- metoda zwraca <code>string</code>,
- zamiast:
<code>return Yyy.ElementAt(idx);</code>
należy wywołać:
<code>Yyy.RemoveAt(idx);
return "Removed item with ID=" + Id;</code>

Otwórz plik Web.config i zmodyfikuj w celu obsługi żądań REST:

W sekcji <code><system.serviceModel></code> dopisz opis serwisu i endpointów:
<code><services>
<service name="MyWebSerwis.RestService">
<endpoint address=""
binding="webHttpBinding"
contract="MyWebSerwis.IRestService"
behaviorConfiguration="myRESTEndpointBehavior">
</endpoint>
</service>
</services></code>
gdzie: MyWebSerwis jest nazwą użytą jako przestrzeń nazw (namespace) aplikacji.

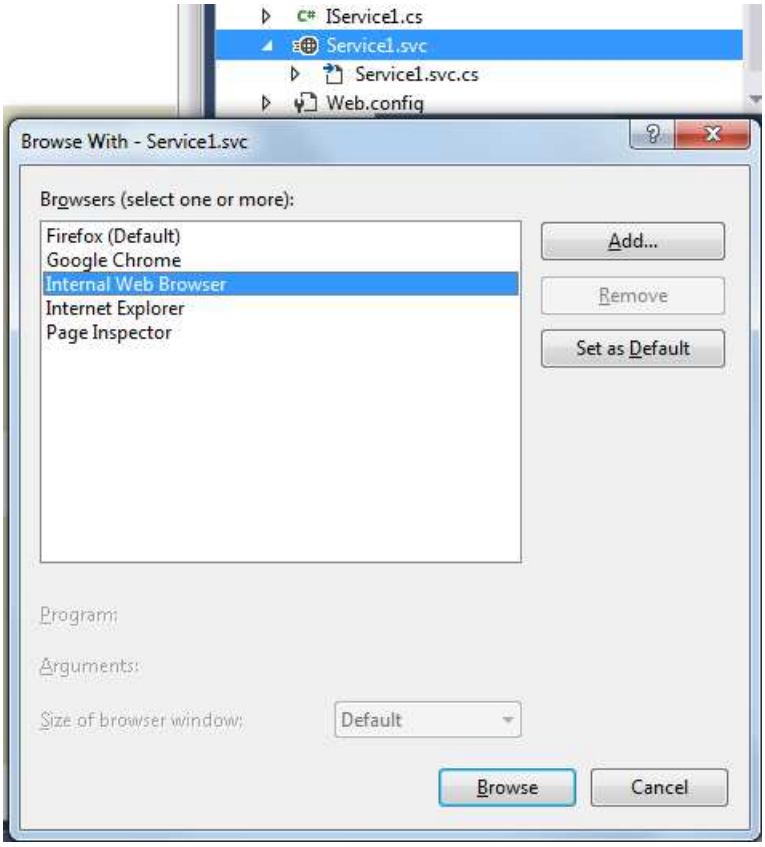
Wiązanie musi być typu webHttpBinding.
Serwis i endpoint jest opisany w zachowaniach.
Nazwa konfiguracji zachowania <code>behaviorConfiguration</code> jest opcjonalna – jeśli ma być użyta to musi być zdefiniowana.

W sekcji <code><behaviors></code> dopisz zachowanie endpointu webHttp pozwalające pobrać opis operacji w przeglądarce:
<code><endpointBehaviors>
<behavior name="myRESTEndpointBehavior">
<webHttp helpEnabled="true" />
</behavior>
</endpointBehaviors></code>
Upewnij się że jest ustawione generowanie metadanych o usłudze (w sekcji <behaviors> w <serviceBehaviors>):
<code>serviceMetadata httpGetEnabled="true"</code>

</td>
</tr>
<tr>
<td>2. Uruchomienie usługi</td>
<td>

Skompiluj projekt.
Kliknij w Solution Explorer prawym klawiszem węzeł z kodem usługi (np. Service1.scv.cs) i wybierz opcję Browse with….i wybierz jedną z możliwych opcji uruchomienia (przeglądarkę).

</td>
</tr>
</table>



- Poprzez ikonę IIS Express na pasku zadań (*TryIcon*) sprawdź uruchomienie usługi i informacje o niej

  - Zapamiętaj adres URL do usługi.

3. Testowanie usługi

Sprawdź działanie usługi za pomocą uruchomionej przeglądarki

- Wyświetl opis dostępnych operacji serwisu:
  w pasku adresu przeglądarki do adresu opisu usługi **Service1.svc** dopisz element ścieżki: /**help**.
  Zapoznaj się z podanymi informacjami.
- Wywołaj operację wyświetlenia wszystkich elementów:
  podaj adres operacji **getAllXml** (tu: dopisz w pasku adresu przeglądarki do adresu usługi **Service1.svc** element ścieżki: /**Xxx**).
  Zwróć uwagę na format wyświetlanych danych.
- Wywołaj operację wyświetlenia jednego elementu:
  podaj adres operacji **getByIdXml** (tu: dopisz w pasku adresu przeglądarki do adresu usługi **Service1.svc** element ścieżki: /**Xxx**/**num**, gdzie **num** – wartość identyfikatora danego elementu).
  Zwróć uwagę na format wyświetlanych danych.

4. Dopisanie operacji używających formatu json	***Uwaga**: teraz metody będą dla dwóch rodzajów reprezentacji danych (xml i nowe dla json) więc adresy operacji usługi dodatkowo będą musiały mieć rozróżnienie (człon specyfikujący te reprezentacje) Tu człon **json**. W przypadku stosowania jednego formatu rozróżnienie taki dodatkowy człon jest niepotrzebny!.*
	• Otwórz plik **IService1.cs** i dopisz w interfejsie usługi metody operacji na zasobach dla json:
	○ metoda **getAllJson**
	- identycznie jak dla **getAllXml** ale w kontrakcie wyspecyfikować adres (Uri) **"/json/Xxx"** zamiast **"/Xxx"** , oraz format danych Json:
	`... = WebMessageFormat.Json`
	○ metoda **getByIdJson**
	- identycznie jak dla **getByIdXml** ale w kontrakcie wyspecyfikować format Json i adres (Uri) **"/json/Xxx/{id}"** zamiast **"/Xxx/{id}"**.
	○ metoda **addJson**
	- identycznie jak dla **addXml** ale w kontrakcie wyspecyfikować format Json i adres (Uri) **"/json/Xxx"** zamiast **"/Xxx"**.
	○ metoda **deleteJson**
	- analogiczne zmiany jak powyżej.
	• Otwórz plik **Service1.svc.cs**. Dopisz kod metod dla formatu json:
	○ Metody są identyczne jak dla xml – mają tylko inne nazwy:
	`getAllJson(),`
	`getByIdJson(string Id),`
	`addJson(contract_type item),`
	`deleteJson(string Id).`
	Można też po prostu wywołać z nich odpowiednie metody dla xml.

5. Uruchomienie i testowanie usługi	• Skompiluj projekt. • Uruchom usługę za pomocą opcji **Browse with…**. • Sprawdź działanie metod **get**… za pomocą przeglądarki.
6. Testowanie działania aplikacji	**Testowanie działania aplikacji za pomocą narzędzia Postman** • Jeśli w systemie nie ma narzędzia Postman to ściągnij je i zainstaluj. Adres witryny: **https://www.getpostman.com** • Przygotuj odpowiednie dane xml i json dla operacji addXml i addJson. Można kierować się informacją zawartą na stronie help opisu operacji serwisu. • Przetestuj działanie aplikacji wysyłając żądania GET, POST, DELETE. ○ Żądania w aplikacji tworzy się opcją New/Request ○ W żądaniach wypełnij odpowiednio pole adresu.

<table>
<tr><td></td><td>

Dla żądań wysyłających dane (np. PUT i POST) trzeba także wyspecyfikować:

w nagłówkach (Headers), nagłówek: Content-type o odpowiedniej wartości dla json (application/json) i dla xml (text/xml),
dane należy umieścić w body żądania;
najlepiej wybrać zakładkę Body typu raw i adekwatnie ustawić dane typu JSON albo XML;
poprawną zawartość body dla XML wpisać wg wskazówek na stronie help operacji serwisu.

Pozostałe pola można zostawić nietknięte.

</td></tr>
<tr><td colspan="2">Poniżej opcjonalna wersja klienta (nieobowiązkowa):</td></tr>
<tr><td>7. Tworzenie konsolowej aplikacji klienckiej</td><td>
Utworzenie klienta konsolowego umożliwiającego wywoływanie zaimplementowanych usług.

Dodaj do solucji nowy projekt aplikacji klienta z szablonu Visual C# Console Application nadając jej własną nazwę.
Uwaga: w tym przypadku nie trzeba dodawać referencji serwisowej.
Otwórz plik Program.cs i dopisz następujący kod:

<pre>
static void Main(string[] args) {
  do   {
    try {
      Console.WriteLine("Podaj format (xml lub json):");
      string format = Console.ReadLine();
      Console.WriteLine("Podaj metode (GET lub POST lub ...):");
      string method = Console.ReadLine();
      Console.WriteLine("Podaj URI:");
      string uri = Console.ReadLine();
      HttpWebRequest req = WebRequest.Create(uri) as
                                                HttpWebRequest;

      req.KeepAlive = false;
      req.Method = method.ToUpper();
      if (format=="xml")
        req.ContentType = "text/xml";
      else if (format == "json")
        req.ContentType = "application/json";
      else {
        Console.WriteLine("Podales zle dane!");
        return;
      }
      switch(method.ToUpper()) {
       case "GET":
          break;
      // cont. on the next page
</pre>
</td></tr>
</table>

```
            case "POST":
              Console.WriteLine("Wklej zawartosc XML-a lub
                                 JSON-a (w jednej linii !)");
              string dane = Console.ReadLine();
              //przekodowanie tekstu wiadomosci:
              byte[] bufor = Encoding.UTF8.GetBytes(dane);
              req.ContentLength = bufor.Length;
              Stream postData = req.GetRequestStream();
              postData.Write(bufor, 0, bufor.Length);
              postData.Close();
              break;
           //tu ewentualnie kolejne opcje
            default:
              break;
          }
          HttpWebResponse resp = req.GetResponse()
                                          as HttpWebResponse;
          //przekodowanie tekstu odpowiedzi:
          Encoding enc = System.Text.Encoding.GetEncoding(1252);
          StreamReader responseStream =
              new StreamReader(resp.GetResponseStream(), enc);
          string responseString = responseStream.ReadToEnd();
          responseStream.Close();
          resp.Close();
          Console.WriteLine(responseString);
        }
        catch (Exception ex) {
          Console.WriteLine(ex.Message.ToString());
        }
        Console.WriteLine();
        Console.WriteLine("Do you want to continue?");
      } while (Console.ReadLine().ToUpper() == "Y");
}
```

Uwagi:
- obiekty typu `HttpWebRequest` i `HttpWebResponse` służą do wysyłania i odbierania żądań HTTP.
- operacje na strumieniach (i buforach) służą do ustawienia odpowiedniego kodowania (niestety Windows używa inny, własny standard).

8. Testowanie działania aplikacji

- Skompiluj aplikację kliencką
- Przygotuj odpowiednie dane xml i json dla operacji addXml i addJson. Można kierować informacją zawartą na stronie help opisu operacji serwisu.
- Uruchom klienta w oknie konsoli i przetestuj działanie klienta dla kilku dostępnych operacji.

3 Zadanie – część II (*ta część może być jeszcze uzupełniona*)

Szczegółowe wymagania i wybór zadania (A, B lub inne) określa prowadzący grupy.

A. Przećwiczyć prezentowane w ćwiczeniu techniki.

B. Przygotować się do napisania aplikacji o podobnych funkcjonalnościach według wskazówek prowadzącego
lub

C. zrealizować zadany przez prowadzącego program.

Przykładowy program do przygotowania na zaliczenie:

Opracować aplikację WCF (serwis i klienta) spełniające następujące wymagania:

Serwis:

1. Serwis jest usługą webową typu REST.
2. Serwis realizuje funkcjonalność typu magazyn danych, przy czym:
   - Obsługuje wszystkie operacje CRUD (Create, Read, Update, Delete), czyli:
     - odczyt pojedynczego rekordu oraz całej listy rekordów,
     - dodawanie rekordów danych.
     - usuwanie rekordów danych,
     - modyfikowanie wskazanego rekordu,
   - W przypadku próby wykonania niedozwolonej operacji (np. próba usunięcia rekordu o identyfikatorze którego nie ma) każda operacja powinna odpowiednio reagować.
   - Dane przechowywane są na serwerze (w dowolny sposób – np. w postaci listy).
3. Usługa używa formaty JSON i XML (każda operacja jest dla obu formatów).

Klient:

Klientem do testowania aplikacji może być narzędzie **Postman** (https://www.getpostman.com), które pozwala wykonywać różne żądania http.

W takim wypadku trzeba ściągnąć, zainstalować i opanować te narzędzie.

Klientem może być też własna (np. konsolowa) aplikacja, która pozwala korzystać z serwisu.

- Klient pozwala korzystać z wszystkich operacji serwisu.
- Wyniki operacji są wizualizowane – wyświetlane na ekranie.
- Wprowadzanie danych dla własnej aplikacji powinno być wygodne. Dobrym podejściem jest wprowadzane kolejnych danych przez użytkownika:
  – podając tylko wartości w konsoli dla kolejnych pól rekordów,
  – lub w polach aplikacji graficznej.