Wprowadzenie

Przygotowany tutorial zawiera przykładową wersję klienta usługi REST oraz skonfigurowany szkielet usługi WCF REST.

Usługa

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="WcfRest.RestService">
        <endpoint binding="webHttpBinding" contract="WcfRest.IRestService"
behaviorConfiguration="webEndpoint" />
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="">
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
          <serviceDebug includeExceptionDetailInFaults="false" />
          <serviceAuthorization
serviceAuthorizationManagerType="WcfRest.AuthorizationManager, WcfRest" />
        </behavior>
      </serviceBehaviors>
      <endpointBehaviors>
        <behavior name="webEndpoint">
          <webHttp defaultBodyStyle="Bare" automaticFormatSelectionEnabled="true"
helpEnabled="true" />
        </behavior>
      </endpointBehaviors>
    </behaviors>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="false"
multipleSiteBindingsEnabled="false" />
  </system.serviceModel>
  <system.web>
    <compilation debug="true" />
  </system.web>
</configuration>
```

Rys. 1 Plik konfiguracyjny usługi

Patrząc od góry, zarejestrowana została usługa *RestService*, przedstawiająca się kontraktem stworzonym przez interfejs *IRestService*. *Endpoint* dla usługi korzysta z wiązania *webHttpBinding.* Dla *endpointa* zdefiniowano poniżej zachowanie zawierające obiekt *webHttp* z ustawionym domyślnym formatowaniem *body* jako *Bare* (bez dodatkowego opakowywania treści) oraz automatyczną negocjacją typu (*automaticFormatSelectionEnabled)*, np. na podstawie nagłówka *Accept. Dodatkowo endpoint generuje także domyślną stronę pomocy – helpEnabled* (http://localhost:18964/RestService.svc/help).

Zdefiniowane zachowanie dla usługi zezwala na jej *debuggowanie,* oraz autoryzację za pomocą własnej klasy *AuthorizationManager.*

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Single, InstanceContextMode
   = InstanceContextMode.Single)]
public class RestService : IRestService
{
    private static OutgoingWebResponseContext Response =>
         WebOperationContext.Current?.OutgoingResponse;
}
```

Rys. 2 Szkielet klasy implementującej usługę

Dla usługi zostały zdefiniowane atrybuty zezwalające na utworzenie tylko jednego obiektu klasy RestService do obsługi zapytań oraz wyłączające współbieżność (dla uproszczenia).

Klasa zawiera także właściwość pozwalającą uzyskać bieżący kontekst odpowiedzi służący do ustawiania nagłówków, innych niż domyślne kodów odpowiedzi itp..

Dodatkowe pliki to:

- *IRestService* – zawiera kontrakt usługi,
- *Account* – przykładowy złożony typ danych, który będzie przesyłany w dalszej części,
- *CustomError* – przykładowa klasa błędu, reprezentująca złożony typ danych,
- *AuthorizationManager* – klasa pozwalająca na implementację autoryzacji zapytań.

Klient

Uwaga: *Dowiązanie w projekcie klienta usługi RestService ma na celu tylko uproszczenie procesu przesyłania danych (nie trzeba implementować od nowa złożonych typów danych do procesu serialiacji).*

```
try
{
    var request = CreateRequest("");
    request.Accept = JsonContentType;
    using (var response = GetResponse(request))
    {
        var body = GetBody(response);
        Console.WriteLine($"Status code: {response.StatusCode} {
               (int) response.StatusCode}");
        Console.WriteLine(body);
    }
}
catch (WebException e)
{
    var response = (HttpWebResponse) e.Response;
    Console.WriteLine("Webexception");
    Console.WriteLine($"Status code: {response.StatusCode} {
         (int) response.StatusCode}");
    Console.WriteLine(GetBody(response));
}
```

Rys. 3 Przykładowy proces przesyłania zapytania i odbierania odpowiedzi (także negatywnej)

Dodatkowe metody pomocnicze to:

- CreateRequest – tworzy obiekt zapytania na podstawie ścieżki relatywnej,
- GetResponse – odbiera odpowiedź na zapytanie i rzutuje na odpowiedni typ,
- GetBody – odczytuje body odpowiedzi i zamyka strumień,
- WriteBody – zapisuje body zapytania i zamyka strumień oraz ustawia nagłówki *Content-Length* i *Content-Type* oraz *Method,*
- *Serialize* i *Deserialize* – metody obsługujące proces serializacji i deserializacji obiektów do postaci JSON.

Zadanie 1

Pierwsze zadanie polega na obsłudze zapytania typu *GET* i zwróceniu odpowiedzi w postaci tekstu.

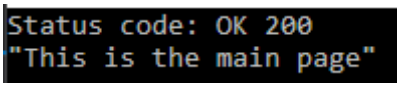
Propozycja rozwiązania:

```
public interface IRestService
{
    [OperationContract]
    [WebGet(UriTemplate = "")]
    string GetMainPage();
}
```

```
public class RestService : IRestService
{
    public string GetMainPage()
    {
        return "This is the main page";
    }

 }
```

Widok po stronie klienta:



Zadanie 2

Uwaga: To i poniższe zadania nie zawierają przykładowego kodu klienta. Nie oznacza to, że nie należy tego zrobić. Należy samodzielnie zaimplementować go za pomocą dostarczonego szkieletu. Sprowadza się to przede wszystkim do ustawienia body, ewentualnej serializacji, deserializacji oraz ścieżki relatywnej.

Polega na przesłaniu (metoda *PUT*) i odczytaniu tekstu.

Propozycja rozwiązania:

```
public interface IRestService
{
    [OperationContract]
    [WebGet(UriTemplate = "username")]
    string GetUserName();

    [OperationContract]
    [WebInvoke(UriTemplate = "username", Method = "PUT")]
    void PutUserName(string username);
}
```

```
public class RestService : IRestService
{
    public string GetUserName()
    {
        return _username;
    }

    public void PutUserName(string username)
    {
        _username = username;
    }

}
```

Zadanie 3

Przesyłanie złożonych typów danych.

Propozycja rozwiązania:

```
public interface IRestService
{
    [OperationContract]
    [WebGet(UriTemplate = "accounts")]
    List<Account> GetAccounts();

    [OperationContract]
    [WebInvoke(UriTemplate = "accounts", Method = "POST")]
    void PostAccount(Account account);
}
```

```
public class RestService : IRestService
{
    public List<Account> GetAccounts()
    {
        return _accounts.ToList();
    }

    public void PostAccount(Account account)
    {
        account.Guid = Guid.NewGuid();
        account.CreationDate = DateTime.Now;
        CreateAccount(account);
    }

    private void CreateAccount(Account account)
    {
        _accounts.Add(account);
        Response.Location = $"accounts/{account.Guid}";
        Response.StatusCode = HttpStatusCode.Created;
    }
}
```

```
[DataContract]
public class Account
{
    [DataMember]
    public Guid Guid { get; set; }

    [DataMember]
    public decimal Balance { get; set; }

    [DataMember]
    public DateTime CreationDate { get; set; }

    public Account()
    {
        CreationDate = DateTime.Now;
    }
}
```

Zadanie 4

Podstawowa obsługa błędów.

Przykładowe rozwiązanie:

```
public interface IRestService
{
    [OperationContract]
    [WebGet(UriTemplate = "accounts/{stringGuid}")]
    Account GetAccount(string stringGuid);
}
```

```
public class RestService : IRestService
{
    public Account GetAccount(string stringGuid)
    {
        var guid = Guid.Parse(stringGuid);
        var account = _accounts.SingleOrDefault(acc => acc.Guid == guid);
        if (account != null)
            return account;

        throw new WebFaultException(HttpStatusCode.NotFound);
    }
}
```

Zadanie 5

Przesyłanie złożonych błędów.

Propozycja rozwiązania:

```
public interface IRestService
{
    [OperationContract]
    [WebInvoke(UriTemplate = "accounts/{stringGuid}", Method = "PUT")]
    void PutAccount(string stringGuid, Account account);
}
```

```
public class RestService : IRestService
{
    public void PutAccount(string stringGuid, Account account)
    {
        var guid = Guid.Parse(stringGuid);
        account.Guid = guid;

        if (account.Balance < 0)
            throw new WebFaultException<CustomError>(
                new CustomError {Account = account,
                        Message = "Invalid balance"},
                        HttpStatusCode.BadRequest);

        var existingAccount = _accounts.SingleOrDefault(acc =>
                acc.Guid == guid);
        if (existingAccount == null)
        {
            CreateAccount(account);
            return;
        }

        _accounts.Remove(existingAccount);
        _accounts.Add(account);
    }
}
```

```
[DataContract]
public class CustomError
{
    [DataMember]
    public Account Account { get; set; }

    [DataMember]
    public string Message { get; set; }
}
```

Zadanie 6 (dla ambitnych)

Zaimplementowanie *Basic Authentication* w klasie *AuthorizationManager. Z*aimplementowane zostały już właściwości pozwalające uzyskać kontekst zapytania i odpowiedzi oraz metoda konwertująca tekst z Base64 do zwykłego tekstu.

Wskazówka: kod 401 zwrócić można za pomocą *WebFaultException* lub zwracając false (po ustawieniu kodu odpowiedzi).