Projektowanie aplikacji ASP.NET
Wykład 13/15
WebAPI / REST

Wiktor Zychla 2018/2019

Spis treści

2 Wprowadzenie

Podsystem WebAPI służy do wprowadzenia do ASP.NET implementacji usług typu REST w których językiem komunikacji klienta i serwera jest JSON.

O usługach typu REST należy myśleć jak o alternatywie technologicznej dla usług typu SOAP. Poniższa tabela podsumowuje kluczowe różnice:

	SOAP	REST
protokół	Jeden z wielu wspieranych przez WCF: • http • TCP • MSMQ • Dual	Tylko HTTP
Typ żądania	Zawsze POST	GET/POST/PUT/DELETE
Język opisu parametrów	SOAP	JSON
Język wartości zwracanej	SOAP	JSON
Metadane	WSDL	Brak oficjalnej specyfikacji, wiele konkurujących „standardów”, w tym interesujące np. OpenAPI
Generowanie kodu klienta na podstawie metadanych	Wiele możliwości	Różne dla każdego „standardu”, w wielu wypadkach – brak
Obsługa rozszerzeń (szyfrowanie, transakcje, itp.)	Standaryzacja WS-*	Brak

3 Tworzenie usługi typu REST

Utworzenie usługi WebAPI typu REST bardzo przypomina tworzenie kontrolerów MVC. Pierwszym krokiem jest router delegujący ścieżki do handlera WebAPI. Konfiguruje się go zwyczajowo w osobnej klasie:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // Web API configuration and services

        // Web API routes
        config.MapHttpAttributeRoutes();

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

którą woła się na starcie aplikacji:

```
void Application_Start(object sender, EventArgs e)
{
     // webapi
     GlobalConfiguration.Configure(WebApiConfig.Register);
     // mvc
     RouteConfig.RegisterRoutes(RouteTable.Routes);
}
```

Definicja routingu dla WebAPI **przed** MVC ma następujące uzasadnienie – routing dopasowuje ścieżki do wzorców w kolejności dopasowania. Ponieważ ścieżki WebAPI nie wymagają specyfikacji nazwy akcji, w ich szablonach występuje wyłącznie **nazwa kontrolera** (i opcjonalny argument). W ścieżkach MVC występowały natomiast zarówno nazwy kontrolerów jak i nazwy akcji.

Dla wywołania

/foo/bar

routing musi więc wiedzieć czy chodzi o kontroler WebAPI czy o kontroler MVC. Wymyślono więc prostą, wygodną konwencję – ścieżki WebAPI mają stały prefix, **/api**, który jest elementem ścieżki. Następujące odwołania

/api/User

/api/Entity

są więc na pewno odwołaniami do kontrolerów User i Entity WebAPI, a nie do akcji User/Entity kontrolera api MVC – ponieważ router WebAPI ma pierwszeństwo.

Dzięki takiej konwencji oraz dodatkowej konwencji samego ASP.NET – w której istnienie pliku statycznego ma zawsze pierwszeństwo routera, możliwe jest **współistnienie** wszystkich rodzajów artefaktów w jednej i tej samej aplikacji ASP.NET:

- Stron WebForms (*.aspx) – bo są plikami statycznymi i odwołania do nich będą miały priorytet
- Usług ASMX WebService (*.asmx) – bo są plikami statycznymi
- Usług WCF (*.svc) – bo są plikami statycznymi
- Usług WCF o dynamicznej aktywacji – jeśli reguły routingu są zdefiniowane odpowiednio wysoko
- Usług WebAPI – bo mają stały prefix ścieżki (/api)
- Kontrolerów MVC – do wszystkich pozostałych żądań

3.1 Model

Implementacja usługi ma zwykle model danych

```
public class Person
{
    public int ID { get; set; }
    public string Name { get; set; }
}
```

3.2 Kontroler

Kontroler udostępnia metody, wedle konwencji nazywane tak jak odpowiednie polecenia http (GET, POST, PUT, DELETE). Konwencja mapowania poleceń na logikę aplikacji jest według REST następujaca

Pobieranie danych	GET
Dodawanie danych	POST
Aktualizacja istniejących danych	PUT
Usuwanie danych	DELETE

Istnieje możliwość **przeciążania** metod o tym samym poleceniu dostępu, WebAPI spróbuje dopasować żądanie na podstawie przekazanych argumentów. Jeden argument może być **typu złożonego** (i umieszcza się go w body żądania), pozostałe mogą być częścią ścieżki:

```
public class IndexController : ApiController
 {
     public IHttpActionResult Get()
     {
         var persons = new[]
         {
             new Person() { ID = 1, Name = "person1" },
             new Person() { ID = 2, Name = "person2" }
         };
         return this.Ok(persons);
     }

     public IHttpActionResult Get(bool filter)
     {
         var persons = new[]
```

```
        {
            new Person() { ID = 1, Name = "person1f1" },
            new Person() { ID = 2, Name = "person2f1" }
        };
        return this.Ok(persons);
    }

    public IHttpActionResult Get(bool filter, string filter2)
    {
        var persons = new[]
        {
            new Person() { ID = 1, Name = "person1f2" },
            new Person() { ID = 2, Name = "person2f2" }
        };
        return this.Ok(persons);
    }

    public IHttpActionResult Post(Person person)
    {
        if (person == null) return this.BadRequest("no data");

        return this.Ok(new Person()
        {
            ID = person.ID,
            Name = person.Name + " accepted"
        });
    }

    public IHttpActionResult Post( Person person, bool data )
    {
        if (person == null) return this.BadRequest("no data");

        return this.Ok(new Person()
        {
            ID = person.ID,
            Name = person.Name + " accepted with " + data.ToString()
        });
    }
}
```

4 Tworzenie kodu klienckiego REST

4.1 Klient w Javascript

Jedną z zalet usług WebAPI jest łatwość budowania kodu klienckiego w Javascript. Wynika to z użycia JSON jako języka komunikacji:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <script src="https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjax.js"></scr
ipt>
    <script src="/service1.svc/js"></script>
</head>
<body>
    <div>
        <button id="invoke">Invoke</button>
        <input id="result" />
    </div>
    <script>
        window.addEventListener('load', function () {
            window.invoke.onclick = function () {
                var xhr = new XMLHttpRequest();

                xhr.open('POST', '/api/Index');
                xhr.setRequestHeader("Content-Type", "application/json");
                xhr.onreadystatechange = function () {
                    if (xhr.readyState == XMLHttpRequest.DONE) {
                        window.alert(xhr.responseText);
                    }
                };
                var person = {
                    ID: 12,
                    Name: 'foo bar'
                };
                xhr.send(JSON.stringify(person));
            };
        });
    </script>
</body>
</html>
```

4.2 Klient w C#

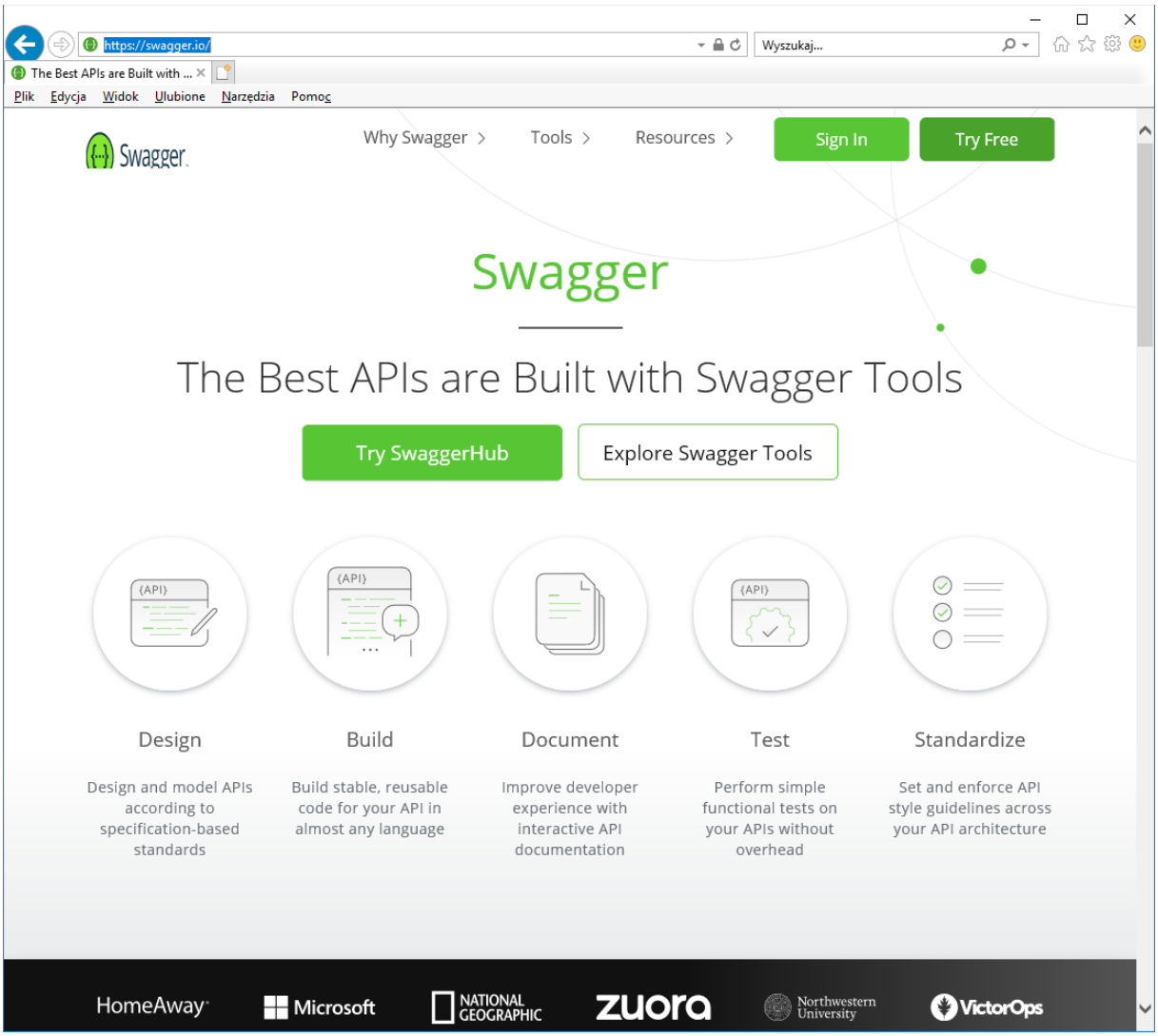
Budowanie kodu klienckiego w C# jest bardziej skomplikowane i wymaga użycia którejś z niskopoziomowych metod wywoływania usług:

```
class Program
 {
     static HttpClient client = new HttpClient();

     static Program()
     {
```

```
        client.DefaultRequestHeaders.Accept.Add(new System.Net.Http.Headers.Med
iaTypeWithQualityHeaderValue("application/json"));
    }

    static void Main(string[] args)
    {
        Get();
        Post();

        Console.ReadLine();
    }

    async static Task Get()
    {
        var responseString = await client.GetStringAsync("http://localhost:6232
7/api/Index");
        var response = JsonConvert.DeserializeObject<IEnumerable<Person>>(respo
nseString);

        foreach ( var person in response)
        {
            Console.WriteLine(string.Format("{0} {1}", person.ID, person.Name))
;
        }
    }

    async static Task Post()
    {
        var person = new Person() { ID = 188, Name = "ghj" };
        var request = JsonConvert.SerializeObject(person);

        StringContent content = new StringContent(request, Encoding.UTF8, "appl
ication/json");
        var response          = await client.PostAsync("http://localhost.fiddle
r:62327/api/Index", content);
        var responseString    = await response.Content.ReadAsStringAsync();
        var personResp        = JsonConvert.DeserializeObject<Person>(responseS
tring);

        Console.WriteLine(string.Format("{0} {1}", personResp.ID, personResp.Na
me));
    }
 }

 public class Person
 {
    public int ID { get; set; }
    public string Name { get; set; }
 }
```

5 OpenAPI / Swagger

Upowszechnianie specyfikacji metadanych REST daje nadzieję na przyszłość – być może wzorem WSDL, usługi typu REST będą mogły być formalnie specyfikowane. W trakcie wykładu obejrzymy specyfikację OpenAPI i jej przykładową implementację – Swagger.



W szczególności zobaczymy jak przy pomocy Swagger zbudować specyfikację usługi odpowiadającej wcześniej pokazanym przykładom oraz jak za pomocą Swagger automatyczne generować kod serwera i klienta odpowiadający zbudowanemu kontraktowi:

```
{
  "swagger" : "2.0",
  "info" : {
    "description" : "This is a Person API",
    "version" : "1.0.0",
    "title" : "Simple Person API",
    "contact" : {
      "email" : "you@your-company.com"
    },
```

```
    "license" : {
      "name" : "Apache 2.0",
      "url" : "http://www.apache.org/licenses/LICENSE-2.0.html"
    }
  },
  "host" : "virtserver.swaggerhub.com",
  "basePath" : "/wzychla/Person2/1.0.0",
  "schemes" : [ "https" ],
  "paths" : {
    "/Person" : {
      "get" : {
        "summary" : "searches persons",
        "description" : "By passing in the appropriate options, you can
search for\navailable inventory in the system\n",
        "operationId" : "getPerson",
        "produces" : [ "application/json" ],
        "parameters" : [ ],
        "responses" : {
          "200" : {
            "description" : "search results matching criteria",
            "schema" : {
              "type" : "array",
              "items" : {
                "$ref" : "#/definitions/PersonItem"
              }
            }
          },
          "400" : {
            "description" : "bad input parameter"
          }
        }
      },
      "post" : {
        "summary" : "adds an inventory item",
        "description" : "Adds an item to the system",
        "operationId" : "postPerson",
        "consumes" : [ "application/json" ],
        "produces" : [ "application/json" ],
        "parameters" : [ {
          "in" : "body",
          "name" : "PersonItem",
          "description" : "Person item to add",
          "required" : false,
          "schema" : {
            "$ref" : "#/definitions/PersonItem"
          }
        } ],
        "responses" : {
          "200" : {
```

```
            "description" : "persons",
            "schema" : {
              "$ref" : "#/definitions/PersonItem"
            }
          },
          "201" : {
            "description" : "item created"
          },
          "400" : {
            "description" : "invalid input, object invalid"
          },
          "409" : {
            "description" : "an existing item already exists"
          }
        }
      }
    }
  },
  "definitions" : {
    "PersonItem" : {
      "type" : "object",
      "required" : [ "ID", "Name" ],
      "properties" : {
        "ID" : {
          "type" : "integer"
        },
        "Name" : {
          "type" : "string",
          "example" : "Widget Adapter"
        }
      }
    }
  }
}
```