Projektowanie aplikacji ASP.NET
Wykład 09/15
WebAPI / REST

Wiktor Zychla 2024/2025

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 lub XML
Język wartości zwracanej	SOAP	JSON lub XML
Metadane	WSDL	Brak oficjalnej specyfikacji, różne konkurujące standardy, 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-*	Istnieją specyfikacje pewnych obszarów (JWS, JWE), innych brak

3 Tworzenie usługi REST - .NET Framework

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);
}
```

W przypadku klasycznego ASP.NET WebAPI należy pamiętać o tym że obsługuje on domyślne dwa sposoby kodowania komunikacji – XML i JSON (REST wcale nie jest tożsame z JSON!), może więc przydać się w konfiguracji wyłączenie formatera XML

```
GlobalConfiguration.Configuration.Formatters.Remove(GlobalConfiguration.Configur
ation.Formatters.XmlFormatter);
```

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()
        });
    }
}
```

Warto w tym miejscu nadmienić, że typ zwracanej odpowiedzi, **IHttpActionResult** trochę przypomina **ActionResult** a sposób jego tworzenia (przez metody fabrykujące konkretną instancję tego typu, na przykład metodę **Ok** która tworzy **HttpActionResult**) przypomina sposób tworzenia odpowiedzi w kontrolerach MVC.

4 Hostowanie usługi REST poza serwerem aplikacji .NET Framework

Podobnie jak w przypadku WCF, podsystem WebAPI jest podatny na hostowanie poza serwerem aplikacji, na przykład w aplikacji konsolowej lub w usłudze systemowej.

Do hostowania usługi służy obiekt **HttpSelfHostServer** z pakietu **Microsoft.AspNet.WebApi.SelfHost**.

```
var config = new HttpSelfHostConfiguration("http://localhost:8087");

 config.Routes.MapHttpRoute(
     "API Default", "api/{controller}/{id}",
     new { id = RouteParameter.Optional });

 using (HttpSelfHostServer server = new HttpSelfHostServer(config))
 {
     server.OpenAsync().Wait();
     Console.WriteLine("Press Enter to quit.");
     Console.ReadLine();
 }
```

5 Tworzenie usługi REST .NET.Core

W .NET Core kontrolery REST i MVC zostały zunifikowane.

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();

var app = builder.Build();

app.UseRouting();

app.UseEndpoints( endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}" );
} );

app.Run();
```

Kontroler REST to niemal zwykły kontroler MVC, tyle że

- Dziedziczy bezpośrednio z klasy bazowe **ControllerBase** (dlaczego?)
- Żeby zachować konwencję routingu, używa atrybutu
- Wymaga stosowania atrybutów **[HttpGet]**, **[HttpPost]**

```
[Route( "api/[controller]" )]
[ApiController]
public class PersonController : ControllerBase
{
    public IActionResult Get()
    {
        var persons = new[]
        {
            new Person() { ID = 1, Name = "person1" },
            new Person() { ID = 2, Name = "person2" }
        };
        return this.Ok( persons );
    }
}

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

6 Tworzenie kodu klienckiego REST

6.1 Klient w Javascript

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

Poniżej zademonstrowano dwa sposoby połączenia:

- Połączenie za pomocą obiektu **XMLHttpRequest** (nie ma już takiej potrzeby)
- Połączenie za pomocą nowszego API **fetch**

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <div>
        <div>
            <button id="invokeXHR">Invoke with XMLHttpRequest</button>
        </div>
        <div>
            <button id="invokeFetch">Invoke with fetch</button>
        </div>
        <div>
            <textarea id="resultArea" cols="80" rows="25">
            </textarea>
        </div>

    </div>
    <script>
        var examplePerson = {
            ID: 12,
            Name: 'foo bar'
        };
        window.addEventListener('load', function () {
            window.invokeXHR.onclick = function () {
                var xhr = new XMLHttpRequest();

                xhr.open('POST', '/api/Person');
                xhr.setRequestHeader("Content-Type", "application/json");
                xhr.onreadystatechange = function () {
                    if (xhr.readyState == XMLHttpRequest.DONE) {
                        window.resultArea.value += xhr.responseText;
                    }
                };
                xhr.send(JSON.stringify(examplePerson));
            };
            window.invokeFetch.onclick = async function() {
                // https://developer.mozilla.org/en-
US/docs/Web/API/Fetch_API/Using_Fetch
                var result = await fetch('/api/Person', {
                    method: 'POST',
                    headers: {
```

```
                        'Content-type': 'application/json'
                    },
                    body: JSON.stringify(examplePerson)
                });
                var resultText = await result.text();
                window.resultArea.value += resultText;
            };
        });
    </script>
</body>
</html>
```

6.2 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: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; }
 }
```

7 Uwierzytelnienie komunikacji z usługami REST

Istnieją dwa sposoby zabezpieczenia komunikacji aplikacji z usługą REST.

7.1 Autentykacja ciastkiem sesji

Pierwszy sposób polega na wykorzystaniu ciastek uwierzytelnienia, które aplikacja wydaje w klasyczny sposób. Ponieważ żądania do serwera w ramach tej samej domeny są wykonywane w taki sposób, że do żądania są dodawane automatycznie wszystkie ciastka wydane dla domeny, również komunikacja z usługą REST będzie niosła za sobą wszystkie potrzebne ciastka. Od strony serwerowej zabezpieczenie usługi wymaga wyłącznie dodania atrybutu **Authorize** (tym razem jest to **System.Web.Http.Authorize**), atrybut zachowuje się tak samo jak znany już nam **Authorize** z MVC (**System.Web.Mvc.Authorize**).

W przypadku braku ciastka z danymi użytkownika, serwer aplikacji przekierowuje nieuwierzytelnione żądanie do strony logowania.

Dla klienta który spodziewa się odpowiedzi w formacie JSON może to być pewien problem. Dlatego bywa, że zamiast standardowego, wbudowanego atrybutu **Authorize**, pisze się własne filtry (filtry omawialiśmy przy okazji MVC) które zamiast przekierowania na stronę logowania zwracają jakiś omówiony kod statusu http (np. **451** albo inny nieużywany typowo) po którym klient może zorientować się że chodzi o żądanie które nie zostało poprawnie autentykowane.

W teorii, zamiast przekierowania **302** na stronę logowania można zwrócić po prostu **401** czyli błąd autentykacji, natomiast powstaje tu ryzyko że przeglądarka lub jakiś element infrastruktury leżący pomiędzy przeglądarką na serwerem zareaguje na **401** w jakiś nieoczekiwany z punktu widzenia klienta sposób (na przykład gdyby klient-przeglądarka w odpowiedzi na 401 pokazał to standardowe okno do wpisania pary login/hasło to z punktu widzenia użytkownika byłoby to wysoce niewłaściwe!)

Należy zwrócić uwagę, że klient przeglądarkowy używający **XMLHTTPRequest** lub **fetch** może wymagać dodatkowe konfiguracji żeby do żądań dodawać bieżące ciasteczka!

Por.
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/withCredentials

7.2 Autentykacja własnym nagłówkiem autentykującym

Ciastka są wygodnym sposobem zabezpieczania dostępu do usług w sytuacji gdy zarówno usługa jak i jej klient pochodzą z tego samego „źródła" – aplikacja przeglądarkowa która pozyskuje ciastko podczas logowania jest dobrym przykładem. W takim przypadku przeglądarka sama dołącza przecież ciastka do żądań do serwera i nie ma znaczenia czy to są żądania POST formularzy czy żądania wysyłane przez Javascript (jak w przykładzie klienta Javascript wyżej).

Natomiast w scenariuszu w którym klient i usługa są od siebie **niezależne** (na przykład mówimy o komunikacji serwer-serwer albo aplikacja mobilna-serwer), pozyskanie ciastka uwierzytelniania może być niełatwe – w tym celu trzeba by bowiem na kliencie powtarzać potok uwierzytelniania. A to z kolei

może być nawet niemożliwe – przecież serwer może delegować uwierzytelnianie do innego serwera (np. do Google).

W takich przypadkach, w celu zabezpieczenia usługi po stronie serwera, zamiast opierać się na ciastkach, należy usługę opatrzyć atrybutem – filtrem autentykującym.

Zadaniem filtra autentykującego jest albo utworzenie obiektu **IPrincipal** na podstawie jakichś parametrów żądania (na przykład właśnie – nagłówka) albo zwrócenie statusu błędu.

Przykładowy filtr autentykujący reaguje na nagłówek **api_key** z wartością **123** i zamienia go na tożsamość **user**. W rzeczywistej aplikacji różni klienci mogliby dostać różne poprawne nagłówki autentykujące a wartość **IPrincipal** mogłaby być ustawiana na poprawną nazwę klienta, w zależności od tego jaką wartością nagłówka autentykującego się przedstawia.

W poniższej implementacji jedyne na co warto zwrócić uwagę to to, że metody interfejsu filtra są zaprojektowane jako asynchroniczne (zwracają **Task**).

```
public class CustomAuthenticationFilter : Attribute, IAuthenticationFilter
{
    const string header_name  = "api_key";
    const string header_value = "123";

    public bool AllowMultiple
    {
        get
        {
            return true;
        }
    }

    public Task AuthenticateAsync(
        HttpAuthenticationContext context,
        CancellationToken cancellationToken)
    {
        var request = context.Request;
        IEnumerable<string> headers;
        if ( request.Headers.TryGetValues(header_name, out headers) )
        {
            var value = headers.FirstOrDefault();

            if ( value == header_value )
            {
                context.Principal =
                    new GenericPrincipal(
                        new GenericIdentity("user", "authenticated"),
                        new string[0]);

                return Task.FromResult(0);
            }
        }

        // fallback - brak nagłówka lub niepoprawny nagłówek
        context.ErrorResult =
            new ResponseMessageResult(
                request.CreateErrorResponse(HttpStatusCode.Unauthorized, "unauthor
ized") );

        return Task.FromResult(0);
```

```
    }

    public Task ChallengeAsync(
        HttpAuthenticationChallengeContext context,
        CancellationToken cancellationToken)
    {
        return Task.FromResult(0);
    }
}
```

Zabezpieczenie kontrolera wymaga atrybutu filtra autentykującego:

```
[CustomAuthenticationFilter]
public class UserController : ApiController
{
```

Od strony klienta – dodanie odpowiedniego nagłówka jest zawsze możliwe, bez względu na to czy mowa jest o kliencie Javascript w przeglądarce czy kliencie z aplikacji serwerowej. W przypadku klienta opartego o **HttpClient** z biblioteki standardowej .NET, warto wiedzieć, że najogólniejszą metodą do tworzenia żądań jest **SendAsync** która wymaga całego obiektu żądania. Tylko w ten sposób można łatwo wysyłać różne wartości nagłówków z każdym żądaniem.

```
HttpClient client = new HttpClient();

// Add an Accept header for JSON format.
client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("appli
cation/json"));

HttpRequestMessage request = new HttpRequestMessage()
{
    Method     = HttpMethod.Get,
    RequestUri = new Uri("http://localhost:49774/api/User")
};

request.Headers.Add("api_key", "123");

HttpResponseMessage response = client.SendAsync( request ).Result;
var users = response.Content.ReadAsAsync<IEnumerable<UserDTO>>().Result;

foreach (var user in users)
{
    Console.WriteLine("{0} {1}", user.Name, user.Surname);
}
```

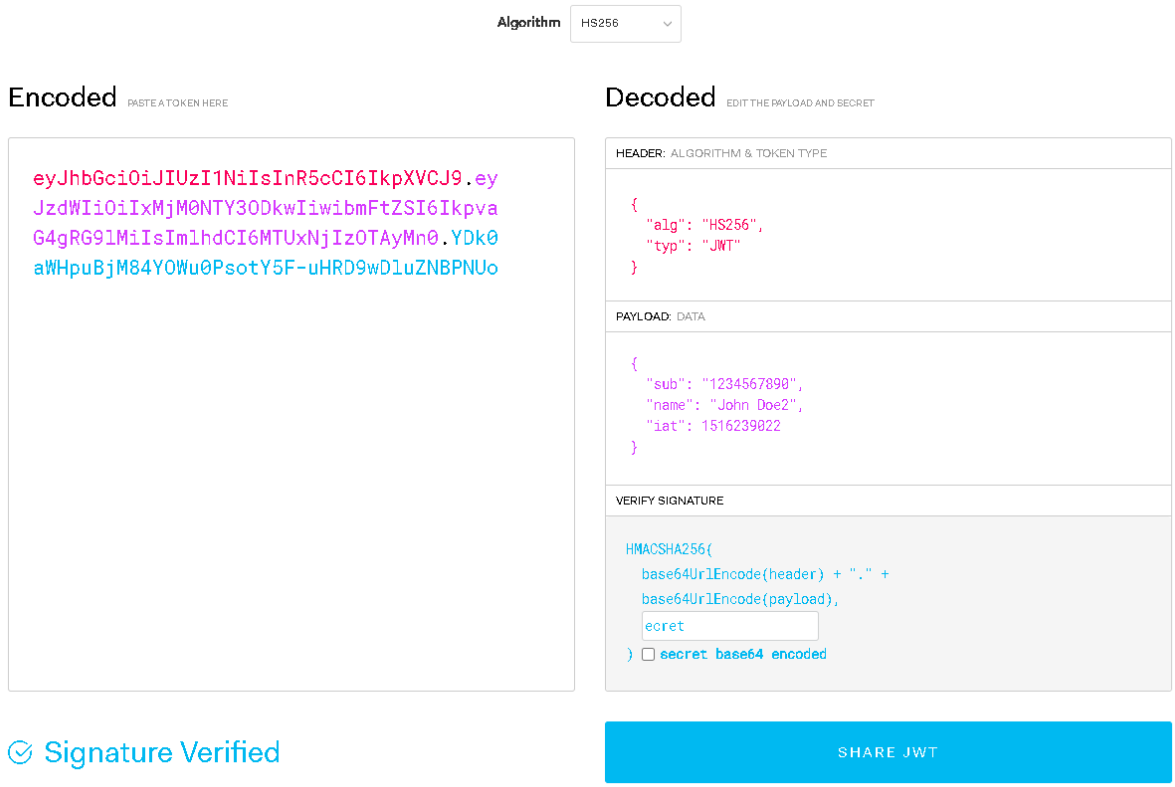
7.3 JSON Web Tokens (JWT)

Alternatywny sposób autentykacji żądań polega na wykorzystaniu jakiegoś przemysłowego standardu uwierzytelniania, który nie opiera się na ciastkach w ramach tej samej domeny, tylko na jawnym nagłówku autentykującym. Jednym z dobrze przyjętych standardów jest JSON Web Token.

Ten sposób jest częściej wykorzystywany wtedy, kiedy za pomocą REST komunikują się dwa serwery – do poprawnego skonstruowania tokena JWT niezbędny jest bowiem *klucz prywatny komunikacji* – w zależności od wybranego sposobu podpisywania tokena będzie to albo hasło współdzielone między oboma uczestnikami komunikacji albo klucz prywatny certyfikatu którym podpisuje się token.

Należy jednak zauważyć, że zamiast JWT można wykorzystać tu jakikolwiek sposób przekazania klucza autentykującego, w ostateczności takim kluczem może być umówiony ciąg znaków, przekazany w nagłówku o umówionej nazwie.

W przypadku obsługi JWT, filtr autentykacji byłby bardzo podobny do tego z przykładu powyżej, tylko w miejscu w którym w przykładzie powyżej następuje porównanie wartości nagłówka przysłanego przez klienta z wartością oczekiwaną przez serwer, należałoby użyć dowolnej techniki obsługi tokenów JWT.



Przykładem takiej biblioteki dla .NET jest **System.IdentityModel.Tokens.JWT** (możliwe do zainstalowania z nuget). Przykładowy kod generowania i weryfikowania tokenów JWT dla .NET Framework:

```
using Microsoft.IdentityModel.Tokens;
using System;
using System.Collections.Generic;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

namespace ConsoleApplication
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            var plainTextSecurityKey = "This is my shared, not so secret, secret
!";

            var signingKey = new Microsoft.IdentityModel.Tokens.SymmetricSecurit
yKey(
                Encoding.UTF8.GetBytes(plainTextSecurityKey));

            var signingCredentials = new Microsoft.IdentityModel.Tokens.SigningC
redentials(signingKey,
                Microsoft.IdentityModel.Tokens.SecurityAlgorithms.HmacSha256Sign
ature);

            // ------------------------

            var claimsIdentity = new ClaimsIdentity(new List<Claim>()
            {
                new Claim(ClaimTypes.Name, "myemail@myprovider.com"),
                new Claim(ClaimTypes.Role, "Administrator"),
            }, "Custom");

            var securityTokenDescriptor = new Microsoft.IdentityModel.Tokens.Sec
urityTokenDescriptor()
            {
                Audience = "http://my.website.com",
                Issuer = "http://my.tokenissuer.com",

                Subject = claimsIdentity,
                SigningCredentials = signingCredentials
            };

            var tokenHandler = new JwtSecurityTokenHandler();
            var plainToken = tokenHandler.CreateToken(securityTokenDescriptor);
            var signedAndEncodedToken = tokenHandler.WriteToken(plainToken);

            Console.WriteLine(plainToken);
            Console.WriteLine(signedAndEncodedToken);

            // ------------------------

            var tokenValidationParameters = new TokenValidationParameters()
            {
                ValidAudiences = new string[]
                {
                    "http://my.website.com",
                    "http://my.otherwebsite.com"
                },
                ValidIssuers = new string[]
                {
                    "http://my.tokenissuer.com",
                    "http://my.othertokenissuer.com"
                },
                IssuerSigningKey = signingKey
            };

            Microsoft.IdentityModel.Tokens.SecurityToken validatedToken;
            var validatedPrincipal = tokenHandler.ValidateToken(signedAndEncoded
Token,
```

```
                tokenValidationParameters, out validatedToken);

            Console.WriteLine(validatedPrincipal.Identity.Name);
            Console.WriteLine(validatedToken.ToString());

            Console.ReadLine();
        }
    }
}
```

W przypadku .NET.Core sytuacja jest prostsza. Istnieje pakiet **Microsoft.AspNetCore.Authentication.JwtBearer** którego dodanie pozwala obsługiwać tokeny JWT po stronie serwera wprost. Przy okazji proszę zwrócić uwagę jak wymusza się stosowanie określonego schematu uwierzytelniania:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using System.Text;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuthentication()
    .AddJwtBearer( cfg =>
    {
        var plainTextSecurityKey = "This is secret";

        var signingKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(pl
ainTextSecurityKey));
        var signingCredentials = new SigningCredentials(signingKey,Security
Algorithms.HmacSha256Signature);

        cfg.RequireHttpsMetadata = false;
        cfg.TokenValidationParameters = new TokenValidationParameters()
        {
            ValidateAudience = false,
            ValidateIssuer = false,
            IssuerSigningKey = signingKey
        };

        cfg.Events = new JwtBearerEvents()
        {
            OnAuthenticationFailed = async context =>
            {
                var ex = context.Exception;
                Console.WriteLine( ex.Message );
            }
        };
    } );

builder.Services.AddControllers();

var app = builder.Build();
```

```
app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();

app.Run();

[Authorize( AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme
)]
[ApiController]
[Route( "api/[controller]" )]
public class ExampleController : ControllerBase
{

    public ExampleController()
    {
    }

    [HttpGet]
    public IActionResult Get()
    {
        return this.Ok( "webapi service. authenticated as " + this.User.Ide
ntity.Name );
    }
}
```

Kod przykładowego klienta dla .NET Core:

```
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

var plainTextSecurityKey = "This is secret";

var signingKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(plainTextS
ecurityKey));
var signingCredentials = new SigningCredentials(signingKey, SecurityAlgorit
hms.HmacSha256Signature);

var claimsIdentity =
    new ClaimsIdentity(new List<Claim>()
    {
        new Claim(JwtRegisteredClaimNames.Name, "username1")
    });

var securityTokenDescriptor = new SecurityTokenDescriptor()
{
    Subject = claimsIdentity,
    SigningCredentials = signingCredentials
};

var tokenHandler          = new JwtSecurityTokenHandler();
var plainToken            = tokenHandler.CreateToken(securityTokenDescripto
r);
var signedAndEncodedToken = tokenHandler.WriteToken(plainToken);
```

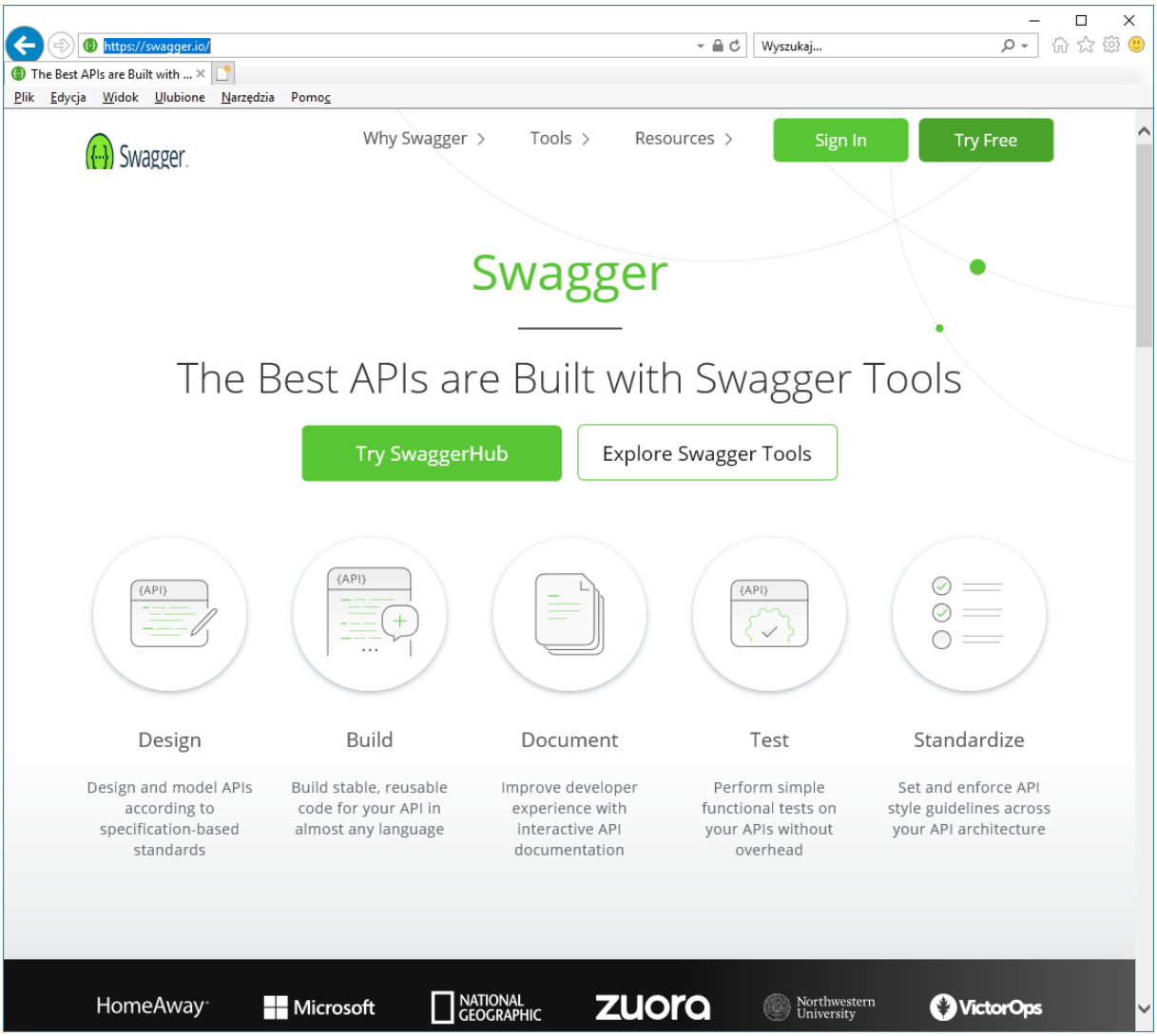
```
Console.WriteLine( plainToken );
Console.WriteLine( signedAndEncodedToken );

HttpClient client = new HttpClient();
client.DefaultRequestHeaders.Add( "Authorization", $"Bearer {signedAndEncod
edToken}" );

var res = await client.GetAsync("http://localhost:5198/api/Example");
if ( res.StatusCode == System.Net.HttpStatusCode.OK )
{
    var result = await res.Content.ReadAsStringAsync();
    Console.WriteLine( result );
}
else
{
    Console.WriteLine( $"Status: {res.StatusCode}" );
}

Console.ReadLine();
```

Bardziej rozbudowany przykład aplikacji, w której równocześnie użyto ciasteczek do logowania i tokenów JWT do autentykacji punktów końcowych WebAPI znajduje się tu:

https://github.com/wzychla/AspJwtBearerDemo

8 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"
        }
      }
    }
  }
}
```

.NET Core wspiera tę specyfikację natywnie. Po dodaniu pakietu **Swashbuckle.AspNetCore** możliwe jest automatycznie generowanie dokumentacji:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddSwaggerGen();

var app = builder.Build();

app.UseSwagger();
app.UseSwaggerUI();

app.UseAuthentication();
app.UseAuthorization();
```

```
app.MapControllers();

app.Run();
```

Automatycznie generowana dokumentacja znajduje się pod adresami:

- /swagger – interfejs UI
- /swagger/v1/swagger.json – opis

Więcej na ten temat w dokumentacji.

9 Rekomendacje architektury

Z powyższych rozważań można zbudować następujące rekomendacje dotyczące architektury aplikacji korzystających z usług sieciowych.

9.1 Aplikacja kliencka (np. React) i usługa siecowa w jednym

Jeżeli naszym zadaniem jest wytworzenie zarówno aplikacji klienckiej jak i usługi sieciowej, to zdecydowanie należy rozważyć dostarczanie obu części z poziomu **jednej** witryny (obsługiwanej przez jeden nagłówek hosta).

Często popełnianym w takim przypadku błędem jest budowanie dwóch osobnych witryn, jedna dostarcza części klienckiej (React), druga usług WebAPI. Problemem takiego podejścia jest to że aplikacja internetowa odwołująca się do usługi obsługiwanej przez inną wiitrynę jest przez przeglądarkę ograniczana (tzw. CORS – będziemy jeszcze o tym mówić).

Autentykację w takim przypadku najłatwiej oprzeć na ciasteczkach. Strona logowania aplikacji jest obsługiwana w sposób standardowy, z serwera (tak jak do tej pory budowaliśmy autentykację). Aplikacja kliencka (React) jest hostowana na stronie, która wymaga od użytkownika zalogowania się za pomocą tej standardowej strony logowania, obsługiwanej przez przekierowania.

Taki sposób obsługi uwierzytelniania daje możliwość bezpiecznego zintegrowania aplikacji z zewnętrznymi dostawcami tożsamości (np. Google) – strona logowania deleguje uwierzytelnianie dalej, do Google, a aplikacja kliencka i tak ładowana jest dopiero po zalogowaniu się.

9.2 Aplikacja kliencka i niezależna usługa sieciowa

Jeżeli naszym zadaniem jest wytworzenie bądź to samej usługi bądź samego klienta usługi, ale jedna i druga część są od siebie faktycznie niezależne, należy rozważyć uwierzytelnianie za pomocą nagłówków, być może w oparciu o standard JWT.

W takim przypadku aplikacja kliencka może albo po prostu mieć dostęp do tokena autentykującego (bo jest to komunikacja serwer-serwer) albo aplikacja kliencka może prosić użytkownika o wpisanie tokena/loginu/hasła autentykacji (i dodawać stosowny nagłówek do każdego żądania).

Proszę zwrócić uwagę, że aplikacja kliencka wcale nie musi umieć sama wytworzyć tokena JWT. Wystarczy po stronie serwera usługa, która z jednej nie wymaga autentykacji, ale przesłaną poprawną parę login/hasło zamienia na token JWT.