

LABORATORIUM 9. WYKORZYSTANIE UWIERZYTELNIENIA W APLIKACJI TYPU API

Cel laboratorium:

Celem zajęć jest opracowanie aplikacji internetowej, która wykorzystuje uwierzytelnianie i autoryzację za pomocą tokenów (znaczników) JWT.

Liczba punktów możliwych do uzyskania: 6 punktów

Zakres tematyczny zajęć:

Dodawanie podsystemu Identity do istniejącego projektu,
Modyfikacja wymagań podsystemu Identity,
Generowanie i testowanie tokenów JWT,
Uwierzytelnianie za pomocą tokenów JWT,
Sprawdzanie roli przechowywanej w tokenie.

Pytania kontrolne:

1. W jaki sposób działa mechanizm „ciasteczek” (*cookies*)?
2. W jaki sposób działają podpisy cyfrowe?
3. Na czym polega bezstanowość protokołu HTTP?

Zadanie 9.1. Dodawanie ASP.NET Identity do istniejącego projektu

Pobierz z platformy Moodle przykładowy projekt „Lab9”. Jest to projekt wzorowany na projekcie, który opracowywany był w Laboratorium 8, jednak został nieznacznie zmodyfikowany, m.in. implementacja interfejsu `IFoxesRepository` opiera się o listę w pamięci, a nie o rzeczywistą implementację bazy danych. Usunięty został też mechanizm uwierzytelnienia w oparciu o HTTP Basic Authentication.

W tym laboratorium będziemy wykorzystywać ASP.NET Identity do przechowywania danych użytkowników, stąd niezbędne będzie jego dodanie do już istniejącego projektu.

Przejdź do emulatora konsoli, narzędzia Terminal lub Wiersza Polecenia, a w nim do folderu projektu Lab9 i wydaj komendy:

```
dotnet add package
Microsoft.VisualStudio.Web.CodeGeneration.Design --version 6.0
dotnet add package Microsoft.EntityFrameworkCore.Design --
version 6.0
dotnet add package
Microsoft.AspNetCore.Identity.EntityFrameworkCore --version
6.0
dotnet add package Microsoft.AspNetCore.Identity.UI --version
6.0
dotnet add package Microsoft.EntityFrameworkCore.Sqlite --
version 6.0
```



```
dotnet add package Microsoft.EntityFrameworkCore.Tools --  
version 6.0  
  
dotnet aspnet-codegenerator identity -sqlite  
dotnet ef migrations add Init
```

Spowoduje to wygenerowanie elementów systemu Identity, utworzenie migracji schematu bazy danych i dołączenie niezbędnych bibliotek do działania systemu. Utworzone zostaną także migracje, które należy zmodyfikować w celu dodania wartości początkowych.

Otwórz plik w folderze Migrations/ o nazwie kończącej się na `_Init.cs` i w metodzie `Up()`, po utworzeniu wszystkich tabel, relacji i indeksów, dodaj następujący kod:

```
migrationBuilder.InsertData(  
    "AspNetRoles",  
    new string[] { "Id", "Name", "NormalizedName",  
"ConcurrencyStamp" },  
    new object[]  
    {  
        Guid.NewGuid().ToString(),  
        "Admin",  
        "ADMIN",  
        Guid.NewGuid().ToString()  
    }  
);
```

Spowoduje on dodanie danych do bazy przy aplikowaniu migracji – zostanie dodana rola o nazwie „Admin”. Role to mechanizm grupowania uprawnień do wykonywania czynności, wykorzystywany w ASP.NET Identity. W dalszej części laboratorium będziemy sprawdzać, czy użytkownik przynależy do pewnej roli.

W pliku `Program.cs` dodaj:

```
app.UseRouting();
```

Powyżej wywołania `UseAuthentication()`, natomiast poniżej `MapControllers()` dodaj:

```
app.MapRazorPages();
```

Spowoduje to możliwość działania jednocześnie kontrolerów API, jak i stron Razor Pages w jednej aplikacji – mechanizm Identity opiera się o Razor Pages.

Wykonaj utworzenie bazy danych wydając komendę w głównym folderze swojej aplikacji:

```
dotnet ef database update
```



Zadanie 9.2. Dostosowywanie mechanizmu Identity

Jak pamiętasz z poprzedniego laboratorium wykorzystującego Identity, domyślne wymagania dotyczące hasła są bardzo skomplikowane. Można to jednak zmodyfikować, modyfikując ustawienia podsystemu Identity.

W pliku `Program.cs` znajdź wywołanie dodawania obsługi podsystemu Identity, o nazwie `builder.Service.AddDefaultIdentity()` i je zmodyfikuj, używając następującego fragmentu kodu:

```
builder.Services
    .AddDefaultIdentity<IdentityUser>(options =>
    {
        options.SignIn.RequireConfirmedAccount = true;
        options.Password.RequireUppercase = false;
    })
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<Lab9IdentityDbContext>();
```

Funkcja `AddRoles()` powoduje uaktywnienie podsystemu ról. Z kolei ustawienia parametrów `options.Password` pozwalają na sterowanie wymogami dotyczącymi złożoności hasła.

Zmodyfikuj te opcje tak, aby nie były wymagane wielkie litery, znaki niealfanumeryczne i cyfry, ale za to minimalna długość hasła powinna wynosić 8 znaków – tak, aby mogło zadziałać np. hasło „password”.

Zadanie 9.3. Kontroler dostarczający tokeny dla użytkownika

Utwórz nową klasę, `UserDto`, która będzie posiadać dwie właściwości typu `string`: `UserName` oraz `Password`. Będzie ona modelem danych, który będzie przysyłał użytkownik do specjalnego kontrolera, który na tej podstawie będzie generował token, a użytkownik uwierzytelniał się tokenem od tej pory.

Dodaj do swojego projektu bibliotekę do obsługi tokenów JWT:

```
dotnet add package
Microsoft.AspNetCore.Authentication.JwtBearer --version 6.0
```

Utwórz nowy, pusty kontroler typu API o nazwie `UserController`. Dodaj w konstruktorze aby wstrzykiwane do niego były `UserManager<IdentityUser>` oraz `IConfiguration`, które zapiszesz do prywatnych pól o nazwie `_user` oraz `_configuration`, odpowiednio.

Dodaj do kontrolera akcję `Token()`, która będzie zwracać użytkownikowi token na podstawie obiektu DTO:

```
[HttpPost]
public async Task<IActionResult> Token([FromBody] UserDto dto)
{
```



```
var user = await _user.FindByEmailAsync(dto.UserName);
var result = await _user.CheckPasswordAsync(user,
dto.Password);

if (result)
{
    var claims = new List<Claim>()
    {
        new Claim(JwtRegisteredClaimNames.Sub,
user.Email),
        new Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString()),
        new Claim(JwtRegisteredClaimNames.Email,
user.Email),
    };

    var key = new SymmetricSecurityKey(
        Encoding.UTF8.GetBytes(_configuration["Tokens:Key"
])
    );

    var creds = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);

    var token = new JwtSecurityToken(
        issuer: _configuration["Tokens:Issuer"],
        audience: _configuration["Tokens:Audience"],
        claims: claims,
        expires: DateTime.UtcNow.AddMinutes(60),
        signingCredentials: creds
    );

    return Ok(
        new
        {
            token = new
JwtSecurityTokenHandler().WriteToken(token),
            expiration = token.ValidTo
        }
    );
}
```

```

        );
    }
    else
    {
        return BadRequest("Login Failure");
    }
}

```

Wykorzystaj następujące przestrzenie nazw:

```

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

```

Metoda ta sprawdza czy kombinacja nazwy użytkownika i hasła jest poprawna, a jeśli tak, to generuje token JWT, wystawiając go dla konkretnego serwera i podpisując tajnym kluczem szyfrującym, który jest pobierany z konfiguracji. Ważność tokena została ustawiona na 60 minut.

Tokeny JWT (JSON Web Tokens) to otwarty standard przenoszenia informacji o użytkowniku. Ciąg znaków, którym jest *de facto* token, to podpisany cyfrowo zakodowany obiekt, który może informować kto go wystawił, dla jakich odbiorców, dla kogo i jakie informacje ma jego właściciel.

Stąd, w pliku `appsettings.json` dodaj konfigurację ustawień wymaganych przez ten mechanizm – informacje wykorzystywane jako dane dla wystawcy, odbiorcy i tajny klucz podpisu cyfrowego (możesz go zmienić):

```

"Tokens": {
  "Audience": "http://localhost:5010",
  "Issuer": "http://localhost:5010",
  "Key": "bardzo tajny klucz szyfrujący"
}

```

Ustawienia te są traktowane jako tajne i nie powinny być przechowywane w kodzie aplikacji. Tutaj stosujemy plik konfiguracyjny, w aplikacjach produkcyjnych dane te powinny być dostarczone przez mechanizm sekretów.

Z uwagi na to, że ustaliliśmy, że dostawca i odbiorca tokena będzie pod znanym adresem URL należy zmienić, aby aplikacja korzystała z tego adresu, a nie z losowo generowanego portu jak zazwyczaj, aby sprawdzanie wystawcy i odbiorcy tokena działało prawidłowo. W pliku `Properties/launchSettings.json` zmień opcję `applicationUrl` na <http://localhost:5010> wewnątrz profilu o nazwie Lab9.

Zadanie 9.4. Migracja użytkowników

W pliku `Program.cs` powyżej wywołania `app.Run()` dodaj fragment kodu:

```

using (var scope = app.Services.CreateScope())

```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



```
{
    using (var roleManager =
scope.ServiceProvider.GetService<RoleManager<IdentityRole>>())
    using (var userManager =
scope.ServiceProvider.GetService<UserManager<IdentityUser>>())
    {
        roleManager.CreateAsync(new
IdentityRole("Admin")).Wait();

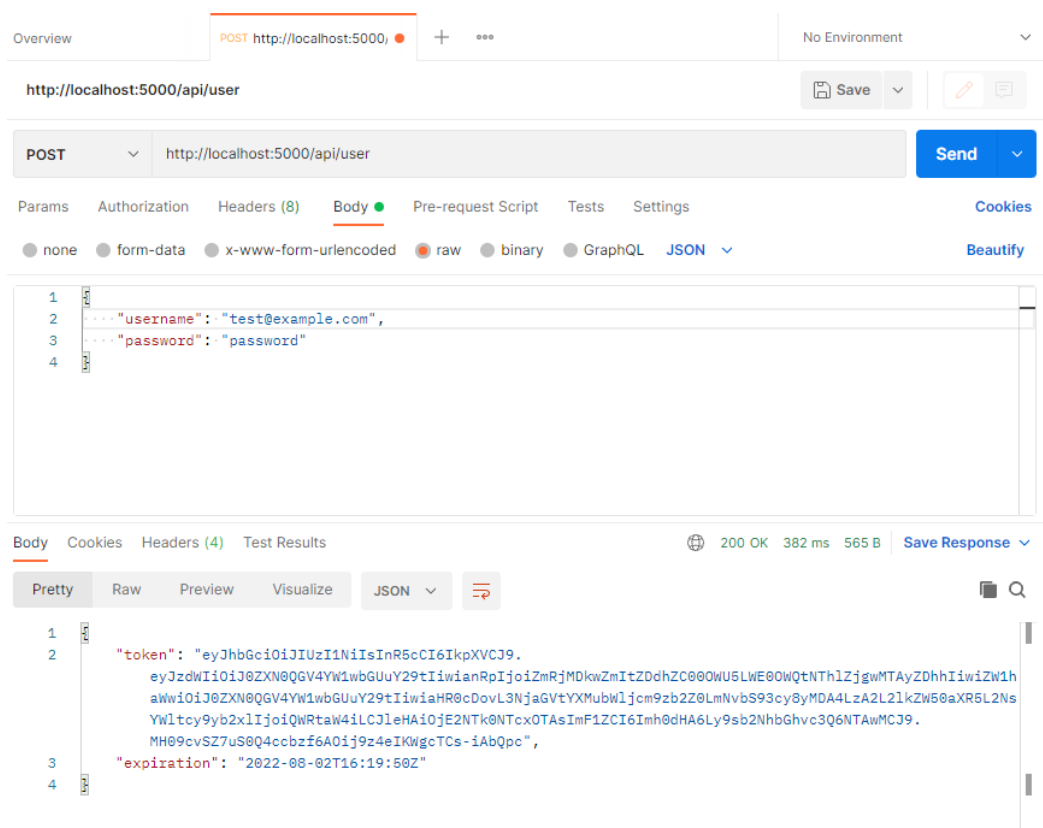
        foreach (var user in userManager.Users.Where(x =>
x.Email.EndsWith("@example.com")))
        {
            userManager.AddToRoleAsync(user, "Admin").Wait();
        }
    }
}
```

Spowoduje on, że wszyscy użytkownicy zapisani na e-mail kończący się na @example.com zostaną przeniesieni do roli „Admin”. W tym przykładzie wykorzystywany jest fragment kodu uruchamiany tuż przed właściwym uruchomieniem aplikacji – będzie to powodowało generowanie ostrzeżeń

Uruchom swoją aplikację i zarejestruj nowego użytkownika na adres e-mail kończący się w ten sposób, następnie zrestartuj ją, aby zadziałała migracja, i użyj narzędzia typu Postman lub Insomnia wysyłając żądanie:



- Typu POST, na adres <http://localhost:5010/api/user>,
- Z zawartością w postaci obiektu JSON: { "userName": "string", "password": "string" } (oczywiście zamiast „string” podaj nazwę użytkownika i hasło zarejestrowanego użytkownika),
- Z nagłówkiem Content-Type ustawionym na application/json.

Przykład w narzędziu Postman przedstawiono na rysunku 9.1.



Rys 9.1. Przykład żądania do pobrania tokenu JWT w narzędziu Postman.

Powinien zostać ci zwrócony obiekt, którego pierwsze pole, token, to długi ciąg znaków. Możesz wkleić ten ciąg znaków do aplikacji internetowej <https://jwt.io>, aby zobaczyć jego strukturę wewnętrzną:


Debugger Libraries Introduction Ask
Crafted by  auth0

Warning: JWTs are credentials, which can grant access to resources. Be careful where you paste them! We do not record tokens, all validation and debugging is done on the client side.

Algorithm
HS256

Encoded
PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0ZXN0QGV4YW1wbGUuY29tIiwianRpIjoizmRjMDkwZmItZDdhZC00OUU5LWE0WQTNhIjZjgMTAyZDhhIiwiaWwiOiJ0ZXN0QGV4YW1wbGUuY29tIiwiaHR0cDovL3NjaGVtYXMubWljcm9zb2Z0LmNvbS93cy8yMDA4LzA2L2lkZW50aXR5L2NsYWltcy9yb2x1IjoiaWwtaW4iLCJleHAiOjE2NTk0NTcxOTAsImF1ZCI6Imh0dHA6Ly9sb2NhbnhGhvc3Q6NTAwMCJ9.MH09cvSZ7uS0Q4ccbf6A0ij9z4eIKWgcTCs-iAbQpc
```

Decoded
EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "test@example.com",
  "jti": "fdc090fb-d7ad-49e9-a49d-58ef80102d8a",
  "email": "test@example.com",
  "http://schemas.microsoft.com/ws/2008/06/identity/claims/role": "Admin",
  "exp": 1659457190,
  "aud": "http://localhost:5000"
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Rys 9.2. Struktura tokena JWT

Zadanie 9.5. Uwierzytelnianie za pomocą tokena

Chcemy, aby metody w kontrolerze `FoxController` były dostępne tylko za pośrednictwem tokena. Chcemy jednocześnie, aby mechanizm „ciasteczek” (*cookies*) za pomocą którego użytkownik może używać ASP.NET Identity w przeglądarce internetowej również działał poprawnie. Należy zatem odpowiednio skonfigurować uwierzytelnienie.

W pliku `Program.cs` powyżej wywołania metody `builder.Build()` dodaj opcje uwierzytelnienia:

```
builder.Services
    .AddAuthentication()
    .AddCookie()
    .AddJwtBearer(
        JwtBearerDefaults.AuthenticationScheme,
        options =>
        {
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita Polska

Unia Europejska
Europejski Fundusz Społeczny




```
options.TokenValidationParameters = new
TokenValidationParameters
{
    ValidateIssuer = true,
    ValidateAudience = true,
    ValidateLifetime = true,
    ValidateIssuerSigningKey = true,
    ValidIssuer =
builder.Configuration["Tokens:Issuer"],
    ValidAudience =
builder.Configuration["Tokens:Audience"],
    IssuerSigningKey = new SymmetricSecurityKey(
        Encoding.UTF8.GetBytes(builder.Configuration["Tokens:Key"]))
};
);

builder.Services.AddAuthorization();
```

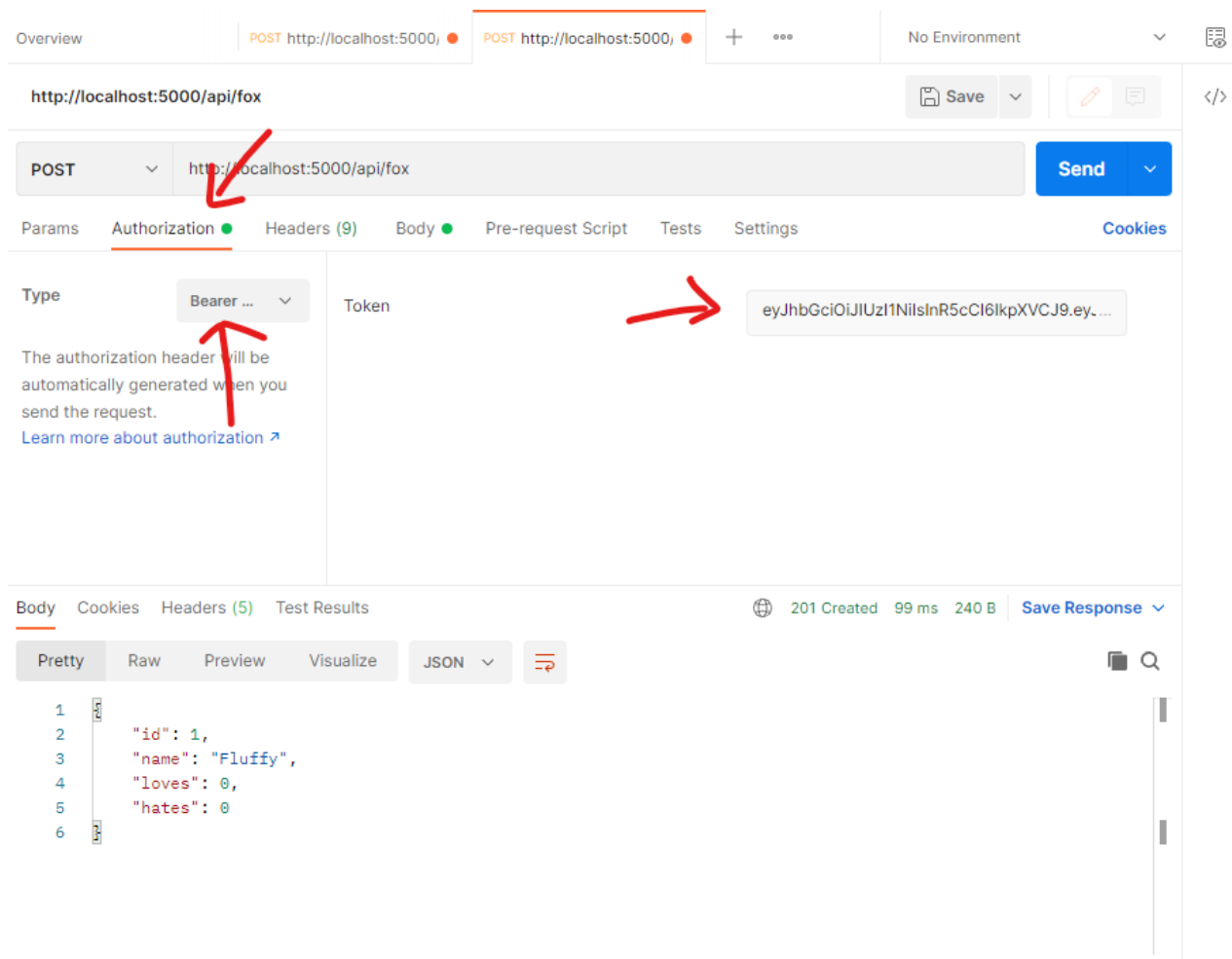
Definiowane tutaj jest, że token, aby mógł zostać zaakceptowany, musi być podpisany znanym kluczem, przeznaczony dla odpowiedniego odbiorcy, wystawiony przez znanego dostawcę i nie może być wygaśnięty. Wcześniej, aktywowany jest mechanizm *cookies* ze swoimi standardowymi parametrami.

Następnie, w metodzie akcji `Post()` kontrolera `FoxController` dodaj atrybut:

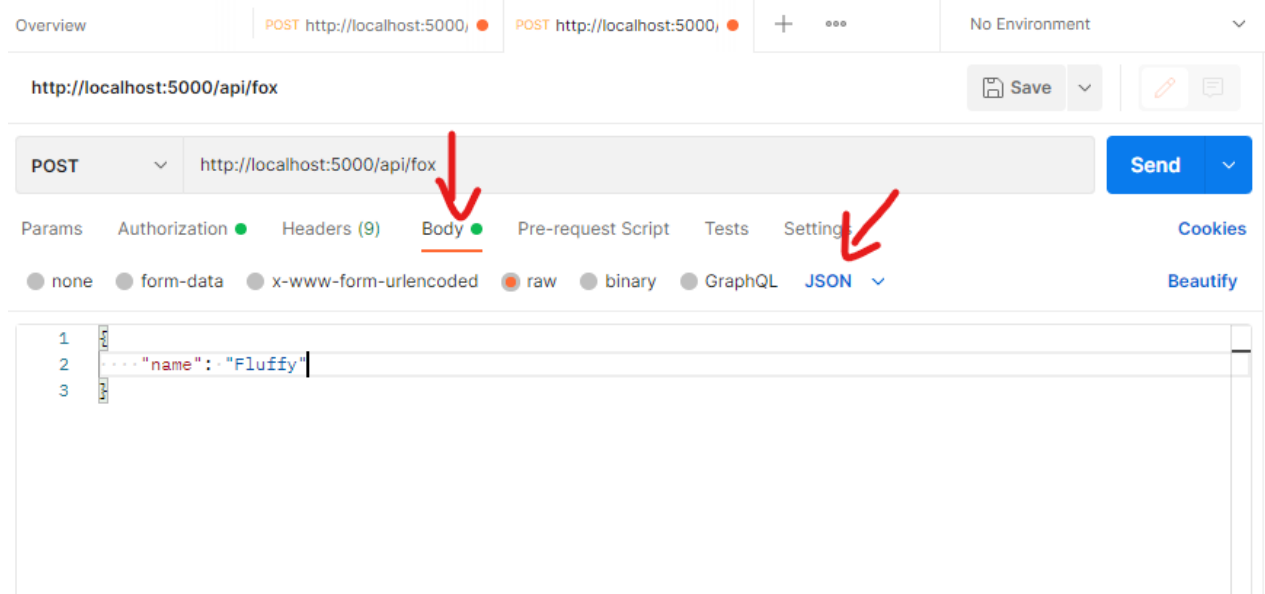
```
[Authorize(AuthenticationSchemes =
JwtBearerDefaults.AuthenticationScheme)]
```

Spowoduje to, że akcja ta będzie dostępna tylko dla użytkowników uwierzytelnionych za pomocą tokena.

Przetestuj działanie aplikacji, wysyłając żądanie uwierzytelnione tokenem, który zwrócił kontroler `UserController`, przykładowo w aplikacji Postman będzie to wyglądało następująco:



Rys 9.3. Ustawianie uwierzytelnienia w narzędziu Postman.



Rys 9.4. Ustawianie przesyłanych danych w narzędziu Postman



Zadanie 9.6. Dostosowywanie reguł bezpieczeństwa

Chcemy, aby akcje `Love()` i `Hate()` kontrolera `FoxController` były dostępne dla wszystkich użytkowników uwierzytelnionych za pomocą tokena, ale aby akcja `Post()` była dostępna tylko dla użytkowników będących w roli administratora.

Dodaj nagłówek uwierzytelnienia do akcji `Love()` i `Hate()`:

```
[Authorize(AuthenticationSchemes =  
JwtBearerDefaults.AuthenticationScheme)]
```

Następnie, w pliku `Program.cs` zmień wywołanie metody `AddAuthorization()`, aby było wykorzystywane wymaganie roli:

```
builder.Services.AddAuthorization(options =>  
{  
    options.AddPolicy(  
        "IsAdminJwt",  
        policy =>  
            policy  
                .RequireRole("Admin")  
                .AddAuthenticationSchemes(JwtBearerDefaults.Au  
thenticationScheme)  
    );  
});
```

A w nagłówku akcji `Post()` kontrolera `FoxController` użyj tych reguł bezpieczeństwa zmieniając atrybut `[Authorize]` na:

```
[Authorize("IsAdminJwt")]
```

Rola użytkownika musi być również zapisywana w tokenie, aby było możliwe jej sprawdzenie. Zmodyfikuj akcję `Token()` kontrolera `User` dodając wszystkie role do `claims` tworzonego tokena:

```
foreach (var role in await _user.GetRolesAsync(user))  
{  
    claims.Add(new Claim(ClaimTypes.Role, role));  
}
```

(po utworzeniu listy o nazwie `claims`)

Przetestuj, czy faktycznie dodawać „polubienia” mogą użytkownicy którzy uzyskali token. Nie powinni móc dodawać nowych obiektów żądaniem `POST` do <http://localhost:5010/api/fox>.

Z kolei administratorzy (użytkownicy należący do roli administratora) ze swoim tokenem powinni móc robić obydwie czynności.

