Projektowanie aplikacji ASP.NET

Wykład 07/15
Autentykacja, autoryzacja

Wiktor Zychla 2024/2025

Spis treści

1 Wprowadzenie

Autentykacja = proces rozpoznania tożsamości użytkownika

Autoryzacja = proces decyzyjny w którym użytkownikowi przyznaje się dostęp do zasobów lub zabrania się dostępu do zasobów

W praktyce, upraszczając, można powiedzieć że autentykacja jest *jakoś* związana z logowaniem, natomiast autoryzacja pozwala sterować dostępem do zasobów (np. „brak dostępu dla niezalogowanych” lub „dostęp tylko dla użytkowników w roli administratorzy” itp.)

2 Autentykacja, autoryzacja

Są dwa podstawowe typy autentykacji

- Oparte o status 401, wbudowane w protokół HTTP
- Oparte o status 302, związane z przekierowaniem na dodatkowy zasób ustalający tożsamość użytkownika (np. stronę logowania)
    - 302 do strony w ramach tej samej witryny, z możliwością współdzielenia stanu (np. ciastka)
    - 302 do strony w ramach innej witryny bez możliwości współdzielenia stanu

Od strony kodu, w aplikacji, pomysł na uwierzytelnianie żądania polega na skojarzeniu z żądaniem obiektu typu **IPrinicipal** który reprezentuje użytkownika – właściciela bieżącego żądania. Obiekt ten może reprezentować użytkownika nie mającego odpowiednika w systemie operacyjnym, dzięki czemu możliwe jest tworzenie aplikacji z *wirtualnymi* rejestrami użytkowników (na przykład przechowywanymi w bazie danych).

Taki **IPrincipal** skojarzony z żądaniem jest więc czymś zupełnie innym niż tożsamość użytkownika – właściciela procesu wykonującego kod po stronie serwera.

Pozyskanie tożsamości właściciela procesu:

```
var user = System.Security.Principal.WindowsIdentity.GetCurrent().Name;
```

Pozyskanie wirtualnego użytkownika – właściciela bieżącego żądania

```
var principal = HttpContext.Current.User;
```

W domyślnej aplikacji, w której nie skonfigurowano żadnego uwierzytelniania:

- Właściciel procesu zawsze jest niepusty (każdy proces w systemie ma właściciela) – jest to konto ustawione na serwerze aplikacyjnym jako właściciel procesu
- Właściciel bieżącego żądania jest pusty – żadna logika nie wymaga od użytkowników bycia „zalogowanym”

Q: skąd w potoku przetwarzania, w **HttpContext.Current.User**, bierze się wartość obiektu reprezentującego bieżącego użytkownika?

A: odpowiada za to moduł autentykacji, albo jeden z wbudowanych w środowisko albo własny, napisany przez programistę. Taki moduł w potoku, w funkcji wywoływanej **przed** aktywacją handlera, ustawia wartość reprezentującą użytkownika na podstawie *jakichś* informacji (niżej pokazujemy jakich). Większość modułów używa do tego zdarzenia **AuthenticateRequest** (por. potok przetwarzania omówiony na jednym z poprzednich wykładów)

2.1 Autentykacja 401 (Kerberos, NTLM)

Żargonowa nazwa tej rodziny mechanizmów uwierzytelniania pochodzi od nazwy statusu HTTP 401, który przewiduje obsługę autentykacji na poziomie samego protokołu HTTP: w przypadku stwierdzenia braku informacji o autentykacji, serwer odsyła do przeglądarki status 401, a w zależności od pewnych dodatkowych nagłówków, przeglądarka podejmuje dalsze kroki (na przykład loguje użytkownika automatycznie lub pokazuje wbudowane okno logowania).

Autentykacja zintegrowana wymaga przełączenia trybu autentykacji w pliku konfiguracyjnym na **Windows**

```
<system.web>
   <authentication mode="Windows" />
   <authorization>
     <deny users="?"/>
     <allow users="*"/>
   </authorization>
```

W tym trybie sesja użytkownika nie jest podtrzymywana ciastkami i zależy od jednego z obsługiwanych przez przeglądarki protokołów. W szczególności:

- Dla protokołu NTLM sesja negocjowana jest „per połączenie” (per socket)
- Dla protokołu Kerberos sesja podtrzymywana jest w nagłówku **Authorization**

Zalety:

- Uwierzytelnianie zintegrowane skonfigurowane w sieci lokalnej **nie wymaga** od użytkowników wprowadzania loginu i hasła - przeglądarka sama podejmuje dialog z serwerem zgodnie z protokołami NTLM lub Kerberos i użytkownik jest automatycznie logowany swoim kontem domenowym. Standardy te są obsługiwane przez współczesne przeglądarki. Dla wdrożeń aplikacji intranetowych w sieciach korporacyjnych z kontrolerem domeny opartym o Active Directory jest to bardzo dobry wybór.

Wady:

- Ten model uwierzytelniania nie przenosi się ładnie na aplikacje w sieci globalnej – w sieci globalnej protokoły NTLM/Kerberos będą wymagać podania hasła użytkownika, a prompt (okno do wpisania loginu/hasła) jest poza kontrolą programisty (przeglądarka ma wbudowane okno promptu – i wygląda ono dość nieszczególnie)
- Samodzielne oprogramowanie protokołów NTLM/Kerberos (przy braku automatycznego wsparcia ze strony kontrolera domeny w sieci bez kontrolera domeny) jest zadaniem bardzo trudnym (raczej się tego nie praktykuje)
- W tym modelu w zasadzie nie funkcjonuje „wylogowanie”

2.2 Autentykacja 302

Żargonowa nazwa tej rodziny mechanizmów uwierzytelnienia pochodzi od nazwy statusu HTTP 302, czyli dobrze już nam znanego przekierowania: w przypadku stwierdzenia braku informacji o autentykacji, serwer przekierowuje użytkownika na podstronę (podstrony), na których użytkownik *jakoś* potwierdza tożsamość, a serwer zapisuje tę informację tak, żeby była dostępna w kolejnych żądaniach (najczęściej: ciastko, rzadzej: sesja).

Najprostsza poprawnie zrealizowana autentykacja typu „przekierowanie 302” opiera się podtrzymywanie stanu sesji zalogowanego użytkownika w **ciastku.**

Za wydawanie i kontrolę ciastka odpowiada **moduł autentykacji**. Najprostszy moduł autentykacji, dostarczony razem z .NET.Framework to wbudowany moduł **FormsAuthenticationModule**. Ciastka powinny być szyfrowane/podpisane, aby uniemożliwić sfałszowanie ich zawartości po stronie przeglądarki.

Konfiguracja modułu Forms w potoku przetwarzania wymaga wyłącznie sekcji **authentication** w pliku konfiguracyjnym.

Dla podsystemów w których żądania mapują się na pliki (**WebForms**), wymagane są reguły autoryzacji aby moduł mógł wymusić przekierowanie nieautoryzowanego żądania do strony logowania.

```
  <system.web>
    <authentication mode="Forms">
      <forms loginUrl="LoginPage.aspx" defaultUrl="WebForm1.aspx" />
    </authentication>
    <authorization>
      <deny users="?"/>
      <allow users="*"/>
    </authorization>
```

Dla podsystemów w których żądania są obsługiwane dynamicznie (**MVC**) stosuje się znany już nam mechanizm **filtrów**, konkretnie – wbudowany filtr **Authorize**

```
public class HomeController : Controller
{
    [Authorize]
    public IActionResult Index()
    {
        return View();
    }
}
```

Moduł **FormsAuthentication** pozwala na

- Automatyczne wydanie ciastka zalogowanemu użytkownikowi i przekierowanie na stronę wskazywaną na stronie logowania przez parametr **ReturnUrl**

```
string username;
FormsAuthentication.RedirectFromLoginPage(username, false);
```

- Pełną kontrolę nad ważnością sesji zapisanej w ciastku autentykacji

```
FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(
    1, username,
    DateTime.Now, DateTime.Now.AddMinutes(20),
    false, string.Empty);
```

```
HttpCookie cookie = new HttpCookie(FormsAuthentication.FormsCookieName);
cookie.Value = FormsAuthentication.Encrypt(ticket);

HttpContext.Current.Response.AppendCookie(cookie);
```

- Wylogowanie (**FormsAuthentication::SignOut**)

3 Architektura warstwy autentykacji

W celu oddzielenia warstwy aplikacji odpowiadającej za techniczną stronę autentykacji (302/401) od samego procesu uwierzytelniania, zaproponowano w .NET.Framework tzw. model dostawców (Provider Model), w którym elementy aplikacji odpowiedzialne za

- Uwierzytelnianie
- Autoryzację
- Ale też - zarządzanie sesją, itd.

mają swoje abstrakcje (klasy abstrakcyjne) a zadaniem programisty jest dostarczenie implementacji, np.:

```
public class MyMembershipProvider : MembershipProvider
{
    public MyMembershipProvider()
    {
    }

    …


    public override bool ValidateUser(string username, string password)
    {
        if (username == password)
            return true;

        return false;
    }
}
```

dla autentykacji oraz

```
public class MyRoleProvider : RoleProvider
{
    public MyRoleProvider()
    {
    }

    …

    public override string[] GetRolesForUser(string username)
    {
        return new string[] { username };
    }
}
```

Rejestracja dostawców wymaga wyłącznie odpowiedniej konfiguracji:

```
    <membership defaultProvider ="MyMembershipProvider">
      <providers>
        <add name="MyMembershipProvider" type="MyMembershipProvider"/>
      </providers>
    </membership>
    <roleManager enabled ="true" defaultProvider="MyRoleProvider">
      <providers>
```

```
        <add name="MyRoleProvider" type="MyRoleProvider"/>
      </providers>
    </roleManager>
```

i od tego momentu możliwe jest posługiwanie się w aplikacji fasadami odpowiadającymi dostawcom, np.: **Membership**, czy **Roles.**

Autoryzacja w modelu RBS (Role-based Security) możliwa jest na trzy sposoby:

1. Zapamiętanie w ciastku Forms wyłącznie nazwy użytkownika, odczytywanie ról za każdym żądaniem
2. Zapamiętanie w ciastku Forms ról w sekcji UserData
3. Zapamiętanie ról w dodatkowym ciastku (wymaga ustawienia wartości **true** atrybutu **cacheRolesInCookie** węzła **roleManager**

Model dostawców to niejedyny pomysł organizacji kodu odpowiedzialnego za autentykację i autoryzację. Jego następcą jest podsystem ASP.NET Identity. Wadą tego typu podejść jest ich coraz wyższa złożoność – ASP.NET Identity niemal „z pudełka” funkcjonuje jako samodzielny podsystem przechowujący użytkowników i role, ale struktura tabel bazodanowych w której trzymane są dane jest narzucona. Jej modyfikacja jest możliwa, ale wymaga niemałych nakładów.

4 Alternatywny moduł obsługi ciastka autentykacji dla .NET.Framework

Moduł **Forms Authentication** można zastąpić modułem **Session Authentication Module**. Wskazówki implementacyjne oraz uzasadnienie dlaczego warto można znaleźć w artykule Forms Authentication Revisited.

W tym przykładzie najistotniejszy jest fragment tworzenia obiektu tożsamości:

```
var identity = new ClaimsIdentity("custom");
identity.AddClaim(new Claim(ClaimTypes.Name, "login1"));

var principal = new ClaimsPrincipal(identity);

SessionAuthenticationModule sam = FederatedAuthentication.SessionAuthenticationMod
ule;
var token =
    sam.CreateSessionSecurityToken(principal, string.Empty,
         DateTime.Now.ToUniversalTime(), DateTime.Now.AddMinutes(20).ToUniversalTi
me(), false);

sam.WriteSessionTokenToCookie(token);
```

Obiekt tożsamości ma typ **ClaimsPrincipal.** Proszę zapamiętać ten typ, to najbardziej ogólny i współcześnie najczęściej używany typ dla interfejsu **IPrincipal**. Kiedy będziemy mówili o ASP.NET w wersji 5, 6 i wyższych – będziemy mieli z nim do czynienia.

Garść szczegółów technicznych:

- Jak sama nazwa wskazuje, **ClaimsPrincipal** to obiekt reprezentujący użytkownika, oparty o tzw. **claims**. Claim (oświadczenie) to para (typ, wartość). Typem może być imię, nazwisko, email, itd.
- Obiekt ClaimsPrincipal może mieć dowolną liczbę oświadczeń, moduł **SessionAuthenticationModule** potrafi nawet podzielić zbyt duże ciastko na wiele mniejszych. Ograniczeniem jest więc maksymalna liczba ciastek jakie można wydać.
- W scenariuszu federacyjnym (będziemy mówić o tym za chwilę) gdzie tożsamość użytkownika pochodzi z zewnętrznego serwisu (np. Google), część oświadczeń (imię, nazwisko, role) może pochodzić z lokalnej bazy
- Istnieje enumeracja **ClaimTypes**, która zawiera wypis najczęściej używanych typów oświadczeń (ale bez problemu można tworzyć nowe, własne)
- Specjalnie traktowane są oświadczenia typu **Role** (ClaimTypes.Role) – te są używane w implementacji metody **IsInRole**(…) – jeżeli więc chcielibyśmy użyć mechanizmu autoryzacji i powiedzieć że użytkownik jest administratorem, mielibyśmy

```
var identity = new ClaimsIdentity("custom");
identity.AddClaim(new Claim(ClaimTypes.Name, "login1"));
identity.AddClaim(new Claim(ClaimTypes.Role, "ADMIN"));
```

5 Autentykacja / autoryzacja w .NET.Core

Middleware do obsługi autoryzacji wymaga aktywacji (**UseAuthorization**) i obsługiwane jest w zależności od podsystemu. W MVC – standardowo, atrybutami [**Authorize**].

Autentykacja wymaga aktywacji (**UseAuthentication**) oraz konfiguracji usług:

```
services
    .AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(CookieAuthenticationDefaults.AuthenticationScheme, options =>
    {
        options.LoginPath = "/Account/Logon";
        options.SlidingExpiration = true;
    });
```

Wszystko co wiemy o tworzeniu tożsamości w obiekcie **ClaimsPrincipal** tu działa tak samo. Zmienia się wyłącznie API do wydania ciastka – jest to metoda **SignInAsync**, która w połączeniu z zarejestrowanymi middleware (**AddAuthentication** i **AddCookie**) powoduje właśnie zapisanie ciasteczka. Do usuwania ciasteczka służy **SignOutAsync**.

```
public class AccountController : Controller
{
    [HttpGet]
    public IActionResult Logon()
    {
        var model = new AccountLogonModel();
        return View(model);
    }

    public async Task<IActionResult> Logon(AccountLogonModel model)
    {
        if ( this.ModelState.IsValid )
        {
            if ( !string.IsNullOrEmpty( model.UserName ) && model.UserName == mo
del.Password )
            {
                List<Claim> claims = new List<Claim>
                {
                    new Claim(ClaimTypes.Name, model.UserName)
                };

                // create identity
                ClaimsIdentity identity   = new ClaimsIdentity(claims, CookieAut
henticationDefaults.AuthenticationScheme);
                ClaimsPrincipal principal = new ClaimsPrincipal(identity);

                await this.HttpContext.SignInAsync(CookieAuthenticationDefaults.
AuthenticationScheme, principal);

                return Redirect("/");
            }
            else
            {
```

```
                this.ViewBag.Message = "Zła nazwa użytkownika lub hasło";
                return View(model);
            }
        }
        else
        {
            return View(model);
        }
    }
}
```