



Распределенные информационно-аналитические системы

Практическое занятие № 16. «Авторизация»

Профессор кафедры КБ-2: д.т.н. Шатовкин Р.Р.

Учебные вопросы:

1. Введение в авторизацию.
2. Авторизация с помощью JWT-токенов в клиенте JavaScript.
3. Авторизация по ролям.
4. Авторизация на основе Claims.
5. Создание ограничений для авторизации.

1. Введение в авторизацию

Важное место в приложении занимает и авторизация. Авторизация представляет процесс определения, имеет ли пользователь право доступа к некоторому ресурсу. Авторизация отвечает на вопрос "Какие права пользователь имеет в системе?".

ASP.NET Core имеет встроенную поддержку авторизации.

Авторизация

Авторизация представляет процесс определения прав пользователя в системе, к каким ресурсам приложения он имеет право доступа и при каких условиях.

Хотя авторизация представляет отдельный независимый процесс, тем не менее для нее также необходимо, чтобы приложение также применяло аутентификацию.

Для подключения авторизации необходимо встроить компонент **Microsoft.AspNetCore.Authorization.AuthorizationMiddleware**. Для этого применяется встроенный метод расширения **UseAuthorization()**

```
1 public static IApplicationBuilder UseAuthorization(this IApplicationBuilder app)
```

Кроме того, для применения авторизации необходимо зарегистрировать сервисы авторизации с помощью метода **AddAuthorization()**:

```
1 public static IServiceCollection AddAuthorization(this IServiceCollection services)
2 public static IServiceCollection AddAuthorization(this IServiceCollection services, Action<AuthorizationOptions> configure)
```

Вторая версия метода принимает делегат, который с помощью параметра **AuthorizationOptions** позволяет сконфигурировать авторизацию.

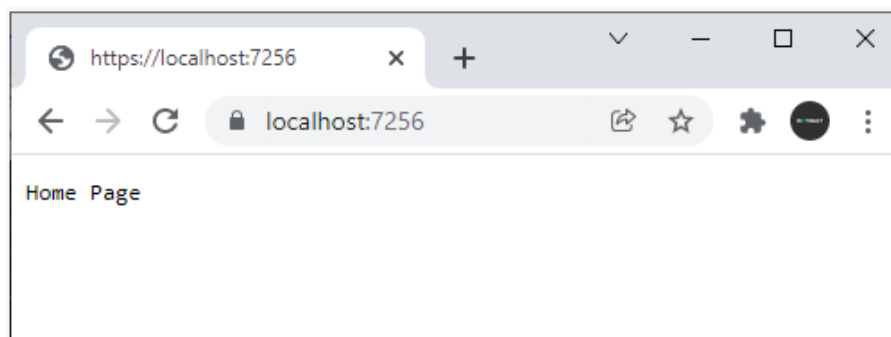
Ключевым элементом механизма авторизации в **ASP.NET Core** является атрибут **AuthorizeAttribute** из пространства имен **Microsoft.AspNetCore.Authorization**, который позволяет ограничить доступ к ресурсам приложения. Например:

```
1 using Microsoft.AspNetCore.Authorization;
2
3 var builder = WebApplication.CreateBuilder();
4
5 builder.Services.AddAuthentication("Bearer") // добавление сервисов аутентификации
6     .AddJwtBearer(); // подключение аутентификации с помощью jwt-токенов
7 builder.Services.AddAuthorization(); // добавление сервисов авторизации
8
9 var app = builder.Build();
10
11 app.UseAuthentication(); // добавление middleware аутентификации
12 app.UseAuthorization(); // добавление middleware авторизации
13
14 app.Map("/hello", [Authorize]() => "Hello World!");
15 app.Map("/", () => "Home Page");
16
17 app.Run();
```

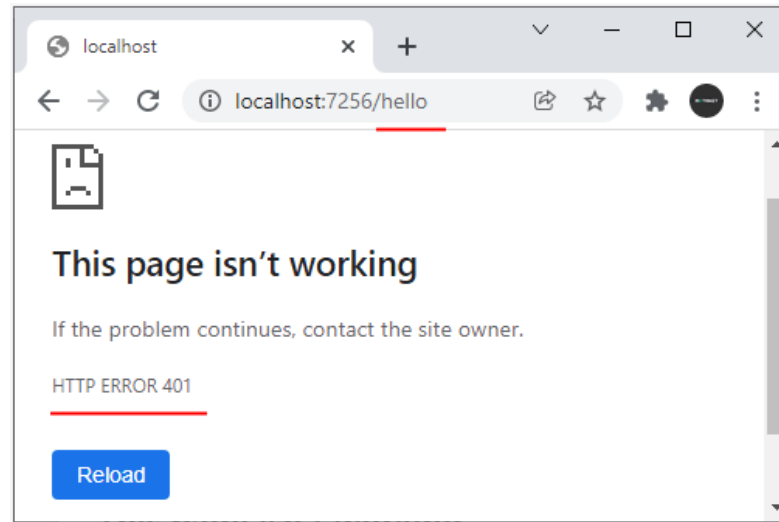
Здесь в приложении определены две конечных точки: "/" и "/hello". При этом конечная точка "/hello" применяет атрибут **Authorize**. Атрибут указывается перед обработчиком конечной точки.

Применение данного атрибута означает, что к конечной точке "/hello" имеют доступ только аутентифицированные пользователи.

Если мы обратимся к конечной точке "/", то у нас не возникнет никаких проблем:



Однако если мы обратимся к ресурсу "/hello", то мы получим ошибку **401**, которая говорит о том, что пользователь не авторизован для доступа к этому ресурсу:



2. Авторизация с помощью JWT-токенов в клиенте JavaScript

В прошлом учебном вопросе был рассмотрен процесс конфигурации и генерации **JWT**-токенов. Теперь посмотрим, как мы можем применить **JWT**-токен для авторизации в приложении. Для этого определим в файле **Program.cs** следующий код:

```
1 using Microsoft.AspNetCore.Authentication.JwtBearer;
2 using Microsoft.AspNetCore.Authorization;
3 using Microsoft.IdentityModel.Tokens;
4 using System.IdentityModel.Tokens.Jwt;
5 using System.Security.Claims;
6 using System.Text;
7
8 // условная бд с пользователями
9 var people = new List<Person>
10 {
11     new Person("tom@gmail.com", "12345"),
12     new Person("bob@gmail.com", "55555")
13 };
14
15 var builder = WebApplication.CreateBuilder();
16
17 builder.Services.AddAuthorization();
18 builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
19     .AddJwtBearer(options =>
20     {
21         options.TokenValidationParameters = new TokenValidationParameters
22         {
23             ValidateIssuer = true,
24             ValidIssuer = AuthOptions.ISSUER,
25             ValidateAudience = true,
26             ValidAudience = AuthOptions.AUDIENCE,
27             ValidateLifetime = true,
28             IssuerSigningKey = AuthOptions.GetSymmetricSecurityKey(),
29             ValidateIssuerSigningKey = true
30         };
31     });
32 var app = builder.Build();
33
34 app.UseDefaultFiles();
35 app.UseStaticFiles();
36
37 app.UseAuthentication();
38 app.UseAuthorization();
39
```

```

40 app.MapPost("/login", (Person loginData) =>
41 {
42     // находим пользователя
43     Person? person = people.FirstOrDefault(p => p.Email == loginData.Email && p.Password == loginData.Password);
44     // если пользователь не найден, отправляем статусный код 401
45     if(person is null) return Results.Unauthorized();
46
47     var claims = new List<Claim> {new Claim(ClaimTypes.Name, person.Email) };
48     // создаем JWT-токен
49     var jwt = new JwtSecurityToken(
50         issuer: AuthOptions.ISSUER,
51         audience: AuthOptions.AUDIENCE,
52         claims: claims,
53         expires: DateTime.UtcNow.Add(TimeSpan.FromMinutes(2)),
54         signingCredentials: new SigningCredentials(AuthOptions.GetSymmetricSecurityKey(), SecurityAlgorithms.HmacSha256));
55     var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);
56
57     // формируем ответ
58     var response = new
59     {
60         access_token = encodedJwt,
61         username = person.Email
62     };
63
64     return Results.Json(response);
65 });
66 app.Map("/data", [Authorize] () => new { message= "Hello World!" });
67
68 app.Run();
69
70 public class AuthOptions
71 {
72     public const string ISSUER = "MyAuthServer"; // издатель токена
73     public const string AUDIENCE = "MyAuthClient"; // потребитель токена
74     const string KEY = "mysupersecret_secretkey!123"; // ключ для шифрации
75     public static SymmetricSecurityKey GetSymmetricSecurityKey() =>
76         new SymmetricSecurityKey(Encoding.UTF8.GetBytes(KEY));
77 }
78
79 record class Person(string Email, string Password);

```

Для представления пользователя в приложении здесь определен **record**-класс **Person**, который имеет два свойства: **email** и **password**. И для упрощения ситуации вместо базы данных все пользователи приложения хранятся в списке **people**. Условно говоря, у нас есть два пользователя.

Для описания некоторых настроек генерации токена, как и в прошлой теме, в коде определен специальный класс **AuthOptions**, и также, как и в прошлой теме, с помощью метода **AddJwtBearer()** в приложение добавляется конфигурация токена.

В конечной точке **"/login"**, которая обрабатывает **POST**-запросы, получаем отправленные клиентом аутентификационные данные опять же для простоты в виде объекта **Person**:

```
1 app.MapPost("/login", (Person loginData) =>
```

Используя полученные данные, пытаемся найти в списке **people** пользователя:

```
1 Person? person = people.FirstOrDefault(p => p.Email == loginData.Email && p.Password == loginData.Password);
```

Если пользователь не найден, то есть переданы некорректные **email** и/или **password**, то отправляем статусный код **401**, который говорит о том, что доступ запрещен:

```
1 if(person is null) return Results.Unauthorized();
```

Если пользователь найден, то создается список объектов **Claim** с одним **Claim**, который представляет **email** пользователя. Генерируем **jwt**-токен:

```
1 var claims = new List<Claim> {new Claim(ClaimTypes.Name, person.Email) };
2 var jwt = new JwtSecurityToken(
3     issuer: AuthOptions.ISSUER,
4     audience: AuthOptions.AUDIENCE,
5     claims: claims,
6     expires: DateTime.UtcNow.Add(TimeSpan.FromMinutes(2)), // действие токена истекает через 2 минуты
7     signingCredentials: new SigningCredentials(AuthOptions.GetSymmetricSecurityKey(), SecurityAlgorithms.HmacSha256));
8 var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);
```


Далее формирует ответ клиенту. Он отправляется в виде объекта в формате **json**, который содержит два свойства: **access_token** – собственно токен и **username – email** аутентифицированного пользователя

```
var response = new { access_token = encodedJwt, username = person.Email }; return Results.Json(response);
```

Еще одна конечная точка – **"/data"** использует атрибут **Authorize**, поэтому для обращения к ней необходимо в запросе отправлять полученный **jwt**-токен.

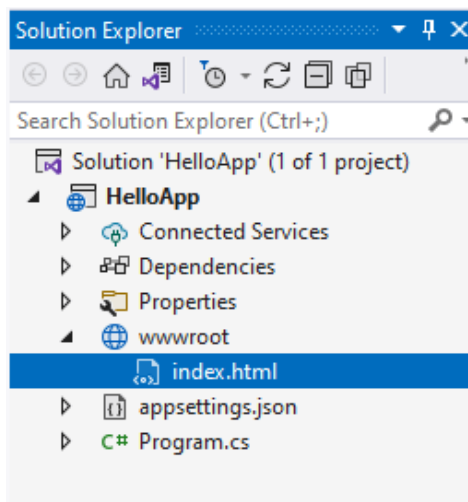
```
1 app.Map("/data", [Authorize] (HttpContext context) => $"Hello World!");
```

Создание клиента на javascript

Теперь определим клиент для тестирования авторизации с помощью токена. Итак, в коде приложения определено подключение статических файлов по умолчанию:

```
1 app.UseDefaultFiles();  
2 app.UseStaticFiles();
```

В качестве веб-страницы по умолчанию добавим в проект для статических файлов папку **wwwroot**, а в нее – новый файл **index.html**:



В файле **index.html** определим следующий код:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <title>METANIT.COM</title>
6 </head>
7 <body>
8   <div id="userInfo" style="display:none;">
9     <p>Добро пожаловать <span id="userName"></span>!</p>
10    <input type="button" value="Выйти" id="logOut" />
11  </div>
12  <div id="loginForm">
13    <h3>Вход на сайт</h3>
14    <p>
15      <label>Введите email</label><br />
16      <input type="email" id="email" />
17    </p>
18    <p>
19      <label>Введите пароль</label><br />
20      <input type="password" id="password" />
21    </p>
22    <input type="submit" id="submitLogin" value="Логин" />
23  </div>
24  <p>
25    <input type="submit" id="getData" value="Получить данные" />
26  </p>
27  <script>
28    var tokenKey = "accessToken";
29    // при нажатии на кнопку отправки формы идет запрос к /login для получения токена
30    document.getElementById("submitLogin").addEventListener("click", async e => {
31      e.preventDefault();
32      // отправляет запрос и получаем ответ
33      const response = await fetch("/login", {
34        method: "POST",
35        headers: { "Accept": "application/json", "Content-Type": "application/json" },
36        body: JSON.stringify({
37          email: document.getElementById("email").value,
38          password: document.getElementById("password").value
39        })
40      });
41      // если запрос прошел нормально
42      if (response.ok === true) {
43        // получаем данные
44        const data = await response.json();
```

```

45         // изменяем содержимое и видимость блоков на странице
46         document.getElementById("userName").innerText = data.username;
47         document.getElementById("userInfo").style.display = "block";
48         document.getElementById("loginForm").style.display = "none";
49         // сохраняем в хранилище sessionStorage токен доступа
50         sessionStorage.setItem(tokenKey, data.access_token);
51     }
52     else // если произошла ошибка, получаем код статуса
53         console.log("Status: ", response.status);
54 });
55
56 // кнопка для обращения по пути "/data" для получения данных
57 document.getElementById("getData").addEventListener("click", async e => {
58     e.preventDefault();
59     // получаем токен из sessionStorage
60     const token = sessionStorage.getItem(tokenKey);
61     // отправляем запрос к "/data"
62     const response = await fetch("/data", {
63         method: "GET",
64         headers: {
65             "Accept": "application/json",
66             "Authorization": "Bearer " + token // передача токена в заголовке
67         }
68     });
69
70     if (response.ok === true) {
71         const data = await response.json();
72         alert(data.message);
73     }
74     else
75         console.log("Status: ", response.status);
76 });
77
78 // условный выход - просто удаляем токен и меняем видимость блоков
79 document.getElementById("logOut").addEventListener("click", e => {
80
81     e.preventDefault();
82     document.getElementById("userName").innerText = "";
83     document.getElementById("userInfo").style.display = "none";
84     document.getElementById("loginForm").style.display = "block";
85     sessionStorage.removeItem(tokenKey);
86 });
87 </script>
88 </body>
89 </html>

```

Первый блок на странице выводит информацию о вошедшем пользователе и ссылку для выхода. Второй блок содержит форму для логина.

После нажатия кнопки на форме логина запрос будет отправляться методом **POST** на адрес **"/login"**. Конечная точка, которая отвечает за обработку **POST**-запросов по этому маршруту, если переданы корректные **email** и пароль, отправит в ответ токен.

Ответом сервера в случае удачной аутентификации будет примерно следующий объект:

```
1 {  
2   access_token : "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzbnFwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjoicXdlcnR5IiwiaHR0cDovL3NjaGVtYXMubWljcm9zb2Z0LmNvbS93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9yb2x1IjoidXNlciIsIm5iZiI6MTQ4MTYzOTMxMSwiZXhwIjoxNDgxNjM5MzcxLCJpc3MiOiJNeUF1dGhTZXJ2ZXIiLCJhdWQiOiJodHRwOi8vbG9jYXRob3N0LjUxODg0LyJ9.dQJF6pALUZW3wGBANy_tCwk5_NR0TVBwgnxRbb1p5Ho",  
3   cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjoicXdlcnR5IiwiaHR0cDovL3NjaGVtYXMubWljcm9zb2Z0LmNvbS93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9yb2x1IjoidXNlciIsIm5iZiI6MTQ4MTYzOTMxMSwiZXhwIjoxNDgxNjM5MzcxLCJpc3MiOiJNeUF1dGhTZXJ2ZXIiLCJhdWQiOiJodHRwOi8vbG9jYXRob3N0LjUxODg0LyJ9.dQJF6pALUZW3wGBANy_tCwk5_NR0TVBwgnxRbb1p5Ho",  
4   Wljcm9zb2Z0LmNvbS93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9yb2x1IjoidXNlciIsIm5iZiI6MTQ4MTYzOTMxMSwiZXhwIjoxNDgxNjM5MzcxLCJpc3MiOiJNeUF1dGhTZXJ2ZXIiLCJhdWQiOiJodHRwOi8vbG9jYXRob3N0LjUxODg0LyJ9.dQJF6pALUZW3wGBANy_tCwk5_NR0TVBwgnxRbb1p5Ho",  
5   I6MTQ4MTYzOTMxMSwiZXhwIjoxNDgxNjM5MzcxLCJpc3MiOiJNeUF1dGhTZXJ2ZXIiLCJhdWQiOiJodHRwOi8vbG9jYXRob3N0LjUxODg0LyJ9.dQJF6pALUZW3wGBANy_tCwk5_NR0TVBwgnxRbb1p5Ho",  
6   odHRwOi8vbG9jYXRob3N0LjUxODg0LyJ9.dQJF6pALUZW3wGBANy_tCwk5_NR0TVBwgnxRbb1p5Ho",  
7   username: "tom@gmail.com"  
8 }
```

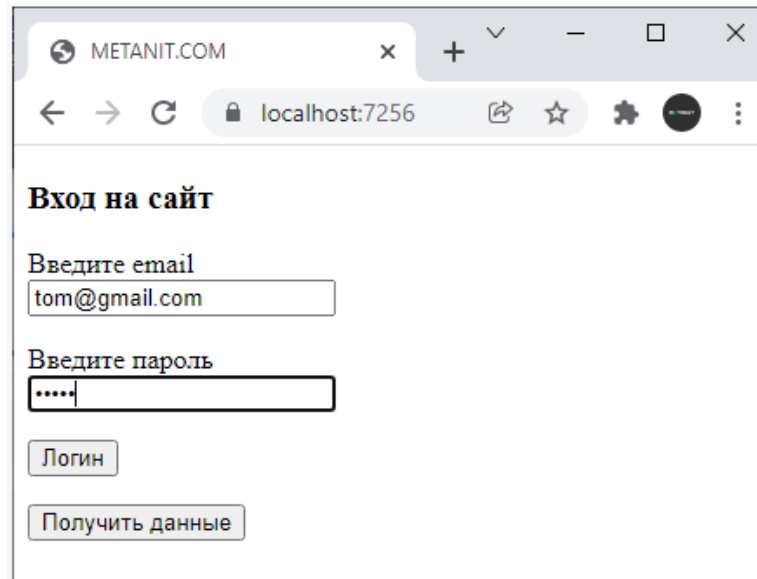
Параметр **access_token** как раз и будет представлять токен доступа. Также в объекте передается дополнительная информация о нике пользователя.

Для того, чтобы в коде **js** данный токен в дальнейшем был доступен, то он сохраняется в хранилище **sessionStorage**.

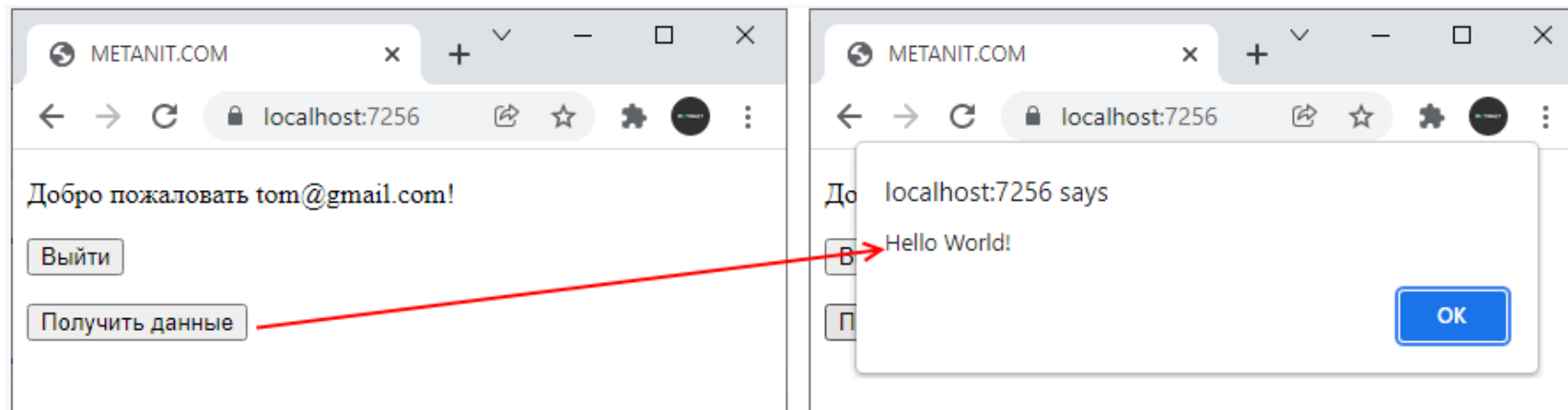
Дополнительная кнопка с **id="getData"** на странице предназначена для тестирования авторизации с помощью токена. По ее нажатию будет выполняться запрос по адресу **"/data"**, для доступа к которому необходимо быть аутентифицированным. Чтобы отправить токен в запросе, нам нужно настроить в запросе заголовок **Authorization**:

```
1 headers: {  
2   "Accept": "application/json",  
3   "Authorization": "Bearer " + token // передача токена в заголовке  
4 }
```

Запустим проект и введем данные одного из пользователя, который есть в списке **people**:



При вводе корректных данных сервер пришлет клиенту объект с **jwt**-токеном и логином пользователя. И после этого мы можем нажать на кнопку **"Получить данные"** и тем самым обратиться к ресурсу **"/data"**, для доступа к которому требуется токен



В то же время если мы попробуем обратиться к этому же ресурсу без токена или с токеном с истекшим сроком, то получим ошибку **401 (Unauthorized)**.

3. Авторизация по ролям

Авторизация по ролям позволяет разграничить доступ к ресурсам приложения в зависимости от роли, к которой принадлежит пользователь.

Допустим, у нас есть следующие классы, которые описывают пользователя и его роль:

```
1 class Person
2 {
3     public string Email { get; set; }
4     public string Password { get; set; }
5     public Role Role { get; set; }
6     public Person(string email, string password, Role role)
7     {
8         Email = email;
9         Password = password;
10        Role = role;
11    }
12 }
13 class Role
14 {
15     public string Name { get; set; }
16     public Role(string name) => Name = name;
17 }
```

Класс роли содержит свойство **Name**, которое хранит название роли. А класс **Person** хранит **email**-адрес, пароль и роль пользователя.

В файле **Program.cs** определим следующий код:

```
1 using Microsoft.AspNetCore.Authentication.Cookies;
2 using System.Security.Claims;
3 using Microsoft.AspNetCore.Authentication;
4 using Microsoft.AspNetCore.Authorization;
5
6 var adminRole = new Role("admin");
7 var userRole = new Role("user");
8 var people = new List<Person>
9 {
10     new Person("tom@gmail.com", "12345", adminRole),
11     new Person("bob@gmail.com", "55555", userRole),
12 };
13
14 var builder = WebApplication.CreateBuilder();
15 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
16     .AddCookie(options =>
17     {
18         options.LoginPath = "/login";
19         options.AccessDeniedPath = "/accessdenied";
20     });
21 builder.Services.AddAuthorization();
22
23 var app = builder.Build();
24
25 app.UseAuthentication();
26 app.UseAuthorization(); // добавление middleware авторизации
27
28 app.MapGet("/accessdenied", async (HttpContext context) =>
29 {
30     context.Response.StatusCode = 403;
31     await context.Response.WriteAsync("Access Denied");
32 });
```

```
33 app.MapGet("/login", async (HttpContext context) =>
34 {
35     context.Response.ContentType = "text/html; charset=utf-8";
36     // html-форма для ввода логина/пароля
37     string loginForm = @"<!DOCTYPE html>
38     <html>
39     <head>
40         <meta charset='utf-8' />
41         <title>METANIT.COM</title>
42     </head>
43     <body>
44         <h2>Login Form</h2>
45         <form method='post'>
46             <p>
47                 <label>Email</label><br />
48                 <input name='email' />
49             </p>
50             <p>
51                 <label>Password</label><br />
52                 <input type='password' name='password' />
53             </p>
54             <input type='submit' value='Login' />
55         </form>
56     </body>
57     </html>";
58     await context.Response.WriteAsync(loginForm);
59 });
60
61 app.MapPost("/login", async (string? returnUrl, HttpContext context) =>
62 {
63     // получаем из формы email и пароль
64     var form = context.Request.Form;
65     // если email и/или пароль не установлены, посылаем статусный код ошибки 400
66     if (!form.ContainsKey("email") || !form.ContainsKey("password"))
67         return Results.BadRequest("Email и/или пароль не установлены");
68     string email = form["email"];
69     string password = form["password"];
70 }
```



```

71 // находим пользователя
72 Person? person = people.FirstOrDefault(p => p.Email == email && p.Password == password);
73 // если пользователь не найден, отправляем статусный код 401
74 if (person is null) return Results.Unauthorized();
75 var claims = new List<Claim>
76 {
77     new Claim(ClaimsIdentity.DefaultNameClaimType, person.Email),
78     new Claim(ClaimsIdentity.DefaultRoleClaimType, person.Role.Name)
79 };
80 var claimsIdentity = new ClaimsIdentity(claims, "Cookies");
81 var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
82 await context.SignInAsync(claimsPrincipal);
83 return Results.Redirect(returnUrl ?? "/");
84 });
85 // доступ только для роли admin
86 app.Map("/admin", [Authorize(Roles = "admin")]() => "Admin Panel");
87
88 // доступ только для ролей admin и user
89 app.Map("/", [Authorize(Roles = "admin, user")](HttpContext context) =>
90 {
91     var login = context.User.FindFirst(ClaimsIdentity.DefaultNameClaimType);
92     var role = context.User.FindFirst(ClaimsIdentity.DefaultRoleClaimType);
93     return $"Name: {login?.Value}\nRole: {role?.Value}";
94 });
95 app.MapGet("/logout", async (HttpContext context) =>
96 {
97     await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
98     return "Данные удалены";
99 });
100
101 app.Run();

```

Для упрощения ситуации здесь данные ролей и пользователей определены напрямую в коде в виде списка **people**:

```

1 var adminRole = new Role("admin");
2 var userRole = new Role("user");
3 var people = new List<Person>
4 {
5     new Person("tom@gmail.com", "12345", adminRole),
6     new Person("bob@gmail.com", "55555", userRole),
7 };

```

Здесь определены две роли – **"admin"** и **"user"** и два пользователя.

Для аутентификации здесь используются куки:

```
1 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
2     .AddCookie(options =>
3     {
4         options.LoginPath = "/login";
5         options.AccessDeniedPath = "/accessdenied";
6     });
```

Здесь свойство **options.AccessDeniedPath** указывает на путь, на который будет перенаправляться аутентифицированный пользователь при обращении к ресурсу, для доступа к которому у него нет прав. То есть важно понимать разницу между назначением свойств **options.LoginPath** и **options.AccessDeniedPath**:

options.LoginPath: определяет путь перенаправления для не аутентифицированного пользователя.

options.AccessDeniedPath: определяет путь перенаправления для аутентифицированного пользователя, который не имеет прав для доступа к ресурсу.

В реальности для обоих параметров можно использовать один и тот же путь. Но в данном случае я их разграничил.

Таким образом, при доступе к ресурсу, для которого у пользователя нет прав, пользователь перенаправляется по адресу **"/accessdenied"**. Запрос по этому пути обрабатывается следующей конечной точкой:

```
1 app.MapGet("/accessdenied", async (HttpContext context) =>
2 {
3     context.Response.StatusCode = 403;
4     await context.Response.WriteAsync("Access Denied");
5 });
```

Здесь просто отправляется сообщение о запрете доступа со статусным кодом **403**.

Если пользователь не аутентифицирован, то его перенаправляет по пути **"/login"**. **GET**-запрос по этому пути обрабатывается конечной точкой **app.MapGet("/login")**, которая отправляет пользователю форму для ввода логина и пароля.

Отправленные пользователем в **POST**-запросе данные логина и пароля будут обрабатываться конечной точкой `app.MapPost("/login")`. Ключевым моментом ее обработчика является установка списка объектов **claim**, в которых сохраняется логин пользователя и его роль:

```
1 var claims = new List<Claim>
2 {
3     new Claim(ClaimsIdentity.DefaultNameClaimType, person.Email),
4     new Claim(ClaimsIdentity.DefaultRoleClaimType, person.Role.Name)
5 };
```

Для указания роли здесь применяется тип **claim ClaimsIdentity.DefaultRoleClaimType**, а в качестве значения для этого типа используется имя роли. По сути, больше для установки роли для пользователя ничего не нужно.

Чтобы на уровне отдельных ресурсов приложения разграничить доступ в зависимости от роли, свойству **Roles** атрибута **Authorize** передается набор допустимых ролей:

```
1 [Authorize(Roles = "admin")]
```

Можно передавать несколько ролей через запятую:

```
1 [Authorize(Roles = "admin, user")]
```

Для определения роли текущего пользователя инфраструктура **ASP.NET Core** будет использовать значения **claim** с типом **ClaimsIdentity.DefaultRoleClaimType**.

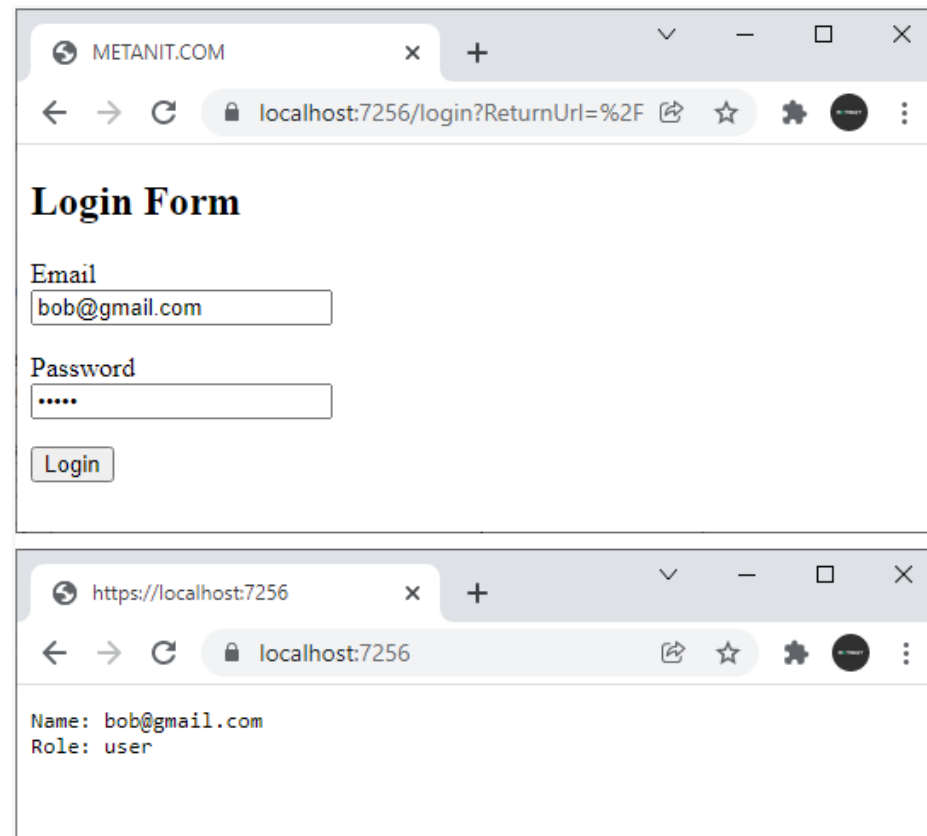
Например, обращаться по пути `"/admin"` могут только пользователи, которые принадлежат роли **"admin"**:

```
1 app.Map("/admin", [Authorize(Roles = "admin")]() => "Admin Panel");
```

В то время как по пути `"/"` могут обращаться представители ролей **"user"** и **"admin"**:

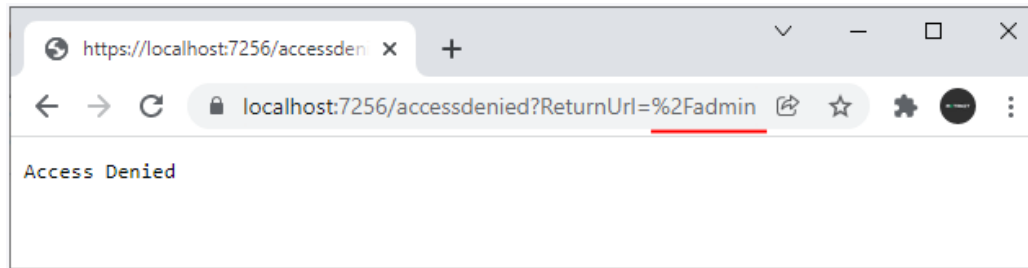
```
1 app.Map("/", [Authorize(Roles = "admin, user")](HttpContext context) =>
2 {
3     var login = context.User.FindFirst(ClaimsIdentity.DefaultNameClaimType);
4     var role = context.User.FindFirst(ClaimsIdentity.DefaultRoleClaimType);
5     return $"Name: {login?.Value}\nRole: {role?.Value}";
6 });
```

Например, залогинимся в приложение пользователем, который имеет роль **"user"**:



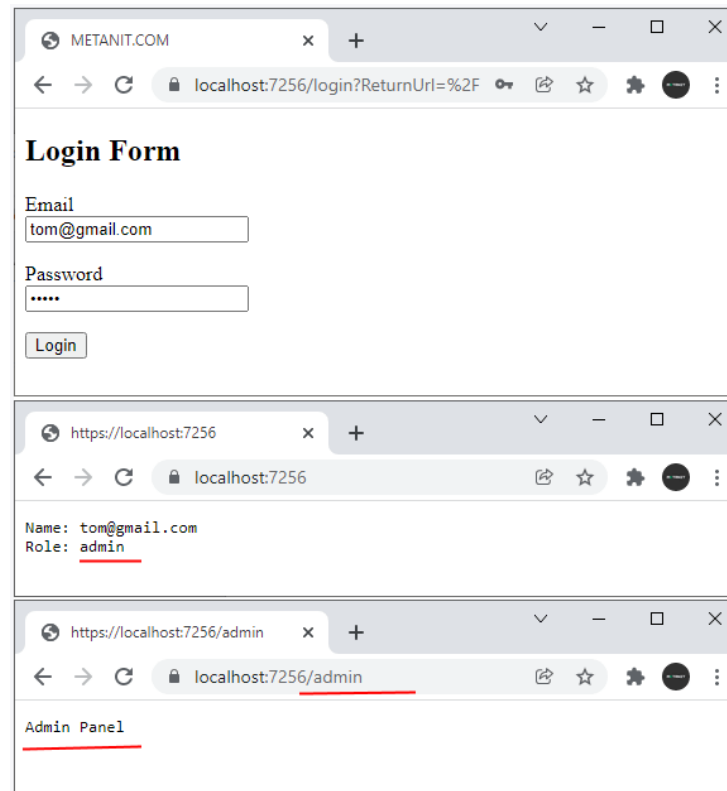
The image shows two browser window screenshots. The top screenshot shows a login form titled "Login Form" on the website METANIT.COM. The form has two input fields: "Email" with the value "bob@gmail.com" and "Password" with masked characters ".....". Below the fields is a "Login" button. The browser's address bar shows "localhost:7256/login?ReturnUrl=%2F". The bottom screenshot shows the result of the login. The browser's address bar shows "https://localhost:7256". The page content displays the user's information: "Name: bob@gmail.com" and "Role: user".

Однако при попытке пользователя с ролью **"user"** обратиться по адресу **"/admin"**, он будет переадресован на адрес **"/accessdenied"**:



Стоит отметить, что при аутентификации куки при перенаправлении по пути из свойства **options.AccessDeniedPath** автоматически передается параметр **ReturnUrl**, из которого можно получить путь, к которому пытался обращаться пользователь.

Теперь выйдем из приложения и снова залогинимся, только теперь под пользователем с ролью **"admin"**:



Таким образом, мы можем разграничивать доступ в приложении в зависимости от роли пользователя.

4. Авторизация на основе Claims

Атрибут **Authorize** легко позволяет разграничить доступ в зависимости от роли, однако для создания авторизации функциональности ролей бывает недостаточно. Например, если мы хотим разграничить доступ на основе возраста пользователя или каких-то других признаков. Для этого применяется авторизация на основе **claims**. Собственно авторизация на основе ролей фактически представляет частный случай авторизации на основе **claims**, так как роль – это тот же объект **Claim**, имеющий тип **ClaimsIdentity.DefaultRoleClaimType**.

Для авторизации на основе **claims** используются политики (**policy**). Политика представляет набор ограничений, которым должен соответствовать пользователь для доступа к ресурсу.

Все применяемые политики добавляются в приложение с помощью метода **builder.Services.AddAuthorization()**. Этот метод устанавливает политики с помощью объекта **AuthorizationOptions**. Например:

```
1 builder.Services.AddAuthorization(opts => {  
2  
3     opts.AddPolicy("OnlyForMicrosoft", policy => {  
4         policy.RequireClaim("company", "Microsoft");  
5     });  
6 });
```

В данном случае добавляется политика с именем **"OnlyForMicrosoft"**. И она требует обязательной установки для текущего пользователя объекта **Claim** с типом **"company"** и значением **"Microsoft"**. Если для пользователя не будет установлено подобного объекта **Claim**, то такой пользователь не будет соответствовать политике.

Для управления политиками в классе **AuthorizationOptions** определены следующие **свойства и методы**:

DefaultPolicy: возвращает политику по умолчанию, которая используется, когда атрибут **Authorize** применяется без параметров.

AddPolicy(name, policyBuilder): добавляет политику.

GetPolicy(name): возвращает политику по имени.

Ключевым методом здесь является **AddPolicy()**. Первый параметр метода представляет название политики, а второй – делегат, который с помощью объекта **AuthorizationPolicyBuilder** позволяет создать политику по определенным условиям. Для создания политики могут применяться следующие **методы класса AuthorizationPolicyBuilder**:

RequireAuthenticatedUser(): пользователь обязательно должен быть аутентифицирован для соответствия политике.

RequireClaim(type): для пользователя должен быть установлен **claim** с типом **type**. Причем не важно, какое значение будет иметь этот **claim**, главное, его наличие.

RequireClaim(type, values): для пользователя должен быть установлен **claim** с типом **type**. Но теперь **claim** должен в качестве значения иметь одно из значений из массива **values**.

RequireRole(roles): пользователь должен принадлежать к одной из ролей из массива **roles**.

RequireUserName(name): для соответствия политике пользователь должен иметь ник (логин) **name**.

RequireAssertion(handler): запрос должен соответствовать условию, которое устанавливается с помощью делегата **handler**.

AddRequirements(requirement): позволяет добавить кастомное ограничение **requirement**, если имеющихся недостаточно.

Фактически данные методы задают ограничения, которым должен соответствовать пользователь, обращающийся к приложению. После установки ограничений политики в атрибуте **Authorize** можем их применять для разграничения доступа:

```
1 [Authorize(Policy = "OnlyForMicrosoft")]
```

Для установки политики у атрибута **AuthorizeAttribute** применяется свойство **Policy**. Оно указывает на название политики, которой должны соответствовать пользователи.

Применение авторизации на основе Claims

Допустим, у нас есть следующий класс, который представляет пользователя:

```
1 record class Person(string Email, string Password, string City, string Company);
```

У класса **Person** кроме свойств для хранения **email** и пароля также определено свойство **City** для хранения города и свойство **Company** для хранения компании пользователя.

Определим в приложении авторизацию на основе свойств **City** и **Company**. Для этого изменим код файла **Program.cs** следующим образом:

```
1 using Microsoft.AspNetCore.Authentication.Cookies;
2 using System.Security.Claims;
3 using Microsoft.AspNetCore.Authentication;
4 using Microsoft.AspNetCore.Authorization;
5
6 var people = new List<Person>
7 {
8     new Person("tom@gmail.com", "12345", "London", "Microsoft"),
9     new Person("bob@gmail.com", "55555", "Лондон", "Google"),
10    new Person("sam@gmail.com", "11111", "Berlin", "Microsoft")
11 };
12
13 var builder = WebApplication.CreateBuilder();
14 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
15     .AddCookie(options =>
16     {
17         options.LoginPath = "/login";
18         options.AccessDeniedPath = "/login";
19     });
20 builder.Services.AddAuthorization(opts => {
21
22     opts.AddPolicy("OnlyForLondon", policy => {
23         policy.RequireClaim(ClaimTypes.Locality, "Лондон", "London");
24     });
25     opts.AddPolicy("OnlyForMicrosoft", policy => {
26         policy.RequireClaim("company", "Microsoft");
27     });
28 });
29
30 var app = builder.Build();
31
32 app.UseAuthentication();
33 app.UseAuthorization();
34
```



```

35 app.MapGet("/login", async (HttpContext context) =>
36 {
37     context.Response.ContentType = "text/html; charset=utf-8";
38     // html-форма для ввода логина/пароля
39     string loginForm = @"<!DOCTYPE html>
40     <html>
41     <head>
42         <meta charset='utf-8' />
43         <title>METANIT.COM</title>
44     </head>
45     <body>
46         <h2>Login Form</h2>
47         <form method='post'>
48             <p>
49                 <label>Email</label><br />
50                 <input name='email' />
51             </p>
52             <p>
53                 <label>Password</label><br />
54                 <input type='password' name='password' />
55             </p>
56             <input type='submit' value='Login' />
57         </form>
58     </body>
59     </html>";
60     await context.Response.WriteAsync(loginForm);
61 });
62
63 app.MapPost("/login", async (string? returnUrl, HttpContext context) =>
64 {
65     // получаем из формы email и пароль
66     var form = context.Request.Form;
67     // если email и/или пароль не установлены, посылаем статусный код ошибки 400
68     if (!form.ContainsKey("email") || !form.ContainsKey("password"))
69         return Results.BadRequest("Email и/или пароль не установлены");
70     string email = form["email"];
71     string password = form["password"];
72

```

```

73 // находим пользователя
74 Person? person = people.FirstOrDefault(p => p.Email == email && p.Password == password);
75 // если пользователь не найден, отправляем статусный код 401
76 if (person is null) return Results.Unauthorized();
77 var claims = new List<Claim>
78 {
79     new Claim(ClaimTypes.Name, person.Email),
80     new Claim(ClaimTypes.Locality, person.City),
81     new Claim("company", person.Company)
82 };
83 var claimsIdentity = new ClaimsIdentity(claims, "Cookies");
84 var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
85 await context.SignInAsync(claimsPrincipal);
86 return Results.Redirect(returnUrl ?? "/");
87 });
88 // доступ только для City = London
89 app.Map("/london", [Authorize(Policy = "OnlyForLondon")]() => "You are living in London");
90
91 // доступ только для Company = Microsoft
92 app.Map("/microsoft", [Authorize(Policy = "OnlyForMicrosoft")]() => "You are working in Microsoft");
93
94 app.Map("/", [Authorize](HttpContext context) =>
95 {
96     var login = context.User.FindFirst(ClaimTypes.Name);
97     var city = context.User.FindFirst(ClaimTypes.Locality);
98     var company = context.User.FindFirst("company");
99     return $"Name: {login?.Value}\nCity: {city?.Value}\nCompany: {company?.Value}";
100 });
101 app.MapGet("/logout", async (HttpContext context) =>
102 {
103     await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
104     return "Данные удалены";
105 });
106
107 app.Run();
108
109 record class Person(string Email, string Password, string City, string Company);

```

Здесь для тестирования механизма авторизации на основе **Claims** определена условная БД – список пользователей **people**:

```
1 var people = new List<Person>
2 {
3     new Person("tom@gmail.com", "12345", "London", "Microsoft"),
4     new Person("bob@gmail.com", "55555", "Лондон", "Google"),
5     new Person("sam@gmail.com", "11111", "Berlin", "Microsoft")
6 };
```

Для настройки авторизации в зависимости от данных пользователя в делегате в методе **AddAuthorization** устанавливаются две политики доступа – **"OnlyForLondon"** и **"OnlyForMicrosoft"**:

```
1 builder.Services.AddAuthorization(opts => {
2
3     opts.AddPolicy("OnlyForLondon", policy => {
4         policy.RequireClaim(ClaimTypes.Locality, "Лондон", "London");
5     });
6     opts.AddPolicy("OnlyForMicrosoft", policy => {
7         policy.RequireClaim("company", "Microsoft");
8     });
9 });
```

Политика **"OnlyForLondon"** требует, чтобы claim с типом **ClaimTypes.Locality** имел значение **"London"** или **"Лондон"**. Если значений много, то мы их можем перечислить через запятую. Вторая политика – **"OnlyForMicrosoft"** требует наличия **Claim** с типом **"company"** и значением **"Microsoft"**.

Для входа пользователей в приложение определена конечная точка **app.MapGet("/login")**, которая обрабатывает **GET**-запросы по пути **"/login"** и отправляет пользователям форму для ввода логина и пароля.

После заполнения и отправки формы логина данные в **POST**-запросе получает конечная точка **app.MapPost("/login")**, которая получает логин и пароль и по ним находит пользователя в списке **people**. Значения свойств найденного пользователя добавляются в список **claims**:

```
1 var claims = new List<Claim>
2 {
3     new Claim(ClaimTypes.Name, person.Email),
4     new Claim(ClaimTypes.Locality, person.City),
5     new Claim("company", person.Company)
6 };
7 var claimsIdentity = new ClaimsIdentity(claims, "Cookies");
8 var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
```

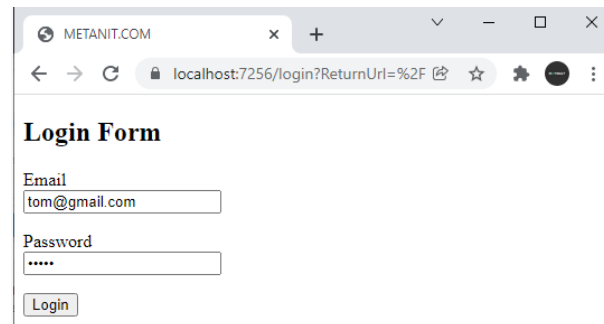
Благодаря этому инфраструктура **ASP.NET Core** сможет получить значения **claims** с типами **ClaimTypes.Locality** и **"company"** (то есть соответственно город и компанию пользователей) и на их основе решить, предоставлять ли доступ пользователю к ресурсам приложения, которые используют политику доступа на основе этих **Claim**.

Для тестирования политик доступа определены две конечных точки:

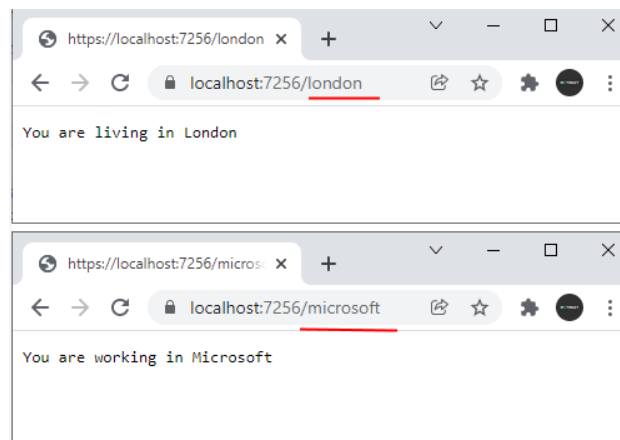
```
1 // доступ только для City = London
2 app.Map("/london", [Authorize(Policy = "OnlyForLondon")]() => "You are living in London");
3
4 // доступ только для Company = Microsoft
5 app.Map("/microsoft", [Authorize(Policy = "OnlyForMicrosoft")]() => "You are working in Microsoft");
```

Здесь доступ по пути **"/london"** имеют только те пользователи, которые удовлетворяют политике **"OnlyForLondon"**. А ресурс **"/microsoft"** доступен только для пользователей, соответствующих политике **"OnlyForMicrosoft"**.

Запустим проект и залогинимся, используя данные одного из пользователей из списка **people**:



И если пользователь живет в Лондоне, то он имеет доступ по пути **"/london"**. Аналогично если пользователь работает в Microsoft, он имеет доступ по пути **"/microsoft"**:



5. Создание ограничений для авторизации

Хотя встроенный функционал по созданию политик авторизации покрывает множество случаев для их определения, но он имеет ограниченные возможности. Например, пусть у нас есть класс **Person**, где свойство **Year** хранит год рождения пользователя:

```
1 record class Person(string Email, string Password, int Year);
```

Что если мы хотим ограничить доступ в зависимости от возраста пользователя? Для этого можно создать свое собственное ограничение. Для этого определим в проекте класс, который назовем **AgeRequirement**:

```
1 using Microsoft.AspNetCore.Authorization;
2
3 class AgeRequirement : IAuthorizationRequirement
4 {
5     protected internal int Age { get; set; }
6     public AgeRequirement(int age) => Age = age;
7 }
```

Класс ограничения должен реализовать интерфейс **IAuthorizationRequirement** из пространства имен **Microsoft.AspNetCore.Authorization**. С помощью свойства **Age** устанавливается минимально допустимый возраст.

Сам класс ограничения только устанавливает некоторые лимиты, больше он ничего не делает. Чтобы его использовать при обработке запроса, нам надо добавить специальный класс – обработчик. Итак, добавим в проект еще один новый класс, который назовем **AgeHandler**:

```

1 using System.Security.Claims;
2 using Microsoft.AspNetCore.Authorization;
3
4 class AgeHandler : AuthorizationHandler<AgeRequirement>
5 {
6     protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
7         AgeRequirement requirement)
8     {
9         // получаем claim с типом ClaimTypes.DateOfBirth - год рождения
10        var yearClaim = context.User.FindFirst(c => c.Type == ClaimTypes.DateOfBirth);
11        if (yearClaim is not null)
12        {
13            // если claim года рождения хранит число
14            if (int.TryParse(yearClaim.Value, out var year))
15            {
16                // и разница между текущим годом и годом рождения больше требуемого возраста
17                if ((DateTime.Now.Year - year) >= requirement.Age)
18                {
19                    context.Succeed(requirement); // сигнализируем, что claim соответствует ограничению
20                }
21            }
22        }
23        return Task.CompletedTask;
24    }
25 }

```

Класс обработчика должен наследоваться от класса **AuthorizationHandler<T>**, где параметр **T** представляет тип ограничения. Вся обработка производится в методе **HandleRequirementAsync()**. Этот метод вызывается системой авторизации при доступе к ресурсу, к которому применяется ограничение, используемое обработчиком.

В качестве параметров метод **HandleRequirementAsync()** получает объект применяемого ограничения и контекст авторизации **AuthorizationHandlerContext**, который содержит информацию о запросе. В частности, через свойство **User** он возвращает объект **ClaimPrincipal**, представляющий текущего пользователя.

А методы класса **AuthorizationHandlerContext** позволяют управлять авторизацией. Так, метод **Succeed(requirement)** вызывается, если запрос соответствует ограничению **requirement**.

И наоборот, метод **Fail()**, если запрос не соответствует ограничению.

В данном случае мы получаем для текущего пользователя **claim** с типом **ClaimTypes.DateOfBirth**. Предполагается, что этот **claim** содержит год рождения пользователя. И далее по этому году получаем возраст пользователя относительно текущей даты. И если возраст оказался больше минимально допустимого, то вызываем метод **context.Succeed(requirement)**. Вызов этого метода будет означать, что работа обработчика завершилась успешно. Если этот метод не вызывается, то считается, что авторизация прошла неудачно.

Определим в файле **Program.cs** следующий код:

```
1 using Microsoft.AspNetCore.Authentication.Cookies;
2 using System.Security.Claims;
3 using Microsoft.AspNetCore.Authentication;
4 using Microsoft.AspNetCore.Authorization;
5
6 var people = new List<Person>
7 {
8     new Person("tom@gmail.com", "12345", 1984),
9     new Person("bob@gmail.com", "55555", 2006)
10 };
11
12 var builder = WebApplication.CreateBuilder();
13 // встраиваем сервис AgeHandler
14 builder.Services.AddTransient<IAuthorizationHandler, AgeHandler>();
15 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
16     .AddCookie(options =>
17     {
18         options.LoginPath = "/login";
19         options.AccessDeniedPath = "/login";
20     });
21 builder.Services.AddAuthorization(opts => {
22     // устанавливаем ограничение по возрасту
23     opts.AddPolicy("AgeLimit", policy => policy.Requirements.Add(new AgeRequirement(18)));
24 });
25
26 var app = builder.Build();
27
28 app.UseAuthentication();
29 app.UseAuthorization();
30
```

```

31 app.MapGet("/login", async (HttpContext context) =>
32 {
33     context.Response.ContentType = "text/html; charset=utf-8";
34     // html-форма для ввода логина/пароля
35     string loginForm = @"<!DOCTYPE html>
36     <html>
37     <head>
38         <meta charset='utf-8' />
39         <title>METANIT.COM</title>
40     </head>
41     <body>
42         <h2>Login Form</h2>
43         <form method='post'>
44             <p>
45                 <label>Email</label><br />
46                 <input name='email' />
47             </p>
48             <p>
49                 <label>Password</label><br />
50                 <input type='password' name='password' />
51             </p>
52             <input type='submit' value='Login' />
53         </form>
54     </body>
55     </html>";
56     await context.Response.WriteAsync(loginForm);
57 });
58
59 app.MapPost("/login", async (string? returnUrl, HttpContext context) =>
60 {
61     // получаем из формы email и пароль
62     var form = context.Request.Form;
63     // если email и/или пароль не установлены, посылаем статусный код ошибки 400
64     if (!form.ContainsKey("email") || !form.ContainsKey("password"))
65         return Results.BadRequest("Email и/или пароль не установлены");
66     string email = form["email"];
67     string password = form["password"];
68

```



```

69 // находим пользователя
70 Person? person = people.FirstOrDefault(p => p.Email == email && p.Password == password);
71 // если пользователь не найден, отправляем статусный код 401
72 if (person is null) return Results.Unauthorized();
73 var claims = new List<Claim>
74 {
75     new Claim(ClaimTypes.Name, person.Email),
76     new Claim(ClaimTypes.DateOfBirth, person.Year.ToString())
77 };
78 var claimsIdentity = new ClaimsIdentity(claims, "Cookies");
79 var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
80 await context.SignInAsync(claimsPrincipal);
81 return Results.Redirect(returnUrl ?? "/");
82 });
83 // доступ только для тех, кто соответствует ограничению AgeLimit
84 app.Map("/age", [Authorize(Policy = "AgeLimit")]() => "Age Limit is passed");
85
86 app.Map("/", [Authorize](HttpContext context) =>
87 {
88     var login = context.User.FindFirst(ClaimTypes.Name);
89     var year = context.User.FindFirst(ClaimTypes.DateOfBirth);
90     return $"Name: {login?.Value}\nYear: {year?.Value}";
91 });
92 app.MapGet("/logout", async (HttpContext context) =>
93 {
94     await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
95     return "Данные удалены";
96 });
97
98 app.Run();

```

Здесь для тестирования механизма авторизации по возрасту определена условная БД – список пользователей **people**:

```

1 var people = new List<Person>
2 {
3     new Person("tom@gmail.com", "12345", 1984),
4     new Person("bob@gmail.com", "55555", 2006)
5 };

```

Для настройки авторизации в коллекции сервисов необходимо зарегистрировать зависимость для сервиса **IAuthorizationHandler**:

```
1 builder.Services.AddTransient<IAuthorizationHandler, AgeHandler>();
```

Далее добавляем политику **"AgeLimit"**? для которой добавляется кастомное ограничение:

```
1 builder.Services.AddAuthorization(opts => {
2     // устанавливаем ограничение по возрасту
3     opts.AddPolicy("AgeLimit", policy => policy.Requirements.Add(new AgeRequirement(18)));
4 });
```

Для входа пользователей в приложение определена конечная точка **app.MapGet("/login")**, которая обрабатывает **GET**-запросы по пути **"login"** и отправляет пользователям форму для ввода логина и пароля.

После заполнения и отправки формы логина данные в **POST**-запросе получает конечная точка **app.MapPost("/login")**, которая получает логин и пароль и по ним находит пользователя в списке **people**. Значения год рождения найденного пользователя добавляется в список **claims**:

```
1 var claims = new List<Claim>
2 {
3     new Claim(ClaimTypes.Name, person.Email),
4     new Claim(ClaimTypes.DateOfBirth, person.Year.ToString())
5 };
6 var claimsIdentity = new ClaimsIdentity(claims, "Cookies");
7 var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
```

И далее мы можем использовать созданную политику для ограничения доступа. Для тестирования политики определена следующая конечная точка:

```
1 // доступ только для тех, кто соответствует ограничению AgeLimit
2 app.Map("/age", [Authorize(Policy = "AgeLimit")]() => "Age Limit is passed");
```

Таким образом, по адресу **"age"** смогут обратиться только те, кто удовлетворяет ограничению **AgeLimit** (в данном случае кому исполнилось 18 лет).