



Распределенные информационно-аналитические СИСТЕМЫ

Практическое занятие № 15. «Аутентификация»

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

Учебные вопросы:

1. Введение в аутентификацию.
2. Аутентификация с помощью JWT-токенов.
3. Аутентификация с помощью куки.
4. HttpContext.User, ClaimPrincipal и ClaimsIdentity.
5. ClaimPrincipal и объекты Claim.

1. Введение в аутентификацию

Важное место в приложении занимает аутентификация. Аутентификация представляет процесс определения пользователя. То есть, аутентификация отвечает на вопрос "Кем является пользователь?". **ASP.NET Core** имеет встроенную поддержку аутентификации.

Аутентификация

Для выполнения аутентификации в конвейере обработки запроса отвечает специальный компонент **middleware** – **AuthenticationMiddleware**. Для встраивания этого **middleware** в конвейер применяется метод расширения **UseAuthentication()**

```
1 public static IApplicationBuilder UseAuthentication (this IApplicationBuilder app);
```

Следует отметить, что метод **UseAuthentication()** должен встраиваться в конвейер до любых компонентов **middleware**, которые используют аутентификацию пользователей.

Для выполнения аутентификации этот компонент использует сервисы аутентификации, в частности, сервис **AuthenticationService**, которые регистрируются в приложении с помощью метода **AddAuthentication()**:

```
1 public static AuthenticationBuilder AddAuthentication(this IServiceCollection services)
2 public static AuthenticationBuilder AddAuthentication(this IServiceCollection services, string defaultScheme)
3 public static AuthenticationBuilder AddAuthentication(this IServiceCollection services, Action<AuthenticationOptions> configureOptions)
```

В качестве параметра вторая версия метода **AddAuthentication()** принимает схему аутентификации в виде строки. Третья версия метода **AddAuthentication** принимает делегат, который устанавливает опции аутентификации – объект **AuthenticationOptions**.

Какую бы мы версию метода не использовали, для аутентификации необходима установить схему аутентификации. Две наиболее **распространенные схемы аутентификации**:

"Cookies": аутентификация на основе куки. Хранится в константе **CookieAuthenticationDefaults.AuthenticationScheme**.

"Bearer": аутентификация на основе **jwt**-токенов. Хранится в константе **JwtBearerDefaults.AuthenticationScheme**.

Схема аутентификации позволяет выбирать определенный обработчик аутентификации. Обработчик аутентификации, собственно, и выполняет непосредственную аутентификацию пользователей на основе данных запросов и исходя из схемы аутентификации.

Например, для аутентификации с помощью куки передается схема **"Cookies"**. Соответственно для аутентификации пользователя будет выбираться встроенный обработчик аутентификации – класс **Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationHandler**, который на основе полученных в запросе **cookie** выполняет аутентификацию.

А если используется схема **"Bearer"**, то это значит, что для аутентификации будет использоваться **jwt**-токен, а в качестве обработчика аутентификации будет применяться класс **Microsoft.AspNetCore.Authentication.JwtBearer.JwtBearerHandler**. Стоит отметить, что для аутентификации с помощью **jwt**-токенов необходимо добавить в проект через **Nuget** пакет **Microsoft.AspNetCore.Authentication.JwtBearer**.

При чем в **ASP.NET Core** мы не ограничены встроенными схемами аутентификации и можем создавать свои кастомные схемы и под них своих обработчиков аутентификации.

Кроме применения схемы аутентификации необходимо подключить аутентификацию определенного типа. Для этого можно использовать следующие **методы**:

AddCookie(): подключает и конфигурирует аутентификацию с помощью куки.

AddJwtBearer(): подключает и конфигурирует аутентификацию с помощью **jwt**-токенов (для этого метода необходим **Nuget**-пакет **Microsoft.AspNetCore.Authentication.JwtBearer**).

Оба метода реализованы как методы расширения для типа **AuthenticationBuilder**, который возвращается методом **AddAuthentication()**:

```
1 var builder = WebApplication.CreateBuilder();
2 // добавление сервисов аутентификации
3 builder.Services.AddAuthentication("Bearer") // схема аутентификации - с помощью jwt-токенов
4     .AddJwtBearer(); // подключение аутентификации с помощью jwt-токенов
5
6 var app = builder.Build();
7
8 app.UseAuthentication(); // добавление middleware аутентификации
```

2. Аутентификация с помощью JWT-токенов

Одним из подходов к авторизации и аутентификации в **ASP.NET Core** представляет механизм аутентификации и авторизации с помощью **JWT**-токенов. Что такое **JWT**-токен? **JWT** (или **JSON Web Token**) представляет собой веб-стандарт, который определяет способ передачи данных о пользователе в формате **JSON** в зашифрованном виде.

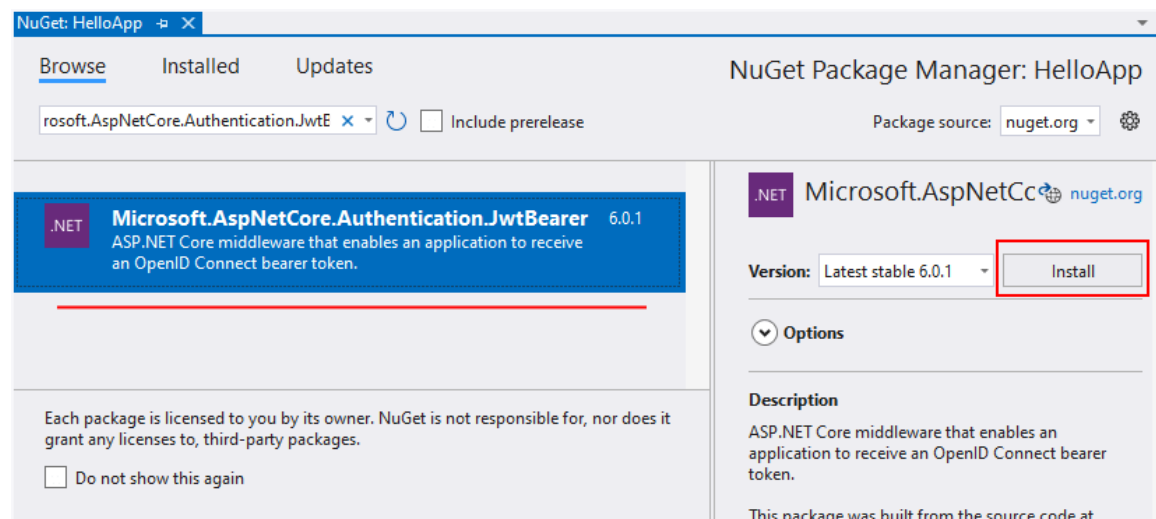
JWT-токен состоит из **трех частей**:

Header – объект **JSON**, который содержит информацию о типе токена и алгоритме его шифрования.

Payload – объект **JSON**, который содержит данные, нужные для авторизации пользователя.

Signature – строка, которая создается с помощью секретного кода, **Header** и **Payload**. Эта строка служит для верификации токена.

Для использования **JWT**-токенов в проект **ASP.NET Core** необходимо добавить **Nuget**-пакет **Microsoft.AspNetCore.Authentication.JwtBearer**.



Сначала рассмотрим принцип генерации и отправки **jwt**-токена. Для этого в файле **Program.cs** определим следующий код приложения:

```
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 var builder = WebApplication.CreateBuilder();
9
10 builder.Services.AddAuthorization();
11 builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
12     .AddJwtBearer(options =>
13     {
14         options.TokenValidationParameters = new TokenValidationParameters
15         {
16             // указывает, будет ли валидироваться издатель при валидации токена
17             ValidateIssuer = true,
18             // строка, представляющая издателя
19             ValidIssuer = AuthOptions.ISSUER,
20             // будет ли валидироваться потребитель токена
21             ValidateAudience = true,
22             // установка потребителя токена
23             ValidAudience = AuthOptions.AUDIENCE,
24             // будет ли валидироваться время существования
25             ValidateLifetime = true,
26             // установка ключа безопасности
27             IssuerSigningKey = AuthOptions.GetSymmetricSecurityKey(),
28             // валидация ключа безопасности
29             ValidateIssuerSigningKey = true,
30         };
31     });
32 var app = builder.Build();
33
```

```

34 app.UseAuthentication();
35 app.UseAuthorization();
36
37 app.Map("/login/{username}", (string username) =>
38 {
39     var claims = new List<Claim> {new Claim(ClaimTypes.Name, username) };
40     // создаем JWT-токен
41     var jwt = new JwtSecurityToken(
42         issuer: AuthOptions.ISSUER,
43         audience: AuthOptions.AUDIENCE,
44         claims: claims,
45         expires: DateTime.UtcNow.Add(TimeSpan.FromMinutes(2)),
46         signingCredentials: new SigningCredentials(AuthOptions.GetSymmetricSecurityKey(), SecurityAlgorithms.HmacSha256));
47
48     return new JwtSecurityTokenHandler().WriteToken(jwt);
49 });
50
51 app.Map("/data", [Authorize] () => new { message= "Hello World!" });
52
53 app.Run();
54
55 public class AuthOptions
56 {
57     public const string ISSUER = "MyAuthServer"; // издатель токена
58     public const string AUDIENCE = "MyAuthClient"; // потребитель токена
59     const string KEY = "mysupersecret_secretkey!123"; // ключ для шифрации
60     public static SymmetricSecurityKey GetSymmetricSecurityKey() =>
61         new SymmetricSecurityKey(Encoding.UTF8.GetBytes(KEY));
62 }

```

Для описания некоторых настроек генерации токена в конце кода определен специальный класс **AuthOptions**:

```

1 public class AuthOptions
2 {
3     public const string ISSUER = "MyAuthServer"; // издатель токена
4     public const string AUDIENCE = "MyAuthClient"; // потребитель токена
5     const string KEY = "mysupersecret_secretkey!123"; // ключ для шифрации
6     public static SymmetricSecurityKey GetSymmetricSecurityKey() =>
7         new SymmetricSecurityKey(Encoding.UTF8.GetBytes(KEY));
8 }

```

Константа **ISSUER** представляет издателя токена. Здесь можно определить любое название.

Константа **AUDIENCE** представляет потребителя токена – опять же может быть любая строка, обычно это сайт, на котором применяется токен.

Константа **KEY** хранит ключ, который будет применяться для создания токена.

И метод **GetSymmetricSecurityKey()** возвращает ключ безопасности, который применяется для генерации токена. Для генерации токена нам необходим объект класса **SecurityKey**. В качестве такого здесь выступает объект производного класса **SymmetricSecurityKey**, в конструктор которого передается массив байт, созданный по секретному ключу.

Чтобы указать, что приложение для аутентификации будет использовать токена, в метод **AddAuthentication()** передается значение константы **JwtBearerDefaults.AuthenticationScheme**.

```
1 builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
```

Конфигурация и валидация токена

С помощью метода **AddJwtBearer()** в приложение добавляется конфигурация токена. Для конфигурации токена применяется объект **JwtBearerOptions**, который позволяет с помощью свойств настроить работу с токеном. Данный объект имеет множество свойств. Здесь же использовано только свойство **TokenValidationParameters**, которое задает параметры валидации токена.

```
1 builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
2     .AddJwtBearer(options =>
3     {
4         options.TokenValidationParameters = new TokenValidationParameters
5         {
6             ValidateIssuer = true,
7             ValidIssuer = AuthOptions.ISSUER,
8             ValidateAudience = true,
9             ValidAudience = AuthOptions.AUDIENCE,
10            ValidateLifetime = true,
11            IssuerSigningKey = AuthOptions.GetSymmetricSecurityKey(),
12            ValidateIssuerSigningKey = true,
13        };
14    });
```


Объект **TokenValidationParameters** обладает множеством свойств, которые позволяют настроить различные аспекты валидации токена. В данном случае применяются следующие **свойства**:

ValidateIssuer: указывает, будет ли валидироваться издатель при валидации токена.

ValidIssuer: строка, которая представляет издателя токена.

ValidateAudience: указывает, будет ли валидироваться потребитель токена.

ValidAudience: строка, которая представляет потребителя токена.

ValidateLifetime: указывает, будет ли валидироваться время существования.

IssuerSigningKey: представляет ключ безопасности – объект **SecurityKey**, который будет применяться при генерации токена.

ValidateIssuerSigningKey: указывает, будет ли валидироваться ключ безопасности.

Здесь устанавливаются наиболее основные свойства. А вообще можно установить кучу других параметров, например, названия **claims** для ролей и логинов пользователя и т.д.

Генерация токена

Чтобы пользователь мог использовать токен, приложение должно отправить ему этот токен, а перед этим соответственно сгенерировать токен. И для генерации токена здесь предусмотрена типовая конечная точка **"/login"**:

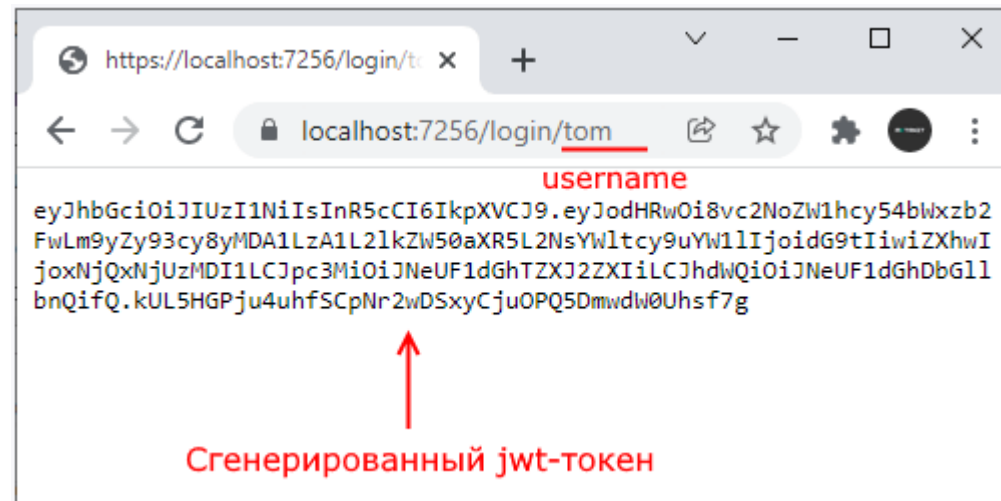
```
1 app.Map("/login/{username}", (string username) =>
2 {
3     var claims = new List<Claim> {new Claim(ClaimTypes.Name, username) };
4     var jwt = new JwtSecurityToken(
5         issuer: AuthOptions.ISSUER,
6         audience: AuthOptions.AUDIENCE,
7         claims: claims,
8         expires: DateTime.UtcNow.Add(TimeSpan.FromMinutes(2)), // время действия 2 минуты
9         signingCredentials: new SigningCredentials(AuthOptions.GetSymmetricSecurityKey(), SecurityAlgorithms.HmacSha256));
10
11     return new JwtSecurityTokenHandler().WriteToken(jwt);
12 });
```

Для простоты конечная точка через параметр маршрута **"username"** получает некоторый логин пользователя и применяет его для генерации токена. На данном этапе для простоты мы пока ничего не проверяем, валидный ли это логин, что это за логин, пока просто смотрим, как генерировать токен.

Для создания токена применяется конструктор **JwtSecurityToken**. Одним из параметров служит список объектов **Claim**. Объекты **Claim** служат для хранения некоторых данных о пользователе, описывают пользователя. Затем эти данные можно применять для аутентификации. В данном случае добавляем в список один **Claim**, который хранит логин пользователя.

Затем, собственно, создаем **JWT**-токен, передавая в конструктор **JwtSecurityToken** соответствующие параметры. Обратите внимание, что для инициализации токена применяются все те же константы и ключ безопасности, которые определены в классе **AuthOptions** и которые использовались для конфигурации настроек в методе **AddJwtBearer()**.

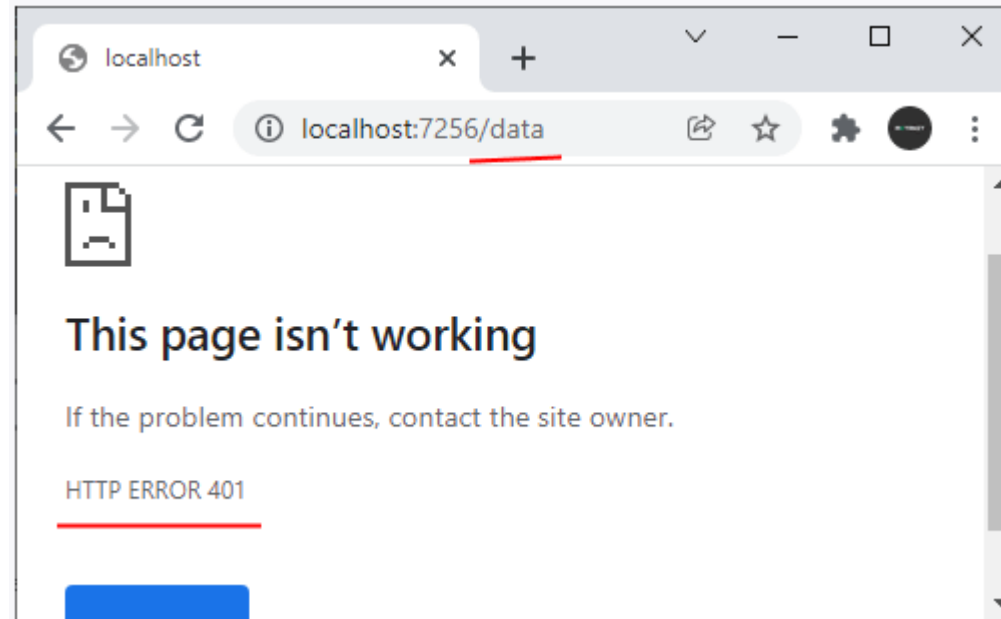
В конце посредством метода **JwtSecurityTokenHandler().WriteToken(jwt)** создается сам токен, который отправляется клиенту. Для тестирования генерации токена обратимся к этой конечной точке:



При обращении к конечной точке **"/login"** (например, по пути **"/login/tom"**, где **"tom"** представляет параметр **"username"**) приложение сгенерирует нам **jwt**-токен, который нам необходимо отправлять для доступа к ресурсам приложения с защищенным доступом. Например, в коде также определена еще одна конечная точка **"/data"**:

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

Она применяет атрибут **Authorize**, соответственно доступ к ней ограничен только для аутентифицированных пользователей, которые имеют токен. Например, если мы попытаемся обратиться по пути **"/data"**, мы столкнемся с ошибкой **401 (Unauthorized)** – доступ не авторизован:



Поэтому для обращения к этому ресурсу (и ко всем другим ресурсам, к которым имеют доступ только аутентифицированные пользователи) необходимо посылать полученный токен в запросе в заголовке **Authorization**:

```
1 "Authorization": "Bearer " + token // token - полученный ранее jwt-токен
```

3. Аутентификация с помощью куки

Одним из распространенных способов аутентификации в веб-приложениях является аутентификация с помощью куки. И **ASP.NET Core** имеет встроенную поддержку для данного типа аутентификации.

Для применения аутентификации с помощью куки в метод **AddAuthentication()** передается схема **"Cookies"**:

```
1 builder.Services.AddAuthentication("Cookies")
```

Чтобы не ошибиться в написании схемы еще можно передавать константу **CookieAuthenticationDefaults.AuthenticationScheme**, которая имеет то же самое значение.

```
1 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
```

Кроме того, для настройки аутентификации с помощью куки необходимо вызвать метод **AddCookie()**, который реализован как метод расширения для типа **AuthenticationBuilder**:

```
1 public static AuthenticationBuilder AddCookie(this AuthenticationBuilder builder, Action<CookieAuthenticationOptions> configureOptions);
```

В качестве параметра метод принимает делегат, который с помощью объекта **CookieAuthenticationOptions** устанавливает настройки аутентификации.

Рассмотрим на примере, как использовать самую простейшую аутентификацию с помощью куки. Для этого определим в файле **Program.cs** следующий код:

```
1 using Microsoft.AspNetCore.Authorization;
2 using Microsoft.AspNetCore.Authentication.Cookies;
3 using System.Security.Claims;
4 using Microsoft.AspNetCore.Authentication;
5
6 var builder = WebApplication.CreateBuilder();
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 // аутентификация с помощью куки
15 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
16     .AddCookie(options => options.LoginPath = "/login");
17 builder.Services.AddAuthorization();
18
19 var app = builder.Build();
20
21 app.UseAuthentication(); // добавление middleware аутентификации
22 app.UseAuthorization(); // добавление middleware авторизации
23
24 app.MapGet("/login", async (HttpContext context) =>
25 {
26     context.Response.ContentType = "text/html; charset=utf-8";
27     // html-форма для ввода логина/пароля
28     string loginForm = @"<!DOCTYPE html>
29     <html>
30     <head>
31         <meta charset='utf-8' />
32         <title>METANIT.COM</title>
33     </head>
34     <body>
35         <h2>Login Form</h2>
36         <form method='post'>
37             <p>
38                 <label>Email</label><br />
39                 <input name='email' />
40             </p>
41             <p>
42                 <label>Password</label><br />
43                 <input type='password' name='password' />
```

```

44         </p>
45         <input type='submit' value='Login' />
46     </form>
47 </body>
48 </html>";
49     await context.Response.WriteAsync(loginForm);
50 });
51
52 app.MapPost("/login", async (string? returnUrl, HttpContext context) =>
53 {
54     // получаем из формы email и пароль
55     var form = context.Request.Form;
56     // если email и/или пароль не установлены, посылаем статусный код ошибки 400
57     if (!form.ContainsKey("email") || !form.ContainsKey("password"))
58         return Results.BadRequest("Email и/или пароль не установлены");
59
60     string email = form["email"];
61     string password = form["password"];
62
63     // находим пользователя
64     Person? person = people.FirstOrDefault(p => p.Email == email && p.Password == password);
65     // если пользователь не найден, отправляем статусный код 401
66     if (person is null) return Results.Unauthorized();
67
68     var claims = new List<Claim> { new Claim(ClaimTypes.Name, person.Email) };
69     // создаем объект ClaimsIdentity
70     ClaimsIdentity claimsIdentity = new ClaimsIdentity(claims, "Cookies");
71     // установка аутентификационных куки
72     await context.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme, new ClaimsPrincipal(claimsIdentity));
73     return Results.Redirect(returnUrl ?? "/" );
74 });
75
76 app.MapGet("/logout", async (HttpContext context) =>
77 {
78     await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
79     return Results.Redirect("/login");
80 });
81
82 app.Map("/", [Authorize]() => $"Hello World!");
83
84 app.Run();
85
86 record class Person(string Email, string Password);

```

Для представления пользователя класс **Person**:

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

В качестве условной базы данных для теста используется список **people**:

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

Именно с этими данными мы будем сравнивать присланные от клиента логин и пароль.
Для подключения аутентификации куки регистрируем соответствующие сервисы:

```
1 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
2     .AddCookie(options => options.LoginPath = "/login");
```

Свойство **LoginPath** класса **CookieAuthenticationOptions** указывает на путь, по которому неаутентифицированный клиент будет автоматически переадресовываться при обращении к ресурсу, для доступа к которому требуется аутентификация.

Конечная точка, которая обрабатывает **get**-запросы по пути **"/login"**, будет отправлять в ответ **html**-форму для ввода **email** и пароля:

```
1 app.MapGet("/login", async (HttpContext context) =>
2 {
3     context.Response.ContentType = "text/html; charset=utf-8";
4     // html-форма для ввода логина/пароля
5     string loginForm = @"<!DOCTYPE html>
6     <html>
7     <head>
8         <meta charset='utf-8' />
9         <title>METANIT.COM</title>
10    </head>
11    <body>
12        <h2>Login Form</h2>
13        <form method='post'>
14            <p>
15                <label>Email</label><br />
16                <input name='email' />
17            </p>
18            <p>
19                <label>Password</label><br />
20                <input type='password' name='password' />
21            </p>
22            <input type='submit' value='Login' />
23        </form>
24    </body>
25    </html>";
26    await context.Response.WriteAsync(loginForm);
27 };
```

В данном случае для простоты весь **html**-код определен в виде строки, но, естественно, можно сделать по-разному, например, определить отдельную **html**-страницу и ее отправлять клиенту.

На отправляемой форме клиент должен будет заполнить поля **"email"** и **"password"**, и после нажатия на кнопку отправки значения этих полей в запросе типа **POST** будут получены в обработке другой конечной точки:

```
1 app.MapPost("/login", async (string? returnUrl, HttpContext context) =>
2 {
```

Обработчик этой конечной точки принимает два параметра. Прежде всего, это передаваемый через механизм внедрения зависимостей объект контекста запроса **HttpContext**. Кроме того, система аутентификации автоматически отправляет путь, с которого пользователь был переадресован на форму логина. Поскольку у нас адрес формы логина и адрес обработки отправленных данных совпадает – **"/login"**, то через параметр **returnUrl** мы можем получить путь, по которому изначально обращался клиент. Однако поскольку клиент также может напрямую обратиться к форме логина, то в этом случае данный параметр будет иметь значение **null**.

В самом же обработчике конечной точки вначале получаем из данных формы отправленные **email** и пароль:

```
1 var form = context.Request.Form;
2 if (!form.ContainsKey("email") || !form.ContainsKey("password"))
3     return Results.BadRequest("Email и/или пароль не установлены");
4
5 string email = form["email"];
6 string password = form["password"];
```

Получив данные, проверяем, а есть ли объект с такими данными в нашей условной базе данных – списке **people**:

```
1 Person? person = people.FirstOrDefault(p => p.Email == email && p.Password == password);
2 if (person is null) return Results.Unauthorized();
```

Далее производится установка аутентификационных кук, которые будут применяться для определения клиента и его прав в приложении:


```
1 var claims = new List<Claim> { new Claim(ClaimTypes.Name, person.Email) };
2 // создаем объект ClaimsIdentity
3 ClaimsIdentity claimsIdentity = new ClaimsIdentity(claims, "Cookies");
4 // установка аутентификационных куки
5 await context.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme, new ClaimsPrincipal(claimsIdentity));
```

Для установки кук у класса **HttpContext** применяется асинхронный метод **SignInAsync()**. В качестве параметра он принимает применяемую схему аутентификации, в нашем случае это **"Cookies"**, то есть значение константы **CookieAuthenticationDefaults.AuthenticationScheme**. А в качестве второго параметра передается объект **ClaimsPrincipal**, который представляет пользователя.

Для правильного создания и настройки объекта **ClaimsPrincipal** вначале создается список **claims** – набор объектов **Claim** – грубо говоря набор данных, которые описывают пользователя. Эти данные шифруются и добавляются в аутентификационные куки. Каждый такой **claim** принимает тип и значение. В нашем случае у нас только один **claim**, который в качестве типа принимает константу **ClaimTypes.Name**, а в качестве значения – **email** пользователя.

Далее создается объект **ClaimsIdentity**, который нужен для инициализации **ClaimsPrincipal**. В **ClaimsIdentity** передается ранее созданный список **claims** и тип аутентификации, в данном случае **"Cookies"**. Тип аутентификации может представлять произвольную строку.

И после вызова метода **context.SignInAsync** будут формироваться аутентификационные куки, которые будут отправлены клиенту и при последующих запросах будут передаваться обратно на сервер, десериализоваться и использоваться для аутентификации пользователя.

В самом конце перенаправляем аутентифицированного пользователя обратно на адрес, с которого его перебросило на форму логина:

```
1 return Results.Redirect(returnUrl ?? "/");
```

Для выхода из сайта определена конечная точка, которая обрабатывает запросы по пути **"/logout"**:

```
1 app.MapGet("/logout", async (HttpContext context) =>
2 {
3     await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
4     return Results.Redirect("/login");
5 });
```

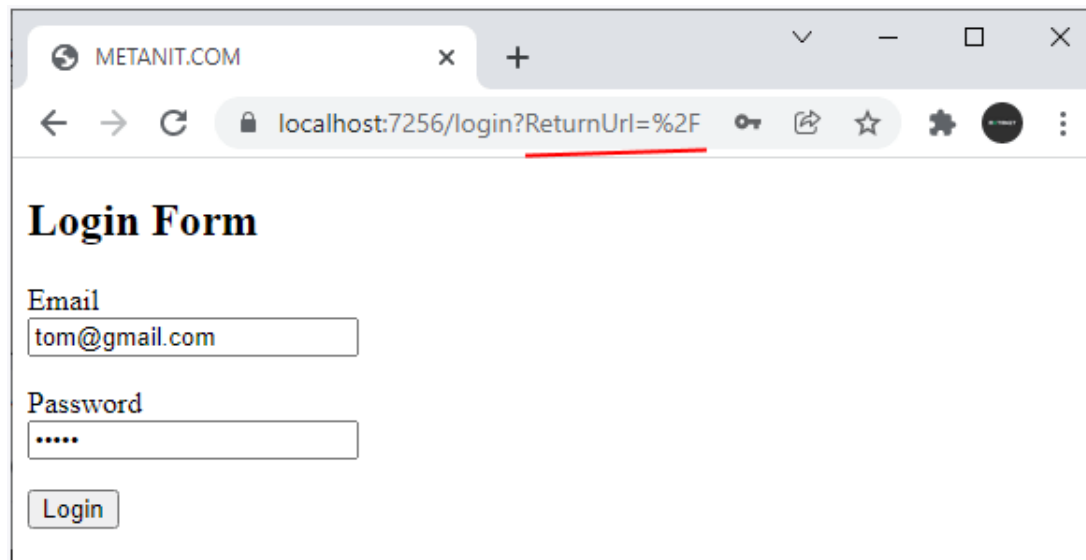
Его основа – вызов метода **context.SignOutAsync()**, который удаляет аутентификационные куки. В качестве параметра он принимает схему аутентификации.

Для тестирования авторизации определена четвертая конечная точка, которая обрабатывает запросы к корню приложения:

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

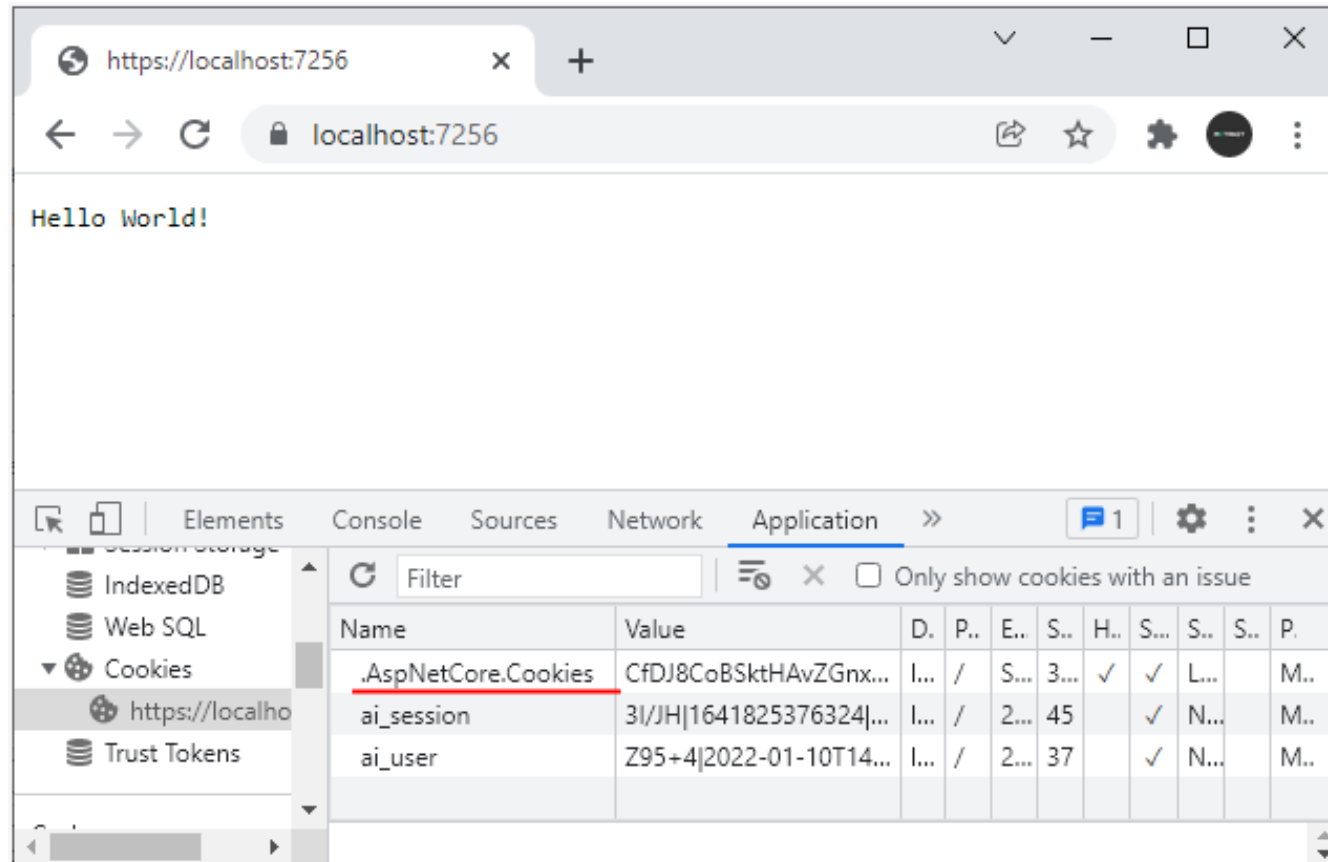
Поскольку эта конечная точка использует атрибут **Authorize**, то доступ к ней имеют только аутентифицированные пользователи.

Таким образом, когда мы, не будучи аутентифицированными, обращаемся по пути **"/"**, нас перебрасывает на путь **"/login"**, где нам надо заполнить форму логина:



The screenshot shows a web browser window with the address bar displaying `localhost:7256/login?ReturnUrl=%2F`. The page title is "Login Form". It contains two input fields: "Email" with the value "tom@gmail.com" and "Password" with masked characters ".....". Below the password field is a "Login" button.

И после ввода корректных данных и нажатия на кнопку отправки, конечная точка `app.MapPost("/login")` получит отправленные данные, проверит, есть ли с такими данными объект в списке `people`, и при наличии такого объекта установит аутентификационные куки и перенаправит пользователя обратно на адрес `"/`:



При этом в самом браузере мы сможем увидеть сохраненные аутентификационные куки, которые называются `AspNetCore.Cookies` или `AspNetCore.[Название схемы аутентификации]`.

Таким образом, мы можем добавить простейшую систему аутентификации и авторизации с помощью куки в приложение **ASP.NET Core**.

4. HttpContext.User, ClaimPrincipal и ClaimsIdentity

Классы ClaimPrincipal и ClaimsIdentity и их роль в аутентификации

Одной из задач аутентификации в приложении **ASP.NET Core** является установка пользователя, который представлен в приложении свойством **User** класса **HttpContext**:

```
1 public abstract System.Security.Claims.ClaimsPrincipal User { get; set; }
```

Данное свойство предоставляет класс **ClaimsPrincipal** из пространства имен **System.Security.Claims**.

Непосредственно данные, которые идентифицируют пользователя (его идентичность) хранятся в свойстве **Identity** класса **ClaimPrincipal**:

```
1 public virtual System.Security.Principal.IIdentity? Identity { get; }
```

Это свойство представляет основную идентичность текущего пользователя. Но поскольку с одним пользователем может быть связан набор идентичностей, то также в классе определено свойство **Identities**:

```
1 public virtual IEnumerable<ClaimsIdentity> Identities { get; }
```

Свойство **Identity** представляет интерфейс **IIdentity**, и, как правило, в качестве такой реализации применяется класс **ClaimsIdentity**.

Объект **IIdentity**, в свою очередь, предоставляет информацию о текущем пользователе через следующие **свойства**:

AuthenticationType: тип аутентификации в строковом виде.

IsAuthenticated: возвращает true, если пользователь аутентифицирован.

Name: возвращает имя пользователя. Обычно в качестве подобного имени используется логин, по которому пользователь входит в приложение.

Для создания объекта **ClaimsIdentity** можно применять ряд конструкторов, но, для того, чтобы пользователь был аутентифицирован, необходимо, как минимум, предоставить тип аутентификации, которая передается через конструктор. Тип аутентификации представляет произвольную строку, которая описывает некоторым образом способ аутентификации. Например:

```
1 var identity = new ClaimsIdentity("Cookies");
```

В данном случае в тип аутентификации называется **"Cookies"**.

Для установки идентичности пользователя объект **ClaimsIdentity** можно передать в **ClaimsPrincipal** либо через конструктор, либо через метод **AddIdentity()**:

```
1 var identity = new ClaimsIdentity("Undefined");
2 var principal = new ClaimsPrincipal(identity);
```

На примере аутентификации куки посмотрим на применение **ClaimsPrincipal** и **ClaimsIdentity**:

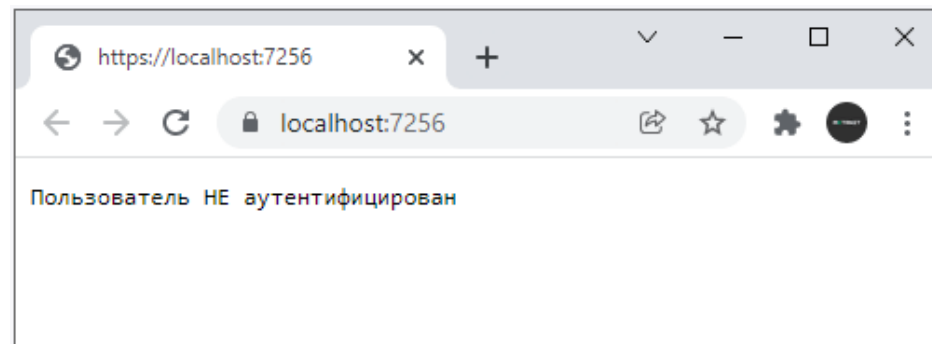
```
1 using Microsoft.AspNetCore.Authentication.Cookies;
2 using System.Security.Claims;
3 using Microsoft.AspNetCore.Authentication;
4
5 var builder = WebApplication.CreateBuilder();
6
7 // аутентификация с помощью куки
8 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
9     .AddCookie();
10
11 var app = builder.Build();
12
13 app.UseAuthentication();
14
15 app.MapGet("/login", async (HttpContext context) =>
16 {
17     var claimsIdentity = new ClaimsIdentity("Undefined");
18     var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
19     // установка аутентификационных куки
20     await context.SignInAsync(claimsPrincipal);
21     return Results.Redirect("/");
22 });
23
24 app.MapGet("/logout", async (HttpContext context) =>
25 {
26     await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
27     return "Данные удалены";
28 });
29 app.Map("/", (HttpContext context) =>
30 {
31     var user = context.User.Identity;
32     if (user is not null && user.IsAuthenticated)
33     {
34         return $"Пользователь аутентифицирован. Тип аутентификации: {user.AuthenticationType}";
35     }
36     else
37     {
38         return "Пользователь НЕ аутентифицирован";
39     }
40 });
41
42 app.Run();
```

Здесь в конечной точке `app.MapGet("/login")` создается идентичность `claimsIdentity` – объект `ClaimsIdentity` с типом аутентификации `"Undefined"`. Далее создается объект `ClaimsPrincipal`, который принимает идентичность `claimsIdentity`. Созданный объект `claimsPrincipal` затем передается в метод `context.SignInAsync()`, который, используя этот объект, устанавливает аутентификационные куки. И в конце происходит редирект на путь `"/"`.

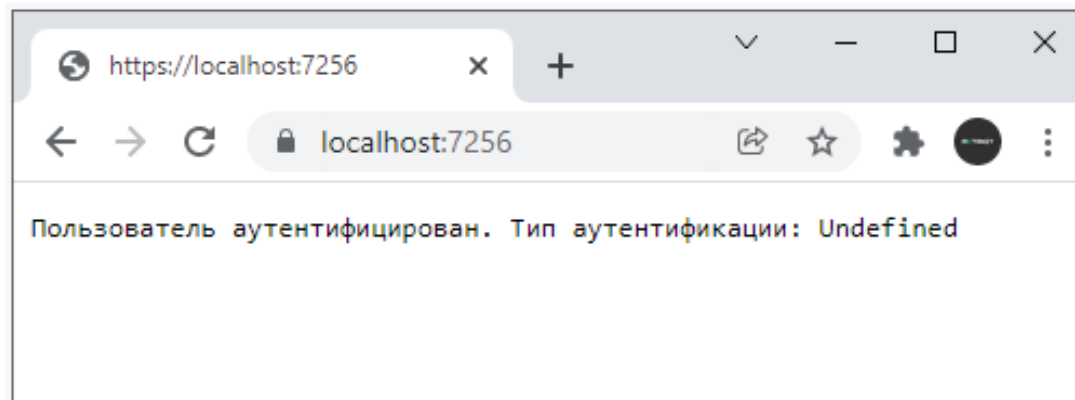
Конечная точка `app.Map("/")` получает через механизм внедрения зависимостей текущего пользователя через свойство `context.User`. Фактически это тот самый объект `ClaimsPrincipal`, созданный выше и сохраненный в куках. И когда приходит запрос к приложению, инфраструктура `ASP.NET Core` дешифрует и десериализует данные запроса и создает по ним объект `ClaimsPrincipal`, который хранится в свойстве `context.User`. Если используется аутентификация на основе куки (как в примере выше), то данные о пользователе будут извлекаться из аутентификационных кук. Если применяются `jwt`-токены, то данные берутся из полученного токена. Причем даже если аутентификационных кук или токена в запросе нет, то объект `ClaimsPrincipal` все равно будет создаваться.

Получив идентичность пользователя, мы можем получить различную информацию о нем. Например, проверить, аутентифицирован ли он, получить тип аутентификации, получить другую связанную с ним информацию.

Таким образом, при первом обращении к приложению, когда у нас не установлено никаких аутентификационных кук, пользователь из `context.User` не аутентифицирован:



Но после перехода по пути `"/login"` будут созданы объекты `ClaimsPrincipal` и `ClaimsIdentity` и по ним будут установлены аутентификационные куки. Соответственно при повторном переходе по пути `"/"` пользователь будет аутентифицирован:



Получение ClaimsPrincipal

Поскольку объект **HttpContext** доступен через механизм внедрения зависимостей в любой точке приложения, то мы можем через этот объект получить пользователя, как в примере выше. Однако, если нам нужно только свойство **User**, а не весь объект **HttpContext**, то мы можем также через механизм внедрения зависимостей получить сервис **ClaimsPrincipal**, который будет аналогичен свойству **context.User**:

```
1 app.Map("/", (ClaimsPrincipal claimsPrincipal) =>
2 {
3     var user = claimsPrincipal.Identity;
4     if (user is not null && user.IsAuthenticated)
5         return "Пользователь аутентифицирован";
6     else return "Пользователь не аутентифицирован";
7 });
```

5. ClaimPrincipal и объекты Claim

В прошлом учебном вопросе был рассмотрен объект `HttpContext.User`, который представляет класс `ClaimsPrincipal` и который хранит данные пользователя в свойстве `Identity` в виде объекта `ClaimsIdentity`. Но что представляют сами данные пользователя? Для хранения различных данных пользователя во фреймворке **ASP.NET Core** определяются объекты `claim`.

Объекты `claim` представляют некоторую информацию о пользователе, которую мы можем использовать для авторизации в приложении. Например, у пользователя может быть определенный возраст, город, страна проживания, любимая музыкальная группа и прочие признаки. И все эти признаки могут представлять отдельные объекты `claim`. И в зависимости от значения этих `claim` мы можем предоставлять пользователю доступ к тому или иному ресурсу. Таким образом, `claims` представляют более общий механизм авторизации нежели стандартные логины или роли, которые привязаны лишь к одному определенному признаку пользователя.

Каждый объект `claim` представляет класс `Claim` из пространства имен `System.Security.Claims`, который определяет следующие свойства:

Issuer: "издатель" или название системы, которая выдала данный `claim`.

Subject: возвращает информацию о пользователе в виде объекта `ClaimsIdentity`.

Type: возвращает тип объекта `claim`.

Value: возвращает значение объекта `claim`.

Создание Claim

Для создания объекта `Claim` определено множество конструкторов, но чаще всего применяется следующая версия конструктора:

```
1 public Claim(string type, string value)
```

В качестве первого параметра в конструктор передается тип `claima` – это некоторая строка, которая, как правило, описывает назначение `claima`. В качестве второго параметра передается значение этого `claima`. Например, простейшее создание `claima`:

```
1 var usernameClaim = new Claim(ClaimTypes.Name, "Tom");
```

В качестве типов можно использовать встроенные константы, типа `ClaimTypes.Name`, которая имеет значение `"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"` и которая обычно применяется для установки имени пользователя (то, что потом мы сможем получить через свойство `HttpContext.User.Identity.Name`). И в данном случае этот `claim` будет иметь значение `"Tom"`.

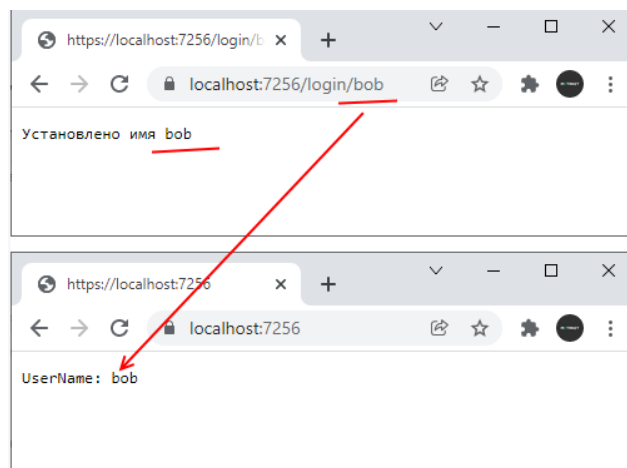
Все объекты **claim**, которые описывают пользователя, затем можно передать в виде коллекции в конструктор **ClaimsIdentity**:

```
1 var usernameClaim = new Claim(ClaimTypes.Name, "Tom");
2 var claims = new List<Claim> { usernameClaim };
3 var claimsIdentity = new ClaimsIdentity(claims, "Cookies");
```

Рассмотрим на небольшом примере:

```
1 using Microsoft.AspNetCore.Authentication.Cookies;
2 using System.Security.Claims;
3 using Microsoft.AspNetCore.Authentication;
4
5 var builder = WebApplication.CreateBuilder();
6
7 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
8     .AddCookie();
9
10 var app = builder.Build();
11
12 app.UseAuthentication();
13
14 app.MapGet("/login/{username}", async (string username, HttpContext context) =>
15 {
16     var claims = new List<Claim> { new (ClaimTypes.Name, username) };
17     var claimsIdentity = new ClaimsIdentity(claims, "Cookies");
18     var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
19     await context.SignInAsync(claimsPrincipal);
20     return $"Установлено имя {username}";
21 });
22 app.Map("/", (HttpContext context) =>
23 {
24     var user = context.User.Identity;
25     if (user is not null && user.IsAuthenticated)
26         return $"UserName: {user.Name}";
27     else return "Пользователь не аутентифицирован.";
28 });
29 app.MapGet("/logout", async (HttpContext context) =>
30 {
31     await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
32     return "Данные удалены";
33 });
34
35 app.Run();
```

В данном примере в конечной точке `app.MapGet("/login/{username}")` через параметр **"username"** получаем некоторое условное имя пользователя, создаем из него **Claim**, передаем его в **ClaimsIdentity**. В итоге после этого **claim** будет сохранен в аутентификационных куках. И когда в следующем запросе приложение получит эти аутентификационные куки, оно сможет извлечь эти данные и использовать их при создании объекта **ClaimsPrincipal**.



Следует отметить, что по умолчанию значение **claim** с типом **ClaimTypes.Name** (либо **ClaimsIdentity.DefaultNameClaimType**) передается свойству **HttpContext.User.Identity.Name**.

Управление объектами Claim

Для работы с объектами **Claim** в классе **ClaimsPrincipal** есть следующие **свойства и методы**:

Claims: свойство, которое возвращает набор ассоциированных с пользователем объектов **claim**.

FindAll(type) / FindAll(predicate): возвращает все объекты **claim**, которые соответствуют определенному типу или условию.

FindFirst(type) / FindFirst(predicate): возвращает первый объект **claim**, который соответствует определенному типу или условию.

HasClaim(type, value) / HasClaim(predicate): возвращает значение **true**, если пользователь имеет **claim** определенного типа с определенным значением.

IsInRole(name): возвращает значение **true**, если пользователь принадлежит роли с названием **name**.

С помощью объекта **ClaimsIdentity**, который возвращается свойством **User.Identity**, мы можем управлять объектами **claim** у текущего пользователя. В частности, класс **ClaimsIdentity** определяет следующие **свойства и методы**:

Claims: свойство, которое возвращает набор ассоциированных с пользователем объектов **claim**.

AddClaim(claim): добавляет для пользователя объект **claim**.

AddClaims(claims): добавляет набор объектов **claim**.

FindAll(type) / FindAll(predicate): возвращает все объекты **claim**, которые соответствуют определенному типу или условию.

FindFirst(type) / FindFirst(predicate): возвращает первый объект **claim**, который соответствует определенному типу или условию.

HasClaim(predicate): возвращает значение **true**, если пользователь имеет **claim**, соответствующий определенному условию.

RemoveClaim(claim): удаляет объект **claim**.

TryRemoveClaim(claim): удаляет объект **claim** и возвращает **true** при успешном удалении.

Например, определим у пользователя несколько объектов **claim**:

```
1 using Microsoft.AspNetCore.Authentication.Cookies;
2 using System.Security.Claims;
3 using Microsoft.AspNetCore.Authentication;
4
5 var builder = WebApplication.CreateBuilder();
6
7 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
8     .AddCookie();
9
10 var app = builder.Build();
11
12 app.UseAuthentication();
13
14 app.MapGet("/login", async (HttpContext context) =>
15 {
16     var username = "Tom";
17     var company = "Microsoft";
18     var phone = "+12345678901";
19
20     var claims = new List<Claim>
21     {
22         new Claim (ClaimTypes.Name, username),
23         new Claim ("company", company),
24         new Claim(ClaimTypes.MobilePhone, phone)
25     };
26     var claimsIdentity = new ClaimsIdentity(claims, "Cookies");
27     var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
28     await context.SignInAsync(claimsPrincipal);
29     return Results.Redirect("/");
30 });
31 app.Map("/", (HttpContext context) =>
32 {
33     // аналогично var username = context.User.Identity.Name
34     var username = context.User.FindFirst(ClaimTypes.Name);
35     var phone = context.User.FindFirst(ClaimTypes.MobilePhone);
36     var company = context.User.FindFirst("company");
37     return $"Name: {username?.Value}\nPhone: {phone?.Value}\nCompany: {company?.Value}";
38 });
39 app.MapGet("/logout", async (HttpContext context) =>
40 {
41     await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
42     return "Данные удалены";
43 });
44
45 app.Run();
```

В конечной точке `app.MapGet("/login")` в куки сохраняются объект **ClaimsPrincipal** с тремя объектами **claims**. Они представляют имя, компанию и телефон пользователя. Причем для добавления некоторых **claim** мы можем воспользоваться встроенными типами, как для имени или телефона пользователя:

```
1 new Claim(ClaimTypes.MobilePhone, phone)
```

Для других данных мы можем определить свои типы, просто передав какую-нибудь строку, как в случае с компанией:

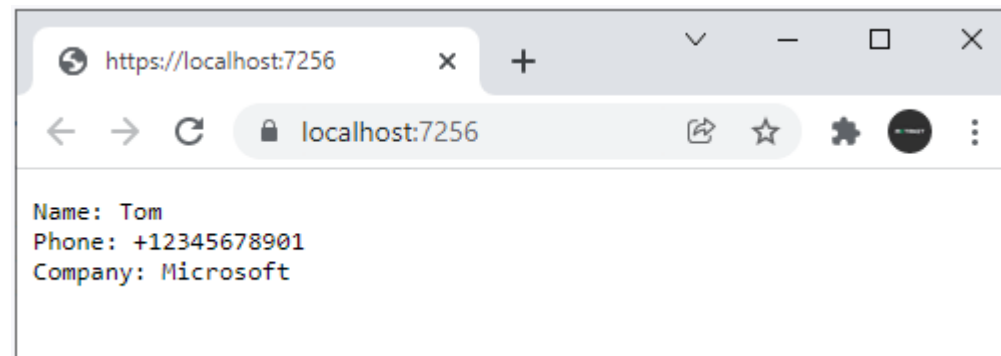
```
1 new Claim ("company", company),
```

Затем в приложении при обработке запроса мы можем получить эти данные. Как в примере выше в конечной точке `app.Map("/")`:

```
1 var phone = context.User.FindFirst(ClaimTypes.MobilePhone);  
2 var company = context.User.FindFirst("company");
```

В метод **FindFirst** передается тип **claim**. Стоит учитывать, что этот метод возвращает объект **Claim?**, то есть результатом метода может быть значение **null** (например, если пользователь не аутентифицирован или объект **claim** не установлен). Соответственно при обращении к значению **claim** необходимо проверять его на **null**.

Таким образом, после обращения по пути `"/login"` в куках будут сохранены данные пользователя:



Если мы динамически решим добавить новый **claim** или удалить существующий, то после изменения **claim** необходимо заново пересоздавать объект **ClaimsPrincipal** и перезаписывать аутентификационные куки или **jwt**-токен, где эти данные хранятся.

```
1 using Microsoft.AspNetCore.Authentication.Cookies;
2 using System.Security.Claims;
3 using Microsoft.AspNetCore.Authentication;
4
5 var builder = WebApplication.CreateBuilder();
6
7 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
8     .AddCookie();
9
10 var app = builder.Build();
11
12 app.UseAuthentication();
13 // Добавление возраста
14 app.MapGet("/addage", async (HttpContext context) =>
15 {
16     if(context.User.Identity is ClaimsIdentity claimsIdentity)
17     {
18         claimsIdentity.AddClaim(new Claim("age", "37"));
19         var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
20         await context.SignInAsync(claimsPrincipal);
21     }
22     return Results.Redirect("/");
23 });
24 // удаление телефона
25 app.MapGet("/removephone", async (HttpContext context) =>
26 {
27     if (context.User.Identity is ClaimsIdentity claimsIdentity)
28     {
29         var phoneClaim = claimsIdentity.FindFirst(ClaimTypes.MobilePhone);
30         // если claim успешно удален
31         if(claimsIdentity.TryRemoveClaim(phoneClaim))
32         {
33             var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
34             await context.SignInAsync(claimsPrincipal);
35         }
36     }
37     return Results.Redirect("/");
38 });
```

```

39 app.MapGet("/login", async (HttpContext context) =>
40 {
41     var username = "Tom";
42     var company = "Microsoft";
43     var phone = "+12345678901";
44
45     var claims = new List<Claim>
46     {
47         new Claim (ClaimTypes.Name, username),
48         new Claim ("company", company),
49         new Claim(ClaimTypes.MobilePhone, phone)
50     };
51     var claimsIdentity = new ClaimsIdentity(claims, "Cookies");
52     var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
53     await context.SignInAsync(claimsPrincipal);
54     return Results.Redirect("/");
55 });
56 app.Map("/", (HttpContext context) =>
57 {
58     var username = context.User.FindFirst(ClaimTypes.Name);
59     var phone = context.User.FindFirst(ClaimTypes.MobilePhone);
60     var company = context.User.FindFirst("company");
61     var age = context.User.FindFirst("age");
62     return $"Name: {username?.Value}\nPhone: {phone?.Value}\n" +
63         $"Company: {company?.Value}\nAge: {age?.Value}";
64 });
65 app.MapGet("/logout", async (HttpContext context) =>
66 {
67     await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
68     return "Данные удалены";
69 });
70
71 app.Run();

```

В данном случае конечная точка `app.MapGet("/removephone")` удаляет телефон, а `app.MapGet("/addage")` добавляет возраст в `claims`.

Если нам надо сохранить набор значений, то все они передаются по одному типу. Затем с помощью метода `FindAll()` можно получить список этих значений. Например, сохраним для пользователя набор иностранных языков:

```

1 using Microsoft.AspNetCore.Authentication.Cookies;
2 using System.Security.Claims;
3 using Microsoft.AspNetCore.Authentication;
4
5 var builder = WebApplication.CreateBuilder();
6
7 builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
8     .AddCookie();
9
10 var app = builder.Build();
11
12 app.UseAuthentication();
13
14 app.MapGet("/login", async (HttpContext context) =>
15 {
16     var claims = new List<Claim>
17     {
18         new Claim (ClaimTypes.Name, "Tom"),
19         new Claim ("languages", "English"),
20         new Claim ("languages", "German"),
21         new Claim ("languages", "Spanish")
22     };
23     var claimsIdentity = new ClaimsIdentity(claims, "Cookies");
24     var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
25     await context.SignInAsync(claimsPrincipal);
26     return Results.Redirect("/");
27 });
28 app.Map("/", (HttpContext context) =>
29 {
30     var username = context.User.FindFirst(ClaimTypes.Name);
31     var languages = context.User.FindAll("languages");
32     // объединяем список claims в строку
33     var languagesToString = "";
34     foreach (var l in languages)
35         languagesToString = $"{languagesToString} {l.Value}";
36     return $"Name: {username?.Value}\nLanguages: {languagesToString}";
37 });
38 app.MapGet("/logout", async (HttpContext context) =>
39 {
40     await context.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
41     return "Данные удалены";
42 });
43
44 app.Run();

```

