УДК 004

Приложение .Net Core Web API с использованием JWT-токенов для авторизации

Голубь Илья Сергеевич
Приамурский государственный университет имени Шолом-Алейхема
Магистрант

Аннотация
Цель работы была написать и на примере рассмотреть использование JWT-токенов для аутенфикации и авторизации в приложении, написанном на языке программирования C#. Разработка программы осуществлялась с помощью Visual Studio Community и VS Code. Из проведенного из исследования становится понятно, что JWT-токены упрощают аутенфиацию, а так же позволяют обмениваться данными между несколькими микросервисам при помощи одного и того же JSON объекта
Ключевые слова: Языки программирования, JWT, токен, C#, VS Community, VS Code, аутенфикация, авторизация

.Net Core Web API application using JWT tokens for authorization

Golub Ilya Sergeevich
Sholom Aleichem Priamursky State University
Undergraduate

Abstract
The purpose of the work was to write and consider using JWT tokens for authentication and authorization in an application written in the C # programming language as an example. The program was developed using Visual Studio Community and VS Code. From the study, it becomes clear that JWT tokens simplify authentication, as well as allow the exchange of data between multiple microservices using the same JSON object
Keywords: programming languages, JWT, token, C #, VS Community, VS Code, authentication, authorization, microservice

В данный момент нет единообразной системы построение авторизации и аутенфикации в приложении. Например, общие походы к аутенфикации и авторизации в .Net Core Web API отличается от .Net MVC. В частности, механизм авторизации, в Web API, преимущественно полагается на JWT-токен.

В работе M. B. Jones, B. Campbell и C. Mortimore говориться об спецификации draft-ietf-oauth-jwt-bearer-05. Эта спецификация определяет использование токена-носителя JSON Web Token (JWT) в качестве средства

для запроса токена доступа OAuth 2.0, а также для использования в качестве средства аутентификации клиента[3].

В статье P. Solapurkar рассматривается OAuth 2.0. Этот стандарт является делегированной средой авторизации, позволяющей осуществлять безопасную авторизацию для приложений, работающих на различных платформах. В сфере медицинских услуг OAuth позволяет пациенту (владельцу ресурса), обращающемуся за медицинской помощью в режиме реального времени, авторизовать автоматические ежемесячные платежи со своего банковского счета (сервера ресурсов), при этом пациенту не требуется предоставлять свои учетные данные в клинику (клиентское приложение). OAuth 2.0 достигает этого с помощью токенов, выданных сервером авторизации, который обеспечивает проверенный доступ к защищенному ресурсу. Для обеспечения безопасности токены доступа имеют срок действия и недолговечны. Таким образом, клиническое приложение может использовать токен обновления для получения нового токена доступа для ежемесячной оплаты наличными за оказание медицинских услуг в режиме реального времени. Токенам обновления необходимо защищенное хранилище, чтобы они не были утечками, поскольку любая злоумышленник может использовать их для получения нового доступа и обновления токенов. Поскольку OAuth 2.0 отбрасывает подписи и полностью полагается на SSL / TLS, он уязвим для фишинг-атак при доступе к совместимым API-интерфейсам. В этой статье мы разрабатываем подход, который объединяет веб-токен JSON (JWT) с OAuth 2.0 для запроса маркера доступа OAuth с сервера авторизации, когда клиент желает использовать предыдущую аутентификацию и авторизацию. Экспериментальная оценка подтверждает, что предложенная схема является практически эффективной, устраняет необходимость в безопасном хранилище, устраняя необходимость иметь или сохраняя маркер обновления, использует сигнатуру и предотвращает различные атаки безопасности, что очень желательно в службах здравоохранения, использующих облачную платформу IOT[4].

В работе M. B. Jones говориться о появлении нового набора протоколов открытой идентификации, который использует представления данных JSON и простые шаблоны связи на основе REST. Эти протоколы и форматы данных специально разработаны для простоты использования в браузерах и современных средах веб-разработки[1].

В статье O. Ethelbert, F. F. Moghaddam, P. Wieder, R. Yahyapour рассматривается как облачные вычисления изменяют компьютерную индустрию, основанную на таких основных понятиях, как виртуализация, вычислительная мощность, возможности подключения и эластичность для хранения и совместного использования ИТ-ресурсов через широкую сеть. Облачная индустрия стала ключевой технологией, которая раскрывает потенциал больших данных, Интернета вещей, мобильных и веб-приложений и других связанных технологий; но это также связано со своими проблемами, такими как управление, безопасность и конфиденциальность. Эта статья посвящена проблемам безопасности и конфиденциальности облачных

вычислений с особым акцентом на аутентификацию пользователей и управление доступом для облачных приложений SaaS. Предлагаемая модель использует платформу, которая использует JWT без сохранения состояния и безопасности для аутентификации клиентов и управления сеансами. Кроме того, авторизованный доступ к защищенным облачным ресурсам SaaS эффективно управляется. Соответственно, были введены компонент Gate Match Policy (PMG) и компонент Policy Activity Monitor (PAM). Кроме того, другие подкомпоненты, такие как Блок проверки политики (PVU) и БД прокси-сервера политики (PPDB), также были созданы для оптимизированной доставки услуг. Теоретический анализ предлагаемой модели изображает систему, которая является безопасной, легкой и легко масштабируемой для повышения безопасности облачных ресурсов и управления ими[2].

Целью данной работы является рассмотреть использование JWT-токенов для авторизации в приложении, написанном на .Net Core Web API с использованием JWT-токенов. Эта спецификация определяет использование токена-носителя JSON Web Token (JWT) в качестве средства для запрпоса токена доступа OAuth 2.0, а также для использования в качестве средства аутентификации клиента.

Методом исследования является создание приложения на языке C# и использующего JWT-токены для авторизации и аутенфикации. Средством исследования является среда разработки MS Visual studio 2019 и язык разработки C#.

Для начала необходимо рассмотреть, что такое JWT-токен. JSON Web Token (JWT) - представляет собой веб-стандарт, который обозначает способ передачи данных о пользователя в виде зашифрованного JSON объекта. Сам 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, Headera и Payload. Эта строка служит для верификации токена

Для использования JWT-токенов создадим новый проект ASP.NET Core по типу Empty.

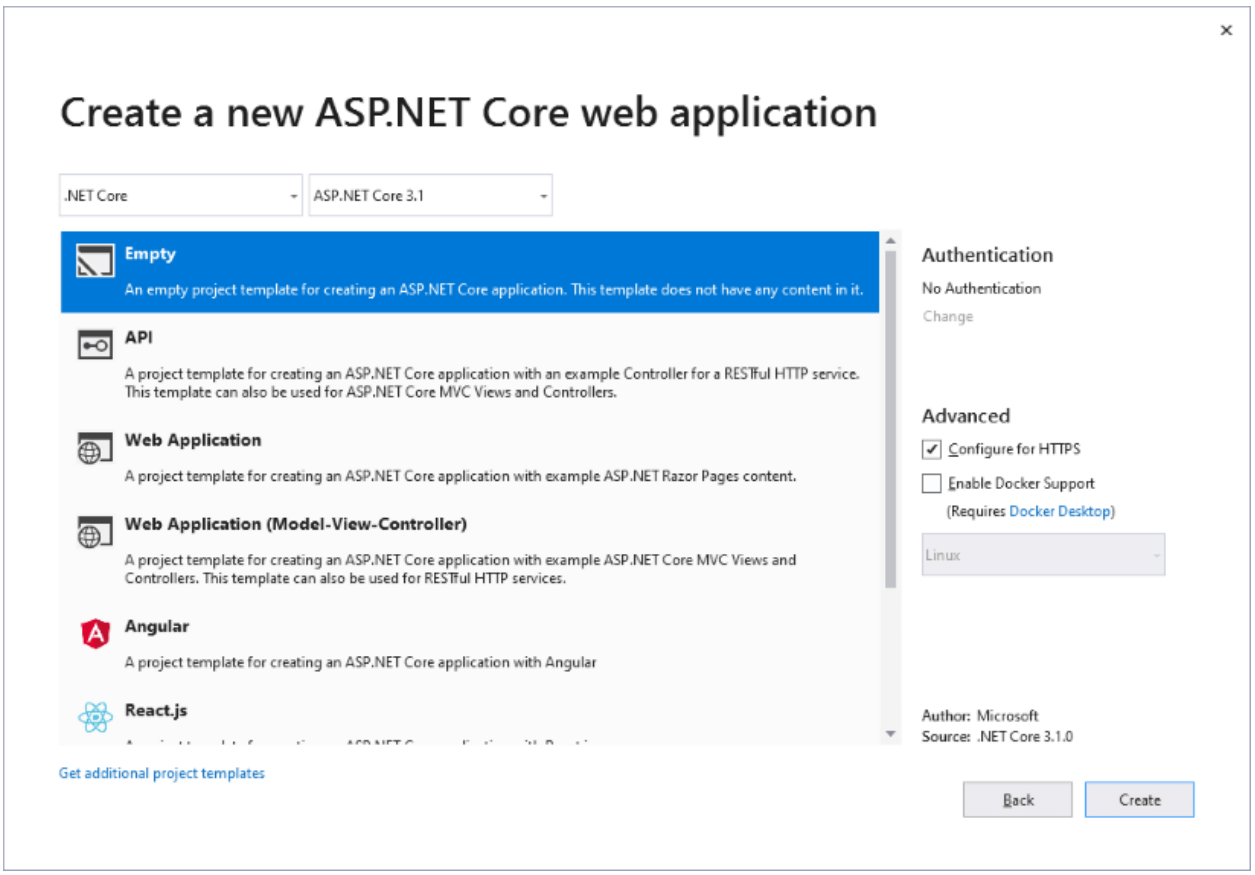

Рис. 1. Создание пустого приложения

Далее требуется создать в проекте папку Models, это папка в которой будут храниться наши классы. Теперь требуется добавить в данную папку класс Person.

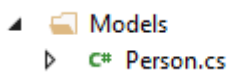

Рис. 2. Создание папки и класса

Так же нужно определить свойства класса Person для дальнейшего их использования.

```
public class Person
{
    public string Login { get; set; }
    public string Password { get; set; }
    public string Role { get; set; }
}
```

Рис. 3. Определение класса Person

Для работы с JWT-токенами потребуется установить через менеджер пакетов (Nuget) пакет Microsoft.AspNetCore.Authentication.JwtBearer. Так же нам необходимо определить специальный класс AuthOptions, описывающий ряд свойств, требующихся для генерации токенов

```
using Microsoft.IdentityModel.Tokens;
using System.Text;

namespace TokenApp
{
    public class AuthOptions
    {
        public const string ISSUER = "MyAuthServer"; // издатель токена
        public const string AUDIENCE = "MyAuthClient"; // потребитель токена
        const string KEY = "mysupersecret_secretkey!123";   // ключ для шифрации
        public const int LIFETIME = 1; // время жизни токена - 1 минута
        public static SymmetricSecurityKey GetSymmetricSecurityKey()
        {
            return new SymmetricSecurityKey(Encoding.ASCII.GetBytes(KEY));
        }
    }
}
```

Рис. 4.Определение класса AuthOptions

Константа ISSUER - создатель токена. Здесь можно определить любое значение. AUDIENCE – потребитель токена – можно хранить любую строку, в данном случае указан адрес данного приложения. KEY- ключ, применяющийся для создания токена. Для того, что бы функционал JWT-токенов работал необходимо его встроить в конвейер обработки запросов, осуществляется это с помощью компонента JwtBearerAuthenticationMiddleware.

Для этого изменим класс Startup следующим образом:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;

namespace TokenApp
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
                    .AddJwtBearer(options =>
                    {
                        options.RequireHttpsMetadata = false;
                        options.TokenValidationParameters = new TokenValidationParameters
                        {
                            // укзывает, будет ли валидироваться издатель при валидации токена
                            ValidateIssuer = true,
                            // строка, представляющая издателя
                            ValidIssuer = AuthOptions.ISSUER,

                            // будет ли валидироваться потребитель токена
                            ValidateAudience = true,
                            // установка потребителя токена
                            ValidAudience = AuthOptions.AUDIENCE,
                            // будет ли валидироваться время существования
                            ValidateLifetime = true,

                            // установка ключа безопасности
                            IssuerSigningKey = AuthOptions.GetSymmetricSecurityKey(),
                            // валидация ключа безопасности
                            ValidateIssuerSigningKey = true,
                        };
                    });
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseDeveloperExceptionPage();

            app.UseDefaultFiles();
            app.UseStaticFiles();

            app.UseRouting();

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

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapDefaultControllerRoute();
            });
        }
    }
}
```

Рис. 5. Класс Startup

Для того, что бы указать приложению на использование токенов для аутенфикации в методе ConfigureServices в вызов services.AddAuthentication передается значение JwtBearerDefaults.AuthenticationScheme. Далее, используя метод AddJwtBearer() добавляется конфигураци токена.

Для конфигурирования токена используется JwtBearerOptions, позволяющий при помощи свойств настроить работы с токеном. В данном случае были реализованные несколько свойств:

- RequireHttpsMetadata: -Определяет использование сертификата SSL, если равно false, то при отправке токена сертификат не используется. Этот вариант возможен, но не рекомандован к использованию, т.к. является не

безопасным. В реальном приложении все же лучше использовать передачу данных по протоколу https.

- TokenValidationParameters: параметры валидации токена - сложный объект, который определяет, как токен будет проверяться. Этот объект имеет множество свойств, позволяющие настроить различные аспекты валидации токена. Но есть несколько наиболее важных свойств: IssuerSigningKey - ключ безопасности, которым подписывается токен, и ValidateIssuerSigningKey - надо ли проверять ключ безопасности. Так же, можно установить ряд других свойств, таких как именно нужно проверять издателя и потребителя токена, сколько времени токен остается актуальным, можно установить название claims для ролей и логинов пользователя и т.д.

Теперь авторизацию при помощи токенов можно использовать. Однако в прокте пока отсутствует генерация токенов. По умолчанию в ASP.NET Core не предусмотрены встроенные возможности создания токена. И в данном случае можно воспользоваться сторонними решениями (например, IdentityServer или OpenIdDict) или создать свой механизм. Выберем второй способ.

Создадим новую папку Controllers и добавим в нее контроллер AccountController.

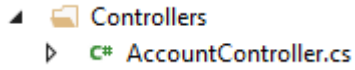

Рис. 6. Папка и контроллер

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using TokenApp.Models; // класс Person

namespace TokenApp.Controllers
{
    public class AccountController : Controller
    {
        // тестовые данные вместо использования базы данных
        private List<Person> people = new List<Person>
        {
            new Person {Login="admin@gmail.com", Password="12345", Role = "admin" },
            new Person { Login="qwerty@gmail.com", Password="55555", Role = "user" }
        };

        [HttpPost("/token")]
        public IActionResult Token(string username, string password)
        {
            var identity = GetIdentity(username, password);
            if (identity == null)
            {
                return BadRequest(new { errorText = "Invalid username or password." });
            }

            var now = DateTime.UtcNow;
            // создаем JWT-токен
            var jwt = new JwtSecurityToken(
                    issuer: AuthOptions.ISSUER,
                    audience: AuthOptions.AUDIENCE,
                    notBefore: now,
                    claims: identity.Claims,
                    expires: now.Add(TimeSpan.FromMinutes(AuthOptions.LIFETIME)),
                    signingCredentials: new SigningCredentials(AuthOptions.GetSymmetricSecurityKey(), SecurityAlgorithms.HmacSha2
            var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);

            var response = new
            {
                access_token = encodedJwt,
                username = identity.Name
            };

            return Json(response);
        }

        private ClaimsIdentity GetIdentity(string username, string password)
        {
            Person person = people.FirstOrDefault(x => x.Login == username && x.Password == password);
            if (person != null)
            {
                var claims = new List<Claim>
                {
                    new Claim(ClaimsIdentity.DefaultNameClaimType, person.Login),
                    new Claim(ClaimsIdentity.DefaultRoleClaimType, person.Role)
                };
                ClaimsIdentity claimsIdentity =
                new ClaimsIdentity(claims, "Token", ClaimsIdentity.DefaultNameClaimType,
                    ClaimsIdentity.DefaultRoleClaimType);
                return claimsIdentity;
            }

            // если пользователя не найдено
            return null;
        }
    }
}
```

Рис. 7. AccountController

Данные пользователей определены в виде обычного списка. Для поиска пользователя в данном списке по логину и паролю определен метод GetIdentity(), возвращающий объект ClaimsIdentity. Принцип создания ClaimsIdentity: создается набор объектов Claim, включающихразличные данные о пользователе, например, логин, роль и т.д. Для обработки запроса в контроллере создан метод Token, который сопоставлен с маршрутом "/token".

Этот метод обрабатывает обращения через POST запросы и в параметры принимает логин и пароль пользователя. Сам токен это объект JwtSecurityToken, для создания которого используются все те же константы и ключ безопасности, которые определены в классе AuthOptions и которые использовались в классе Startup для настройки JwtBearerAuthenticationMiddleware. Важно, чтобы эти значения совпадали. С помощью параметра claims: identity.Claims в токен добавляются набор объектов Claim, которые содержат информацию о логине и роли пользователя. Далее посредством метода JwtSecurityTokenHandler().WriteToken(jwt) создается Json-представление токена. И в конце он сериализуется и отправляет клиенту с помощью метода Json(). Таким образом генерируется токен. Для тестирования токена создадим простенький контроллер ValuesController.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace TokenApp.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ValuesController : Controller
    {
        [Authorize]
        [Route("getlogin")]
        public IActionResult GetLogin()
        {
            return Ok($"Ваш логин: {User.Identity.Name}");
        }

        [Authorize(Roles = "admin")]
        [Route("getrole")]
        public IActionResult GetRole()
        {
            return Ok("Ваша роль: администратор");
```

Рис. 8. ValuesController

Так же добавим папку для статических файлов wwwroot, а в нее - новый файл index.html:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using TokenApp.Models; // класс Person

namespace TokenApp.Controllers
{
    public class AccountController : Controller
    {
        // тестовые данные вместо использования базы данных
        private List<Person> people = new List<Person>
        {
            new Person {Login="admin@gmail.com", Password="12345", Role = "admin" },
            new Person { Login="qwerty@gmail.com", Password="55555", Role = "user" }
        };

        [HttpPost("/token")]
        public IActionResult Token(string username, string password)
        {
            var identity = GetIdentity(username, password);
            if (identity == null)
            {
                return BadRequest(new { errorText = "Invalid username or password." });
            }

            var now = DateTime.UtcNow;
            // создаем JWT-токен
            var jwt = new JwtSecurityToken(
                    issuer: AuthOptions.ISSUER,
                    audience: AuthOptions.AUDIENCE,
                    notBefore: now,
                    claims: identity.Claims,
                    expires: now.Add(TimeSpan.FromMinutes(AuthOptions.LIFETIME)),
                    signingCredentials: new SigningCredentials(AuthOptions.GetSymmetricSecurityKey(), SecurityAlgorithms.HmacSha2
            var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);

            var response = new
            {
                access_token = encodedJwt,
                username = identity.Name
            };

            return Json(response);
        }

        private ClaimsIdentity GetIdentity(string username, string password)
        {
            Person person = people.FirstOrDefault(x => x.Login == username && x.Password == password);
            if (person != null)
            {
                var claims = new List<Claim>
                {
                    new Claim(ClaimsIdentity.DefaultNameClaimType, person.Login),
                    new Claim(ClaimsIdentity.DefaultRoleClaimType, person.Role)
                };
                ClaimsIdentity claimsIdentity =
                new ClaimsIdentity(claims, "Token", ClaimsIdentity.DefaultNameClaimType,
                    ClaimsIdentity.DefaultRoleClaimType);
                return claimsIdentity;
            }

            // если пользователя не найдено
            return null;
        }
    }
}
```

Рис. 9. index.html

Первый блок – блок вывода на страницу информации 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 "/token". Поскольку за обработку запросов по этому маршруту отвечает метод Token контроллера AccountController, по результатам работы которого будет формироваться токен. Ответом сервера в случае удачной аутентификации будет примерно следующий объект.

```
{
    access_token : "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93
                    cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjoicXdlcnR5IiwiaHR0cDovL3NjaGVtYXMub
                    Wljcm9zb2Z0LmNvbS93cy8yMDA4LzA2L2lkZW50aXR5L2NsYWltcy9yb2xlIjoidXNlciIsIm5iZi
                    I6MTQ4MTYzOTMxMSwiZXhwIjoxNDgxNjM5MzcxLCJpc3MiOiJNeUF1dGhTZXJ2ZXIiLCJhdWQiOiJ
                    odHRwOi8vbG9jYWxob3N0OjUxODg0LyJ9.dQJF6pALUZW3wGBANy_tCwk5_NR0TVBwgnxRbblp5Ho",
    username: "qwerty@gmail.com"
}
```

Рис. 9. JSON объект ответа сервера

Параметр access_token как раз и будет представлять токен доступа. Также в объекте передается дополнительная информация об имени пользователя. Для того чтобы в коде js данный токен в дальнейшем был доступен, то он будет храниться в хранилище sessionStorage. Последние два блока предназначены для отправки запросов к методам ValuesController. Чтобы отправить токен в запросе, нам нужно настроить в запросе заголовок Authorization.

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

Рис. 10. Authorization

В итоге весь проект будет выглядеть следующим образом.

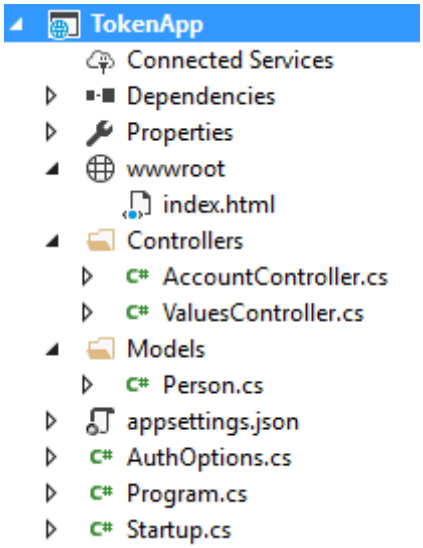

Рис. 11. Solution

По ранее сохраненному ключу получен, из хранилища приложения, токен и формирован заголовок. Далее, после получения токена, требуется осуществить запрос к методам контроллера ValuesController.

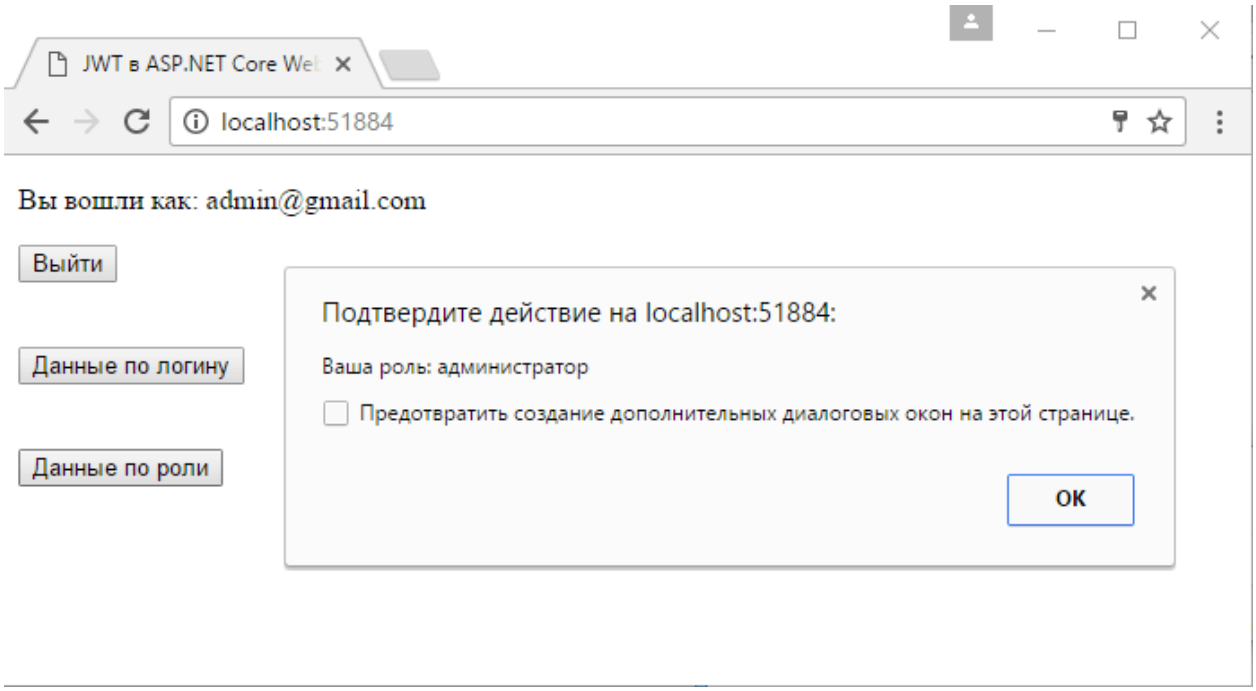

Рис. 12. Страница программы

Так же, если обратиться к тем же методам без токена или с токеном с истекшим сроком, то вернется ошибка 401 (Unauthorized).

Подводя итог, мы получаем готовое не большое приложение, которое использует авторизацию и аутенфикацию с помощью JWT-токенов. Данный способ упрощает общение нескольких микросервисов между собой так, как использует JWT-токен для авторизации пользователей в отдельных микросервисах.

Библиографический список

1. Jones M. B. The emerging JSON-based identity protocol suite //W3C workshop on identity in the browser. – 2017. – С. 1-3.
2. Ethelbert O. et al. A JSON token-based authentication and access management schema for Cloud SaaS applications //2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud). IEEE, 2017. С. 47-53.
3. Jones M. B. et al. JSON Web Token (JWT) profile for OAuth 2.0 client authentication and authorization Grants. – 2016.
4. Solapurkar P. Building secure healthcare services using OAuth 2.0 and JSON web token in IOT cloud scenario //2016 2nd International Conference on Contemporary Computing and Informatics (IC3I). – IEEE, 2016. С. 99-104.