Implement OAuth JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1 - Part 3

February 16, 2015 By Taiseer Joudeh – 124 Comments

This is the third part of Building Simple Membership system using ASP.NET Identity 2.1, ASP.NET Web API 2.2 and AngularJS. The topics we'll cover are:

- Configure ASP.NET Identity with ASP.NET Web API (Accounts Management) – Part 1.
- ASP.NET Identity 2.1 Accounts Confirmation, and Password/User Policy Configuration – Part 2.
- Implement JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1 – (This Post)
- ASP.NET Identity 2.1 Roles Based Authorization with ASP.NET Web API – Part 4
- ASP.NET Web API Claims Authorization with ASP.NET Identity 2.1 – Part 5
- *AngularJS Authentication and Authorization with ASP.NET Web API and Identity 2.1 – Part 6*

The source code for this tutorial is available on GitHub.

Implement JSON Web Tokens Authentication in ASP.NET Web API and

and Identity 2.1

Currently our API doesn't support authentication and authorization, all the requests we receive to any end point are done anonymously, In this post we'll configure our API which will act as our Authorization Server and Resource Server on the same time to issue JSON Web Tokens for authenticated users and those users will present this JWT to the protected end points in order to access it and process the request.

I will use step by step approach as usual to implement this, but I highly recommend you to read the post JSON Web Token in ASP.NET Web API 2 before completing this one; where I cover deeply what is JSON Web Tokens, the benefits of using JWT over default access tokens, and how they can be used to decouple Authorization server from Resource server. In this tutorial and for the sake of keeping it simple; both OAuth 2.0 roles (Authorization Server and Recourse Server) will live in the same API.

Step 1: Implement OAuth 2.0 Resource Owner Password Credential Flow

We are going to build an API which will be consumed by a trusted client (AngularJS front-end) so we only interested in implementing a single OAuth 2.0 flow where the registered user will present username and

password to a specific end point, and the API will validate those credentials, and if all is valid it will return a JWT for the user where the client application used by the user should store it securely and locally in order to present this JWT with each request to any protected end point.

present this JWT with each request to any protected end point.

The nice thing about this JWT that it is a self contained token which contains all user claims and roles inside it, so there is no need to do any extra DB queries to fetch those values for the authenticated user. This JWT token will be configured to expire after 1 day of its issue date, so the user is requested to provide credentials again in order to obtain new JWT token.

If you are interested to know how to implement sliding expiration tokens and how you can keep the user logged in; I recommend you to read my other post Enable OAuth Refresh Tokens in AngularJS App which covers this deeply, but adds more complexity to the solution. To keep this tutorial simple we'll not add refresh tokens here but you can refer to the post and implement it.

To implement the Resource Owner Password Credential flow; we need to add new folder named "Providers" then add a new class named "CustomOAuthProvider", after you add then paste the code below:

```
public class CustomOAuthProvider : OAuthAuthorizationServerProvider
{

    public override Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)
    {
        context.Validated();
        return Task.FromResult<object>(null);
    }

    public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
    {

        var allowedOrigin = "*";

        context.OwinContext.Response.Headers.Add("Access-Control-Allow-Origin", new[] { allowedOrigin });

        var userManager = context.OwinContext.GetUserManager<ApplicationUserManager>();

        ApplicationUser user = await userManager.FindAsync(context.UserName, context.Password);

        if (user == null)
        {
            context.SetError("invalid_grant", "The user name or password is incorrect.");
```

```
                return;
            }

            if (!user.EmailConfirmed)
            {
                context.SetError("invalid_grant", "User did not confirm email.");
                return;
            }

            ClaimsIdentity oAuthIdentity = await user.GenerateUserIdentityAsync(userManager, "JWT");

            var ticket = new AuthenticationTicket(oAuthIdentity, null);

            context.Validated(ticket);

        }
    }
```

This class inherits from class "OAuthAuthorizationServerProvider" and overrides the below two methods:

- As you notice the "**ValidateClientAuthentication**" is empty, we are considering the request valid always, because in our implementation our client (AngularJS front-end) is trusted client and we do not need to validate it.
- The method "**GrantResourceOwnerCredentials**" is responsible for receiving the username and password from the request and validate them against our ASP.NET 2.1 Identity system, if the credentials are valid and the email is confirmed we are building an identity for the logged in user, this identity will contain all the roles and claims for the authenticated user, until now we didn't cover roles and claims part of the tutorial, but for the mean time you can consider all users registered in our system without any roles or claims mapped to them.
- The method "GenerateUserIdentityAsync" is not implemented yet, we'll add this helper method in the next step. This method will be responsible to fetch the authenticated user identity from the database and returns an object of type "ClaimsIdentity".
- Lastly we are creating an Authentication ticket which contains the identity for the authenticated user, and when we call "context.Validated(ticket)" this will transfer this identity to an OAuth 2.0 bearer access token.

Step 2: Add method "GenerateUserIdentityAsync" to "ApplicationUser" class

Now we'll add the helper method which will be responsible to get the authenticated user identity (all roles and claims mapped to the user). The "UserManager" class contains a method named "CreateIdentityAsync" to do this task, it will basically query the DB and get all the roles and claims for this user, to implement this open class "ApplicationUser" and paste the code below:

```
//Rest of code is removed for brevity
public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser> manager, string authentication
{
    var userIdentity = await manager.CreateIdentityAsync(this, authenticationType);
    // Add custom user claims here
    return userIdentity;
}
```

Step 3: Issue JSON Web Tokens instead of Default Access Tokens

Now we want to configure our API to issue JWT tokens instead of default access tokens, to understand what is JWT and why it is better to use it, you can refer back to this post.

First thing we need to installed 2 NueGet packages as the below:

```
Install-package System.IdentityModel.Tokens.Jwt -Version 4.0.1
Install-package Thinktecture.IdentityModel.Core -Version 1.3.0
```

There is no direct support for issuing JWT in ASP.NET Web API, so in order to start issuing JWTs we need to implement this manually by implementing the interface "ISecureDataFormat" and implement the method "Protect".

To implement this add new file named "CustomJwtFormat" under folder "Providers" and paste the code below:

```
    public class CustomJwtFormat : ISecureDataFormat<AuthenticationTicket>
    {

        private readonly string _issuer = string.Empty;

        public CustomJwtFormat(string issuer)
        {
            _issuer = issuer;
        }

        public string Protect(AuthenticationTicket data)
        {
            if (data == null)
            {
                throw new ArgumentNullException("data");
            }

            string audienceId = ConfigurationManager.AppSettings["as:AudienceId"];

            string symmetricKeyAsBase64 = ConfigurationManager.AppSettings["as:AudienceSecret"];

            var keyByteArray = TextEncodings.Base64Url.Decode(symmetricKeyAsBase64);

            var signingKey = new HmacSigningCredentials(keyByteArray);

            var issued = data.Properties.IssuedUtc;

            var expires = data.Properties.ExpiresUtc;

            var token = new JwtSecurityToken(_issuer, audienceId, data.Identity.Claims, issued.Value.UtcDateTime, expi

            var handler = new JwtSecurityTokenHandler();


            var jwt = handler.WriteToken(token);

            return jwt;
        }

        public AuthenticationTicket Unprotect(string protectedText)
        {
            throw new NotImplementedException();
        }
    }
```

What we've implemented in this class is the following:

- The class "CustomJwtFormat" implements the interface "ISecureDataFormat<AuthenticationTicket>", the JWT generation will take place inside method "Protect".
- The constructor of this class accepts the "Issuer" of this JWT which will be our API. This API acts as Authorization and Resource Server on the same time, this can be string or URI, in our case we'll fix it to URI.
- Inside "Protect" method we are doing the following:
  - As we stated before, this API serves as Resource and Authorization Server at the same time, so we are fixing the Audience Id and Audience Secret (Resource Server) in web.config file, this Audience Id and Secret will be used for HMAC265 and hash the JWT token, I've used this implementation to generate the Audience Id and Secret.
  - Do not forget to add 2 new keys "as:AudienceId" and "as:AudienceSecret" to the web.config AppSettings section.
  - Then we prepare the raw data for the JSON Web Token which will be issued to the requester by providing the issuer, audience, user claims, issue date, expiry date, and the signing key which will sign (hash) the JWT payload.
  - Lastly we serialize the JSON Web Token to a string and return it to the requester.
- By doing this, the requester for an OAuth 2.0 access token from our API will receive a signed token which contains claims for an authenticated Resource Owner (User) and this access token is intended to certain (Audience) as well.

Step 4: Add Support for OAuth 2.0 JWT Generation

Till this moment we didn't configure our API to use OAuth 2.0 Authentication workflow, to do so open class "Startup" and add new method named "ConfigureOAuthTokenGeneration" as the below:

```
        private void ConfigureOAuthTokenGeneration(IAppBuilder app)
        {
            // Configure the db context and user manager to use a single instance per request
            app.CreatePerOwinContext(ApplicationDbContext.Create);
            app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);

            OAuthAuthorizationServerOptions OAuthServerOptions = new OAuthAuthorizationServerOptions()
            {
                //For Dev enviroment only (on production should be AllowInsecureHttp = false)
                AllowInsecureHttp = true,
                TokenEndpointPath = new PathString("/oauth/token"),
                AccessTokenExpireTimeSpan = TimeSpan.FromDays(1),
                Provider = new CustomOAuthProvider(),
                AccessTokenFormat = new CustomJwtFormat("http://localhost:59822")
            };

            // OAuth 2.0 Bearer Access Token Generation
            app.UseOAuthAuthorizationServer(OAuthServerOptions);
        }
```

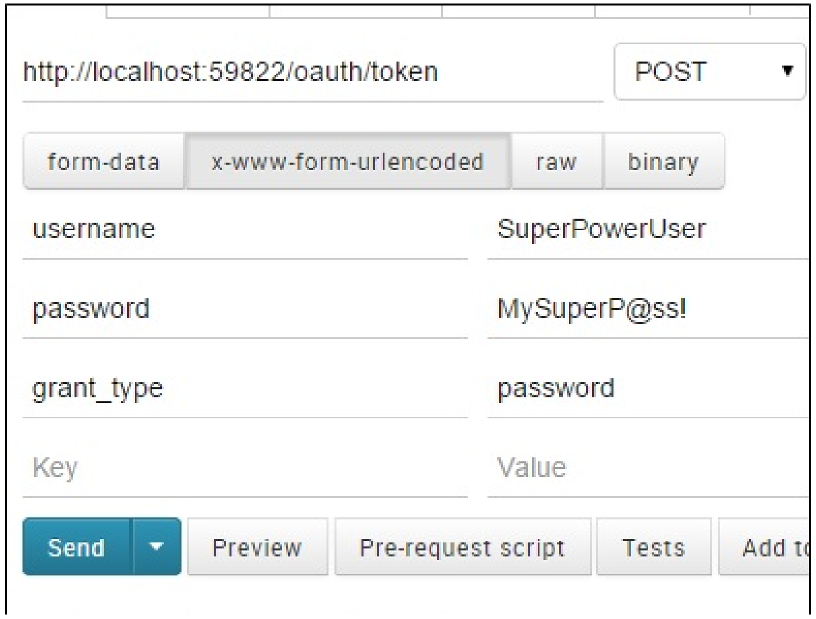
What we've implemented here is the following:

- The path for generating JWT will be as :"http://localhost:59822/oauth/token".
- We've specified the expiry for token to be 1 day.
- We've specified the implementation on how to validate the Resource owner user credential in a custom class named "CustomOAuthProvider".
- We've specified the implementation on how to generate the access token using JWT formats, this custom class named "CustomJwtFormat" will be responsible for generating JWT instead of default access token using DPAPI, note that both format will use **Bearer** scheme.

Do not forget to call the new method "ConfigureOAuthTokenGeneration" in the Startup "Configuration" as the class below:

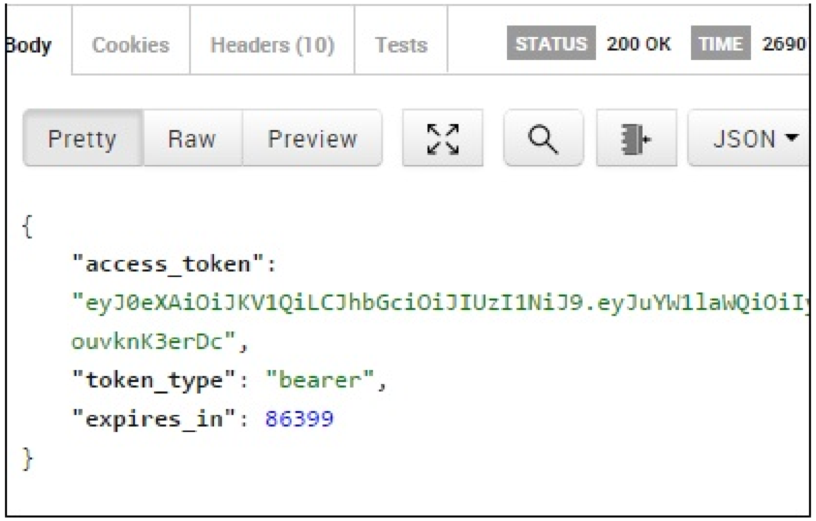
```
public void Configuration(IAppBuilder app)
{
    HttpConfiguration httpConfig = new HttpConfiguration();

    ConfigureOAuthTokenGeneration(app);

    //Rest of code is removed for brevity

}
```

Our API currently is ready to start issuing JWT access token, so test this out we can issue HTTP POST request as the image below, and we should receive a valid JWT token for the next 24 hours and accepted only by our API.


http://localhost:59822/oauth/token POST

form-data [✓ x-www-form-urlencoded] raw binary

username	SuperPowerUser
password	MySuperP@ss!
grant_type	password
Key	Value



```
{
    "access_token":
    "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJuYW1laWQiOiI
    ouvknK3erDc",
    "token_type": "bearer",
    "expires_in": 86399
}
```

Step 5: Protect the existing end points with [Authorize] Attribute

Now we'll visit all the end points we have created earlier in previous posts in the "AccountsController" class,

and attribute the end points which need to be protected (only authenticated user with valid JWT access token can access it) with the **[Authorize]** attribute as the below:

– **GetUsers, GetUser, GetUserByName, and DeleteUser** end points should be accessed by users enrolled in Role "Admin". Roles Authorization is not implemented yet and for now we will only allow any authentication user to access it, the code change will be as simple as the below:

```
[Authorize]
[Route("users")]
public IHttpActionResult GetUsers()
{}

[Authorize]
[Route("user/{id:guid}", Name = "GetUserById")]
public async Task<IHttpActionResult> GetUser(string Id)
{}

```

```

[Authorize]
[Route("user/{username}")]
public async Task<IHttpActionResult> GetUserByName(string username)
{
}

[Authorize]
[Route("user/{id:guid}")]
public async Task<IHttpActionResult> DeleteUser(string id)
{
}
```

- CreateUser and ConfirmEmail endpoints should be accessed anonymously always, so we need to attribute it with **[AllowAnonymous]** as the below:

```
[AllowAnonymous]
[Route("create")]
public async Task<IHttpActionResult> CreateUser(CreateUserBindingModel createUserModel)
{
}

[AllowAnonymous]
[HttpGet]
[Route("ConfirmEmail", Name = "ConfirmEmailRoute")]
public async Task<IHttpActionResult> ConfirmEmail(string userId = "", string code = "")
{
}
```

- ChangePassword endpoint should be accessed by the authenticated user only, so we'll attribute it with **[Authorize]** attribute as the below:

```
[Authorize]
[Route("ChangePassword")]
public async Task<IHttpActionResult> ChangePassword(ChangePasswordBindingModel model)
{
}
```

Step 6: Consume JSON Web Tokens

Now if we tried to obtain an access token by sending a request to the end point "oauth/token" then try to access one of the protected end points we'll receive 401 Unauthorized status, the reason for this that our API doesn't understand those JWT tokens issued by our API yet, to fix this we need to the following:

Install the below NuGet package:

```
Install-Package Microsoft.Owin.Security.Jwt -Version 3.0.0
```

The package "Microsoft.Owin.Security.Jwt" is responsible for protecting the Resource server resources using JWT, it only validate and de-serialize JWT tokens.

Now back to our "Startup" class, we need to add the below method "ConfigureOAuthTokenConsumption" as

the below:

```
        private void ConfigureOAuthTokenConsumption(IAppBuilder app) {

            var issuer = "http://localhost:59822";
            string audienceId = ConfigurationManager.AppSettings["as:AudienceId"];
            byte[] audienceSecret = TextEncodings.Base64Url.Decode(ConfigurationManager.AppSettings["as:AudienceSecret

            // Api controllers with an [Authorize] attribute will be validated with JWT
            app.UseJwtBearerAuthentication(
                new JwtBearerAuthenticationOptions
                {
                    AuthenticationMode = AuthenticationMode.Active,
                    AllowedAudiences = new[] { audienceId },
                    IssuerSecurityTokenProviders = new IIssuerSecurityTokenProvider[]
                    {
                        new SymmetricKeyIssuerSecurityTokenProvider(issuer, audienceSecret)
                    }
                });
        }
```

This step will configure our API to trust tokens issued by our Authorization server only, in our case the Authorization and Resource Server are the same server (http://localhost:59822), notice how we are providing the values for audience, and the audience secret we used to generate and issue the JSON Web providing the values for audience, and the audience secret we used to generate and issue the JSON Web Token in step3.

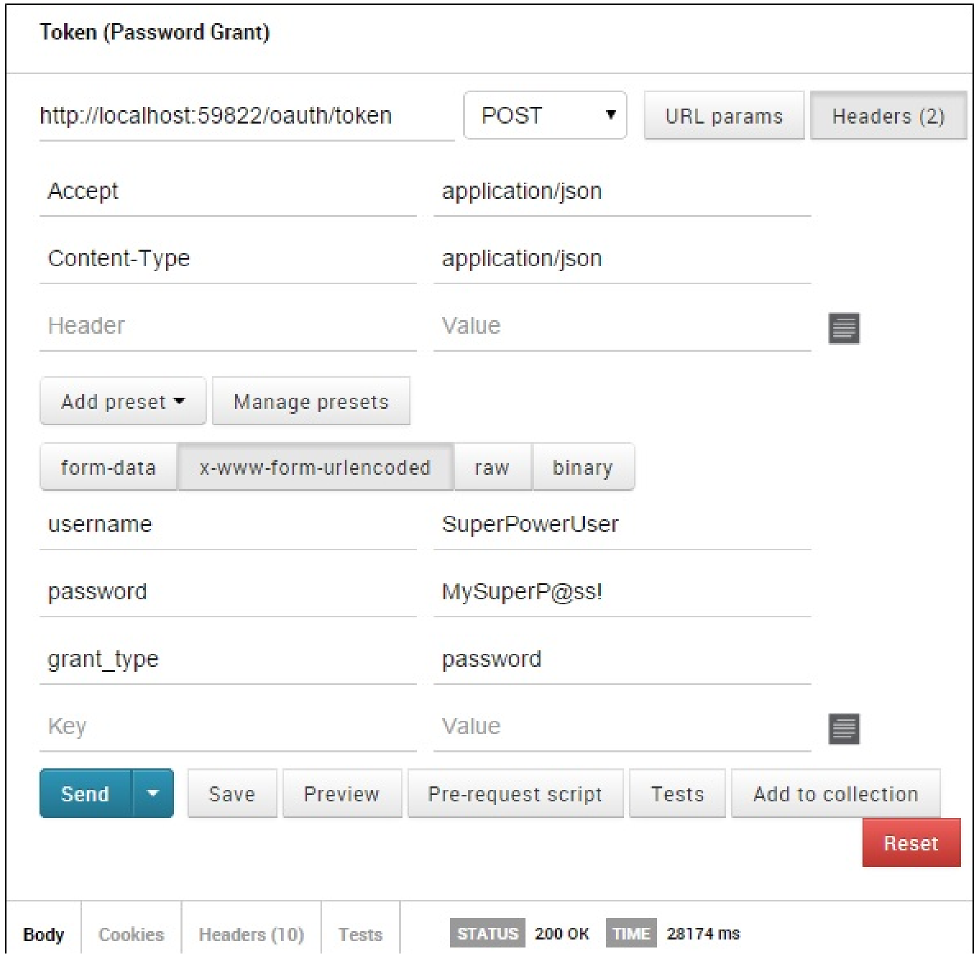
By providing those values to the “JwtBearerAuthentication” middleware, our API will be able to consume only JWT tokens issued by our trusted Authorization server, any other JWT tokens from any other Authorization server will be rejected.

Lastly we need to call the method “ConfigureOAuthTokenConsumption” in the “Configuration” method as the below:

```
public void Configuration(IAppBuilder app)
    {
        HttpConfiguration httpConfig = new HttpConfiguration();

        ConfigureOAuthTokenGeneration(app);

        ConfigureOAuthTokenConsumption(app);

        //Rest of code is here

    }
```

Step 7: Final Testing

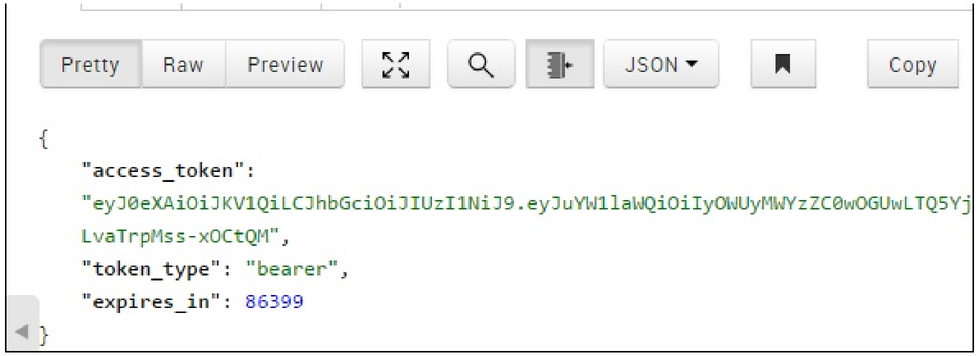
All the pieces should be in place now, to test this we will obtain JWT access token for the user “SuperPowerUser” by issuing POST request to the end point “oauth/token”



Accept	application/json
Content-Type	application/json
Header	Value

username	SuperPowerUser
password	MySuperP@ss!
grant_type	password
Key	Value

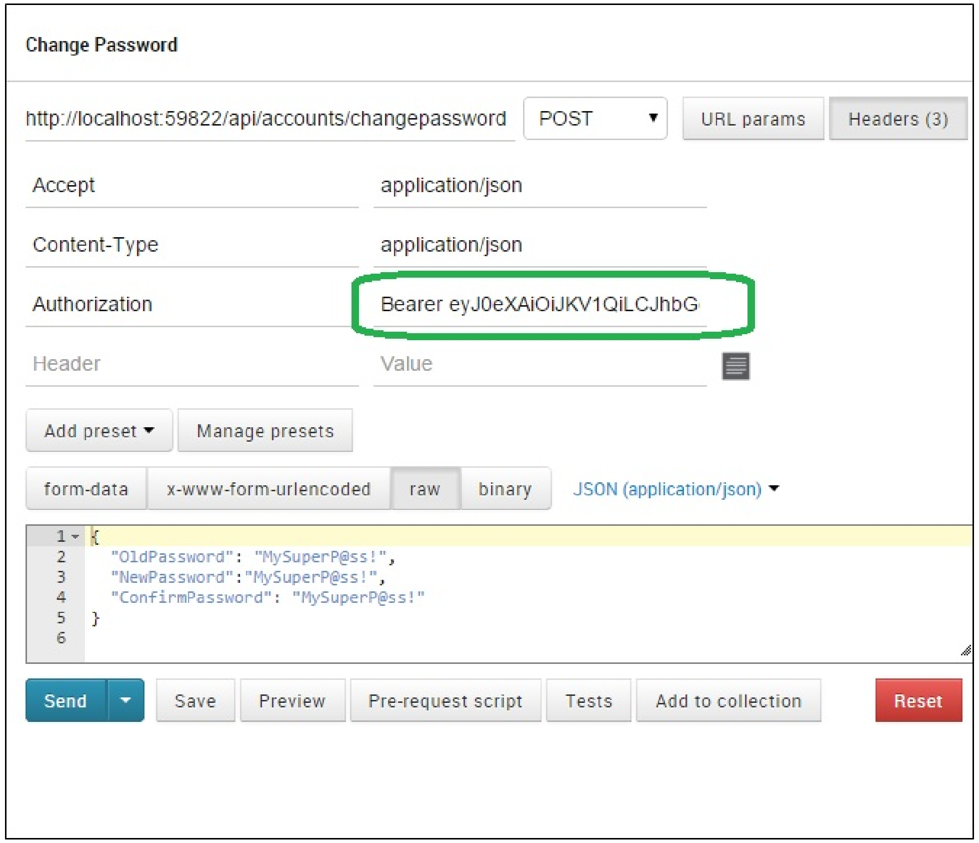
STATUS 200 OK TIME 28174 ms



```
{
    "access_token":
    "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJuYW1laWQiOiIyOWUyMWYzZC0wOGUwLTQ5Yj
    LvaTrpMss-xOCtQM",
    "token_type": "bearer",
    "expires_in": 86399
}
```

Then we will use the JWT received to access protected end point such as “ChangePassword”, if you remember once we added this end point, we were not able to test it directly because it was anonymous and inside its implementation we were calling the method “User.Identity.GetUserId()”. This method will return nothing for anonymous user, but after we’ve added the [Authorize] attribute, any user needs to access this end point should be authenticated and has a valid JWT.

To test this out we will issue POST request to the end point “/accounts/ChangePassword”as the image below, notice he we are sitting the **Authorization header** using **Bearer** scheme setting its value to the JWT we received for the user “SuperPwoerUser”. If all is valid we will receive 200 OK status and the user password should be updated.



http://localhost:59822/api/accounts/changepassword POST

Accept	application/json
Content-Type	application/json
Authorization	Bearer eyJ0eXAiOiJKV1QiLCJhbG
Header	Value

JSON (application/json)

```
{
  "OldPassword": "MySuperP@ss!",
  "NewPassword":"MySuperP@ss!",
  "ConfirmPassword": "MySuperP@ss!"
}
```

The source code for this tutorial is available on GitHub.

In the next post we'll see how we'll implement Roles Based Authorization in our Identity service.

Follow me on Twitter @tjoudeh

References

- Understanding OWIN/Katana Authentication/Authorization Part I: Concepts by John Atten
- Featured Image Source

Related Posts

ASP.NET Web API Claims Authorization with ASP.NET Identity 2.1 – Part 5
ASP.NET Identity 2.1 Roles Based Authorization with ASP.NET Web API – Part 4
AngularJS Authentication Using Azure Active Directory Authentication Library (ADAL)
JSON Web Token in ASP.NET Web API 2 using Owin
Decouple OWIN Authorization Server from Resource Server

Filed Under: ASP.NET, ASP.NET Identity, ASP.Net Web API, CodeProject, Web API Security, Web API Tutorial
Tagged With: Authentication, Authorization, JSON Web Tokens, JWT, OAuth

Comments

alexismeillandAlexis says
August 23, 2015 at 8:08 pm

Hi,

I have followed your tutorial. It's awesome. But I have an issue.

When I try to Change the password. I pass the bearer token.

And I have a notimplemented exception on JwtFormat class in provider folder.

public AuthenticationTicket Unprotect(string protectedText)

Why is it going in this method?

How can I fix it?

Thanks,

Reply

Taiseer Joudeh says
August 24, 2015 at 10:53 pm

Hi Alex,
I'm not sure if you implemented something incorrectly in this class, but are you receiving this exception for all protected endpoints or just this one?

Reply

alex says
August 25, 2015 at 12:06 am

Hi,

Thanks I could fix it. The nugget Microsoft.Owin.Security.Jwt -Version 3.0.0 was not installed.

Thanks!

Reply

alexismeilland says
August 29, 2015 at 3:06 am

Hi,

I can create my token with postman and it works perfectly but with my app it does not work.I have the error:

XMLHttpRequest cannot load http://localhost:7241/oauth/token. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:9000′ is therefore not allowed access. The response had HTTP status code 400.

I have enabled cors in my web api.

And in my angularjs app.js file:

$httpProvider.defaults.useXDomain = true;
delete $httpProvider.defaults.headers.common['X-Requested-With'];
// $httpProvider.defaults.withCredentials = true;
$httpProvider.interceptors.push('AuthInterceptor');

I cannot sort it out. Any ideas?

I tried with chrome anf irefox.

Thanks,

Reply

Taiseer Joudeh says
September 7, 2015 at 6:52 am

Hi,
you need to allow it here too As well make sure you are using the same NuGet versions I'm using.

Reply

Avi says
September 8, 2015 at 2:50 am

Hi Taiseer, This absolutely awesome. I do have a question, let's say a third party untrusted client is trying to accessing the web api. So obviously, I don't want the user to create their account and authenticate via the third party client, how can I make sure third party request redirects to my webapi backend, so that they can authenticate on my webapi and then go back to the third party client once done.

Reply

Taiseer Joudeh says
September 10, 2015 at 1:05 pm

Hi Avi,
You need to use a different grant here which is the implicit flow or authorization code flow, where the user will enter his credentials not in the untrusted client, but directly in the authorization server. I didn't implement this but this post will help you to achieve what you are looking for, or you can consider using the ThinkTecture Identity Server.

Reply

Avi says
September 10, 2015 at 9:23 pm

Thanks Taiseer. I'm thinking of looking at the Thinktecture Id server.

Reply

Jeroen-bart Engelen says
September 10, 2015 at 6:29 pm

Great article, but I'm having trouble getting it to work.
I've made a test project based on this article and this
(http://odetocode.com/blogs/scott/archive/2015/01/15/using-json-web-tokens-with-katana-and-

webapi.aspx) article. The project succesfully creates a JWT token, but I cannot perform any requests using that token. I keep getting an HTTP 401 error. The problem is that I have no idea how to debug this inner process. It;s all handled by the Authorize attribute. Do you have any pointers how to troubleshoot this problem?

Thanks!

Reply

Taiseer Joudeh says
September 10, 2015 at 6:46 pm

Hi Jeroen, this issued JWT token should be trusted by your resource APi, please check step 6 from my post and make sure that AudienceId, secret and issuer are all set correctly.

Reply

Jeroen-bart Engelen says
September 10, 2015 at 9:04 pm

After building the complete Owin framework from source to be able to debug it, I was able to solve it.
I'm using Web API and I want my API calls to be authorized. A call with bearer token was denied constantly, even though I was able to get an OAuth token.
I put a breakpoint on AuthenticateCoreAsync() in OAuthBearerAuthenticationHandler from the Microsoft Owin security library I noticed it wouldn't get hit. At all. As if the authentication middleware

wasn't loaded. I checked my implementation 10 times and it all seemed good. Per chance I requested the index page (which is just an empty directory) and then the breakpoint was hit! But for some reason, requesting the Web API endpoint didn't work.
I then changed the order in which I loaded the middleware from:
app.UseWebApi(config);

app.UseWebApi(config);
app.UseJwtBearerAuthentication(new JwtOptions());
app.UseOAuthAuthorizationServer(new OAuthOptions());

to:
app.UseJwtBearerAuthentication(new JwtOptions());
app.UseOAuthAuthorizationServer(new OAuthOptions());
app.UseWebApi(config);

and then it worked….
I would've never thought the order would be important….

Reply

Carlos Boichuk says
September 13, 2015 at 11:27 pm

Taiseer, this is an excellent article, I have a question about how to customize the json web token consumption in a web api resource server, i want to not only validate the json web token receive on each request also i want to read the token received, which methods should i override in the web api resource server in order to perform the current json web token validation as the one you mentioned in this article and in the others, and implementing

this kind of additional logic to it ?

Thanks for your articles, ill be waiting for your reply.

Reply

Taiseer Joudeh says
September 18, 2015 at 2:50 pm

Hi Carols, I need to check this, if you find an answer please share it.

Reply

Michael Heribert says
September 20, 2015 at 11:08 am

Hi, thank you for the great tutorial, it helped me a lot! Could you pls explain how exactly you implemented the as:AudienceId and Secret? How can I create the values for my application? And when using the function linked in the description, how should it be implemented into the application?

thanks in advance 😃

Reply

Michael Heribert says
September 20, 2015 at 8:59 pm

Hi, I managed to solve the problem: using the x-www-form-urlencoded option for the body produces an error, using the raw tab instead solved the problem whereas I expected the audience-key to be the source of the problem.

greetings
Michael Heribert

Reply

mbeddedsoft says
September 21, 2015 at 5:09 pm

I had a problem when I accidently updated packages to latest versions for *.owin.* in PM. I meant to only update EF and the .net framework to 4.5.1. After updating I noticed that the method 'public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)' was not being called. I set a breakpoint in the method and it never hit when I tried making a REST client to call POST http://localhost:port/oauth/token with username, password, and grant_type params. Reverting back to the 2.0 versions identified in article 1 fixed this.

Would you know why this would be? Just asking.

BTW, I am really enjoying your article. Good information. I also really appreciate all the referenced links and articles. You are doing a fantastic job. It's good to be able to follow the tutorial and also have references to more detailed information on Identity 2.0 and OWIN.

thank you so much.

Reply

« Older Comments

Trackbacks

OAuth JWT access token expiration depending on type of client | FYTRO SPORTS says:

May 15, 2015 at 5:07 pm

[…] I created a JWT token implementation based on Taiseer’s tutorial. […]

Reply

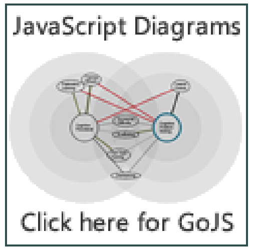
Leave a Reply

ABOUT TAISEER

Father, MVP (ASP .NET/IIS), Scrum Master, Life Time Learner

CONNECT WITH ME

Microsoft® MVP Most Valuable Professional


JavaScript Diagrams
Click here for GoJS

RECENT POSTS

ASP.NET Web API Claims Authorization with ASP.NET Identity 2.1 – Part 5

ASP.NET Identity 2.1 Roles Based Authorization with ASP.NET Web API – Part 4

Implement OAuth JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1 – Part 3

ASP.NET Identity 2.1 Accounts Confirmation, and Password Policy Configuration – Part 2

Interview with John about establishing a successful blog

BLOG ARCHIVES

Select Month ▾

LEAVE YOUR EMAIL AND KEEP TUNED!

Sign up to receive email updates on every new post!

Email Address

SUBSCRIBE

RECENT POSTS

- ASP.NET Web API Claims Authorization with ASP.NET Identity 2.1 – Part 5
- ASP.NET Identity 2.1 Roles Based Authorization with ASP.NET Web API – Part 4
- Implement OAuth JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1 – Part 3
- ASP.NET Identity 2.1 Accounts Confirmation, and Password Policy Configuration – Part 2
- Interview with John about establishing a successful blog

TAGS

AJAX AngularJS API API Versioning ASP.NET Attribute Routing Authentication Autherization Server basic authentication C# CacheCow Client Side Templating Code First Dependency Injection Entity Framework ETag Foursquare API HTTP Caching HTTP Verbs IMDB API IoC Javascript jQuery JSON JSON Web Tokens JWT Model Factory Ninject OAuth OData Pagination Resources Association Resource Server REST RESTful Single Page Applications SPA Token Authentication Tutorial Web API Web API 2 Web API Security Web Service wordpress.com wordpress.org

CONNECT WITH ME

SEARCH

Search this website…