Token Based Authentication using ASP.NET Web API 2, Owin, and Identity

June 1, 2014 By Taiseer Joudeh – 663 Comments

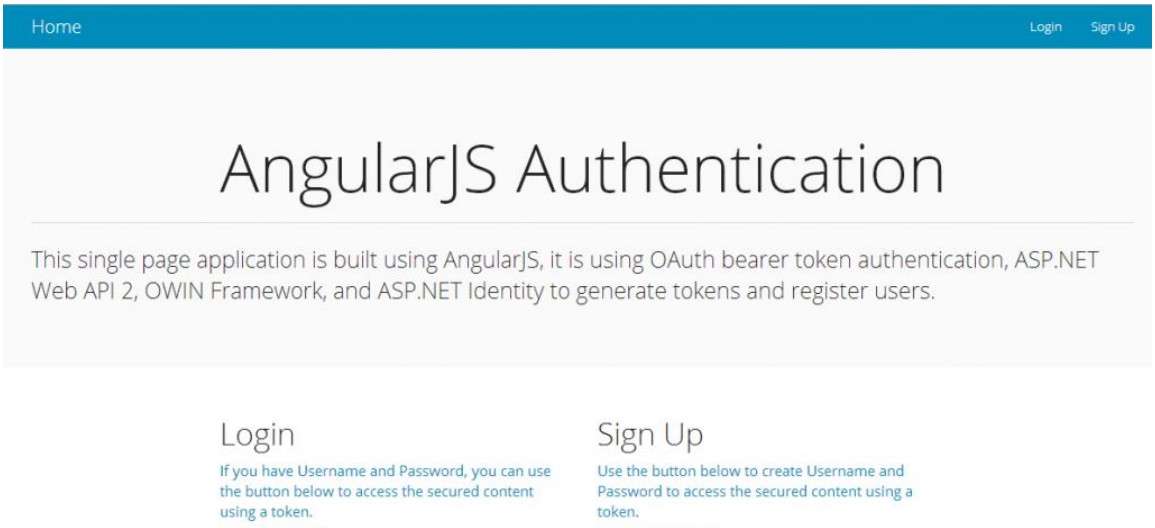
Last week I was looking at the top viewed posts on my blog and I noticed that visitors are interested in the authentication part of ASP.NET Web API, CORS Support, and how to authenticate users in single page applications built with AngularJS using token based approach.

So I decided to compile mini tutorial of ~~three~~ five posts which covers and connects those topics. In this tutorial we'll build SPA using AngularJS for the front-end, and ASP.NET Web API 2, Owin middleware, and ASP.NET Identity for the back-end.

- AngularJS Token Authentication using ASP.NET Web API 2, Owin, and ASP.NET Identity – Part 2.
- Enable OAuth Refresh Tokens in AngularJS App using ASP .NET Web API 2, and Owin – Part 3.
- ASP.NET Web API 2 external logins with Facebook and Google in AngularJS app – Part 4.
- Decouple OWIN Authorization Server from Resource Server – Part 5.

- New post: Secure ASP.NET Web API 2 using Azure Active Directory, Owin Middleware, and ADAL.
- New post: Two Factor Authentication in ASP.NET Web API & AngularJS using Google Authenticator.

The demo application can be accessed on (http://ngAuthenticationWeb.azurewebsites.net). The back-end API can be accessed on (http://ngAuthenticationAPI.azurewebsites.net/) and both are hosted on Microsoft Azure, for learning purposes feel free to integrate and play with the back-end API with your front-end application. The API supports CORS and accepts HTTP calls from any origin. You can check the source code for this tutorial on Github.



Token Based Authentication

As I stated before we'll use token based approach to implement authentication between the front-end application and the back-end API, as we all know the common and old way to implement authentication is the cookie-based approach were the cookie is sent with each request from the client to the server, and on the server it is used to identify the authenticated user.

With the evolution of front-end frameworks and the huge change on how we build web applications nowadays the preferred approach to authenticate users is to use signed token as this token sent to the server with each request, some of the benefits for using this approach are:

- **Scalability of Servers:** The token sent to the server is self contained which holds all the user information needed for authentication, so adding more servers to your web farm is an easy task, there is no dependent on shared session stores.
- **Loosely Coupling:** Your front-end application is not coupled with specific authentication mechanism, the token is generated from the server and your API is built in a way to understand this token and do the authentication.
- **Mobile Friendly:** Cookies and browsers like each other, but storing cookies on native platforms (Android, iOS, Windows Phone) is not a trivial task, having standard way to authenticate users will simplify our life if we decided to consume the back-end API from native applications.

What we'll build in this tutorial?

The front-end SPA will be built using HTML5, AngularJS, and Twitter Bootstrap. The back-end server will be built using ASP.NET Web API 2 on top of Owin middleware not directly on top of ASP.NET; the reason for doing so that we'll configure the server to issue OAuth bearer token authentication using Owin middleware too, so setting up everything on the same pipeline is better approach. In addition to this we'll use ASP.NET Identity system which is built on top of Owin middleware and we'll use it to register new users and validate their credentials before generating the tokens.

As I mentioned before our back-end API should accept request coming from any origin, not only our front-end, so we'll be enabling CORS (Cross Origin Resource Sharing) in Web API as well for the OAuth bearer token provider.

Use cases which will be covered in this application:

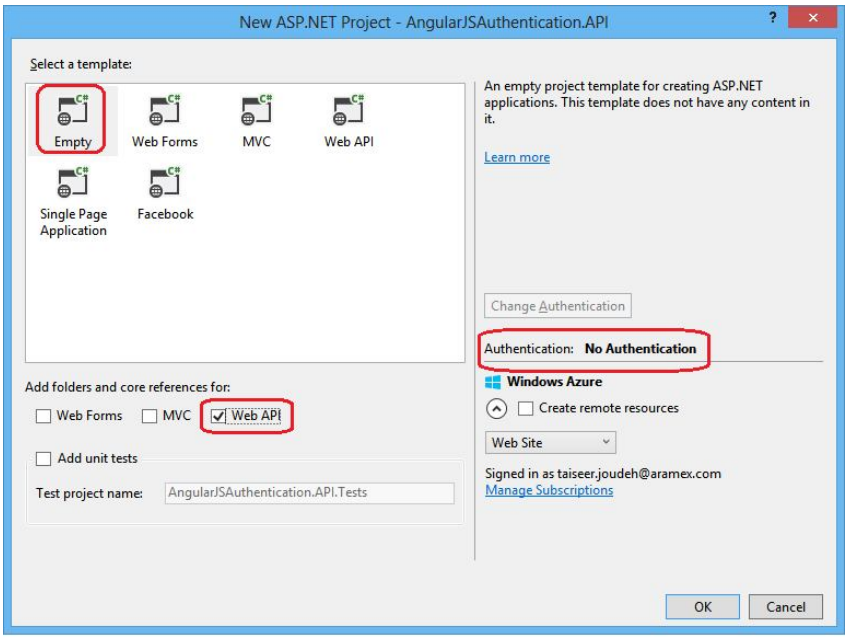
- Allow users to signup (register) by providing username and password then store credentials in secure medium.
- Prevent anonymous users from viewing secured data or secured pages (views).
- Once the user is logged in successfully, the system should not ask for credentials or re-authentication for the next ~~24 hours~~ 30 minutes because we are using refresh tokens.

So in this post we'll cover step by step how to build the back-end API, and on the next post we'll cover how we'll build and integrate the SPA with the API.

Enough theories let's get our hands dirty and start implementing the API!

2012 but you need to install Web Tools 2013.1 for VS 2012 by visiting this link.

Now create an empty solution and name it "AngularJSAuthentication" then add new ASP.NET Web application named "AngularJSAuthentication.API", the selected template for project will be as the image below. Notice that the authentication is set to "No Authentication" taking into consideration that we'll add this manually.



Step 2: Installing the needed NuGet Packages:

Now we need to install the NuGet packages which are needed to setup our Owin server and configure ASP.NET Web API to be hosted within an Owin server, so open NuGet Package Manger Console and type the below:

```
Install-Package Microsoft.AspNet.WebApi.Owin -Version 5.1.2
Install-Package Microsoft.Owin.Host.SystemWeb -Version 2.1.0
```

The package "Microsoft.Owin.Host.SystemWeb" is used to enable our Owin server to run our API on IIS using ASP.NET request pipeline as eventually we'll host this API on Microsoft Azure Websites which uses IIS.

Step 3: Add Owin "Startup" Class

Right click on your project then add new class named "Startup". We'll visit this class many times and modify it, for now it will contain the code below:

```
using Microsoft.Owin;
using Owin;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;

[assembly: OwinStartup(typeof(AngularJSAuthentication.API.Startup))]
namespace AngularJSAuthentication.API
{
```

```
20
21      }
22  }
```

What we've implemented above is simple, this class will be fired once our server starts, notice the "assembly" attribute which states which class to fire on start-up. The "Configuration" method accepts parameter of type "IAppBuilder" this parameter will be supplied by the host at run-time. This "app" parameter is an interface which will be used to compose the application for our Owin server.

The "HttpConfiguration" object is used to configure API routes, so we'll pass this object to method "Register" in "WebApiConfig" class.

Lastly, we'll pass the "config" object to the extension method "UseWebApi" which will be responsible to wire up ASP.NET Web API to our Owin server pipeline.

Usually the class "WebApiConfig" exists with the templates we've selected, if it doesn't exist then add it under the folder "App_Start". Below is the code inside it:

```
1       public static class WebApiConfig
2       {
3           public static void Register(HttpConfiguration config)
4           {
5
6               // Web API routes
7               config.MapHttpAttributeRoutes();
8
9               config.Routes.MapHttpRoute(
10                  name: "DefaultApi",
11                  routeTemplate: "api/{controller}/{id}",
12                  defaults: new { id = RouteParameter.Optional }
13              );
14
15              var jsonFormatter = config.Formatters.OfType<JsonMediaTypeFormatter>().First();
16              jsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();
17          }
18      }
```

Step 4: Delete Global.asax Class

No need to use this class and fire up the Application_Start event after we've configured our "Startup" class so feel free to delete it.

Step 5: Add the ASP.NET Identity System

After we've configured the Web API, it is time to add the needed NuGet packages to add support for registering and validating user credentials, so open package manager console and add the below NuGet packages:

```
1 Install-Package Microsoft.AspNet.Identity.Owin -Version 2.0.1
2 Install-Package Microsoft.AspNet.Identity.EntityFramework -Version 2.0.1
```

The first package will add support for ASP.NET Identity Owin, and the second package will add support for using ASP.NET Identity with Entity Framework so we can save users to SQL Server database.

Now we need to add Database context class which will be responsible to communicate with our database, so add new class and name it "AuthContext" then paste the code snippet below:

```
    }
```

As you can see this class inherits from "IdentityDbContext" class, you can think about this class as special version of the traditional "DbContext" Class, it will provide all of the Entity Framework code-first mapping and DbSet properties needed to manage the identity tables in SQL Server. You can read more about this class on Scott Allen Blog.

Now we want to add "UserModel" which contains the properties needed to be sent once we register a user, this model is POCO class with some data annotations attributes used for the sake of validating the registration payload request. So under "Models" folder add new class named "UserModel" and paste the code below:

```
public class UserModel
    {
        [Required]
        [Display(Name = "User name")]
        public string UserName { get; set; }

        [Required]
        [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirm password")]
        [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
        public string ConfirmPassword { get; set; }
    }
```

Now we need to add new connection string named "AuthContext" in our Web.Config class, so open you web.config and add the below section:

```
<connectionStrings>
    <add name="AuthContext" connectionString="Data Source=.\sqlexpress;Initial Catalog=AngularJSAuth;Integrated Securit
  </connectionStrings>
```

Step 6: Add Repository class to support ASP.NET Identity System

Now we want to implement two methods needed in our application which they are: "RegisterUser" and "FindUser", so add new class named "AuthRepository" and paste the code snippet below:

```
    public class AuthRepository : IDisposable
    {
        private AuthContext _ctx;

        private UserManager<IdentityUser> _userManager;

        public AuthRepository()
        {
            _ctx = new AuthContext();
            _userManager = new UserManager<IdentityUser>(new UserStore<IdentityUser>(_ctx));
        }

        public async Task<IdentityResult> RegisterUser(UserModel userModel)
        {
            IdentityUser user = new IdentityUser
            {
                UserName = userModel.UserName
            };

            var result = await _userManager.CreateAsync(user, userModel.Password);

            return result;
```

```
31
32          public void Dispose()
33          {
34              _ctx.Dispose();
35              _userManager.Dispose();
36
37          }
38      }
```

What we've implemented above is the following: we are depending on the "UserManager" that provides the domain logic for working with user information. The "UserManager" knows when to hash a password, how and when to validate a user, and how to manage claims. You can read more about ASP.NET Identity System.

Step 7: Add our "Account" Controller

Now it is the time to add our first Web API controller which will be used to register new users, so under file "Controllers" add Empty Web API 2 Controller named "AccountController" and paste the code below:

```
1   [RoutePrefix("api/Account")]
2       public class AccountController : ApiController
3       {
4           private AuthRepository _repo = null;
5
6           public AccountController()
7           {
8               _repo = new AuthRepository();
9           }
10
11          // POST api/Account/Register
12          [AllowAnonymous]
13          [Route("Register")]
14          public async Task<IHttpActionResult> Register(UserModel userModel)
15          {
16              if (!ModelState.IsValid)
17              {
18                  return BadRequest(ModelState);
19              }
20
21              IdentityResult result = await _repo.RegisterUser(userModel);
22
23              IHttpActionResult errorResult = GetErrorResult(result);
24
25              if (errorResult != null)
26              {
27                  return errorResult;
28              }
29
30              return Ok();
31          }
32
33          protected override void Dispose(bool disposing)
34          {
35              if (disposing)
36              {
37                  _repo.Dispose();
38              }
39
40              base.Dispose(disposing);
41          }
42
43          private IHttpActionResult GetErrorResult(IdentityResult result)
44          {
45              if (result == null)
46              {
47                  return InternalServerError();
48              }
49
50              if (!result.Succeeded)
51              {
52                  if (result.Errors != null)
53                  {
54                      foreach (string error in result.Errors)
55                      {
56                          ModelState.AddModelError("", error);
```

```
65
66                  return BadRequest(ModelState);
67              }
68
69              return null;
70          }
71      }
```

By looking at the "Register" method you will notice that we've configured the endpoint for this method to be "/api/account/register" so any user wants to register into our system must issue HTTP POST request to this URI and the pay load for this request will contain the JSON object as below:

```
1 {
2   "userName": "Taiseer",
3   "password": "SuperPass",
4   "confirmPassword": "SuperPass"
5 }
```

Now you can run your application and issue HTTP POST request to your local URI: "http://localhost:port /api/account/register" or you can try the published API using this end point: http://ngauthenticationapi.azurewebsites.net/api/account/register if all went fine you will receive HTTP status code 200 and the database specified in connection string will be created automatically and the user will be inserted into table "dbo.AspNetUsers".

Note: It is very important to send this POST request over HTTPS so the sensitive information get encrypted between the client and the server.

The "GetErrorResult" method is just a helper method which is used to validate the "UserModel" and return the correct HTTP status code if the input data is invalid.

Step 8: Add Secured Orders Controller

Now we want to add another controller to serve our Orders, we'll assume that this controller will return orders only for Authenticated users, to keep things simple we'll return static data. So add new controller named "OrdersController" under "Controllers" folder and paste the code below:

```
1  [RoutePrefix("api/Orders")]
2      public class OrdersController : ApiController
3      {
4          [Authorize]
5          [Route("")]
6          public IHttpActionResult Get()
7          {
8              return Ok(Order.CreateOrders());
9          }
10
11     }
12
13     #region Helpers
14
15     public class Order
16     {
17         public int OrderID { get; set; }
18         public string CustomerName { get; set; }
19         public string ShipperCity { get; set; }
20         public Boolean IsShipped { get; set; }
21
22         public static List<Order> CreateOrders()
23         {
24             List<Order> OrderList = new List<Order>
25             {
26                 new Order {OrderID = 10248, CustomerName = "Taiseer Joudeh", ShipperCity = "Amman", IsShipped = true }
27                 new Order {OrderID = 10249, CustomerName = "Ahmad Hasan", ShipperCity = "Dubai", IsShipped = false},
```

```
36
37      #endregion
```

Notice how we added the "Authorize" attribute on the method "Get" so if you tried to issue HTTP GET request to the end point "http://localhost:port/api/orders" you will receive HTTP status code 401 unauthorized because the request you send till this moment doesn't contain valid authorization header. You can check this using this end point: http://ngauthenticationapi.azurewebsites.net/api/orders

Step 9: Add support for OAuth Bearer Tokens Generation

Till this moment we didn't configure our API to use OAuth authentication workflow, to do so open package manager console and install the following NuGet package:

```
1 Install-Package Microsoft.Owin.Security.OAuth -Version 2.1.0
```

After you install this package open file "Startup" again and call the new method named "ConfigureOAuth" as the first line inside the method "Configuration", the implemntation for this method as below:

```
1  public class Startup
2      {
3          public void Configuration(IAppBuilder app)
4          {
5              ConfigureOAuth(app);
6          //Rest of code is here;
7          }
8
9          public void ConfigureOAuth(IAppBuilder app)
10         {
11             OAuthAuthorizationServerOptions OAuthServerOptions = new OAuthAuthorizationServerOptions()
12             {
13                 AllowInsecureHttp = true,
14                 TokenEndpointPath = new PathString("/token"),
15                 AccessTokenExpireTimeSpan = TimeSpan.FromDays(1),
16                 Provider = new SimpleAuthorizationServerProvider()
17             };
18
19             // Token Generation
20             app.UseOAuthAuthorizationServer(OAuthServerOptions);
21             app.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions());
22
23         }
24     }
```

Here we've created new instance from class "OAuthAuthorizationServerOptions" and set its option as the below:

- The path for generating tokens will be as :"http://localhost:port/token". We'll see how we will issue HTTP POST request to generate token in the next steps.
- We've specified the expiry for token to be 24 hours, so if the user tried to use the same token for authentication after 24 hours from the issue time, his request will be rejected and HTTP status code 401 is returned.
- We've specified the implementation on how to validate the credentials for users asking for tokens in custom class named "SimpleAuthorizationServerProvider".

Now we passed this options to the extension method "UseOAuthAuthorizationServer" so we'll add the authentication middleware to the pipeline.

Step 10: Implement the "SimpleAuthorizationServerProvider" class

```
3          public override async Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)
4          {
5              context.Validated();
6          }
7
8          public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
9          {
10
11             context.OwinContext.Response.Headers.Add("Access-Control-Allow-Origin", new[] { "*" });
12
13             using (AuthRepository _repo = new AuthRepository())
14             {
15                 IdentityUser user = await _repo.FindUser(context.UserName, context.Password);
16
17                 if (user == null)
18                 {
19                     context.SetError("invalid_grant", "The user name or password is incorrect.");
20                     return;
21                 }
22             }
23
24             var identity = new ClaimsIdentity(context.Options.AuthenticationType);
25             identity.AddClaim(new Claim("sub", context.UserName));
26             identity.AddClaim(new Claim("role", "user"));
27
28             context.Validated(identity);
29
30         }
31     }
```

As you notice this class inherits from class “OAuthAuthorizationServerProvider”, we’ve overridden two methods “ValidateClientAuthentication” and “GrantResourceOwnerCredentials”. The first method is responsible for validating the “Client”, in our case we have only one client so we’ll always return that its validated successfully.

The second method “GrantResourceOwnerCredentials” is responsible to validate the username and password sent to the authorization server’s token endpoint, so we’ll use the “AuthRepository” class we created earlier and call the method “FindUser” to check if the username and password are valid.

If the credentials are valid we’ll create “ClaimsIdentity” class and pass the authentication type to it, in our case “bearer token”, then we’ll add two claims (“sub”,”role”) and those will be included in the signed token. You can add different claims here but the token size will increase for sure.

Now generating the token happens behind the scenes when we call “context.Validated(identity)”.

To allow CORS on the token middleware provider we need to add the header “Access-Control-Allow-Origin” to Owin context, if you forget this, generating the token will fail when you try to call it from your browser. Not that this allows CORS for token middleware provider not for ASP.NET Web API which we’ll add on the next step.

Step 11: Allow CORS for ASP.NET Web API

First of all we need to install the following NuGet package manger, so open package manager console and type:

```
1 Install-Package Microsoft.Owin.Cors -Version 2.1.0
```

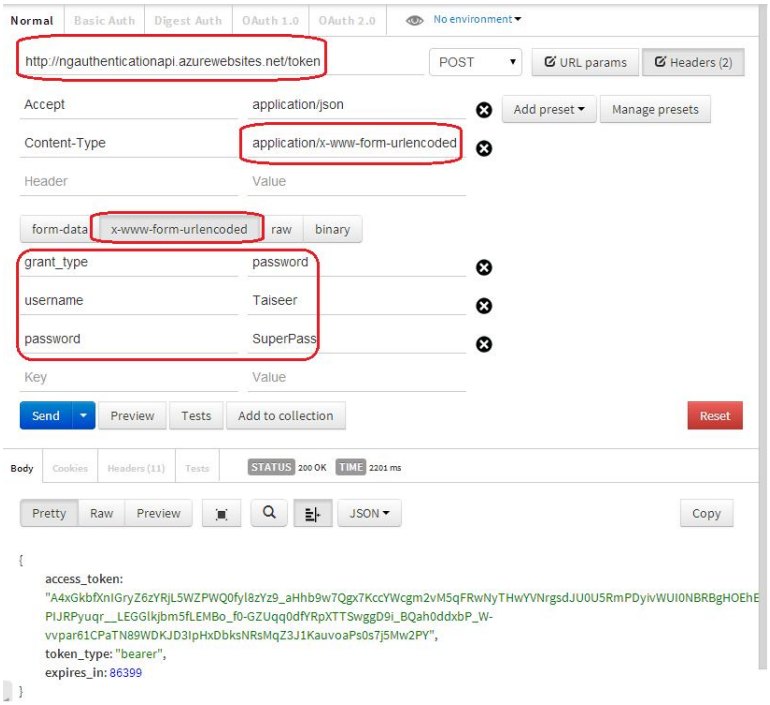
Now open class “Startup” again and add the highlighted line of code (line 8) to the method “Configuration” as the below:

```
8             app.UseCors(Microsoft.Owin.Cors.CorsOptions.AllowAll);
9             app.UseWebApi(config);
10
11        }
```

Step 12: Testing the Back-end API

Assuming that you registered the username "Taiseer" with password "SuperPass" in the step below, we'll use the same username to generate token, so to test this out open your favorite REST client application in order to issue HTTP requests to generate token for user "Taiseer". For me I'll be using PostMan.

Now we'll issue a POST request to the endpoint http://ngauthenticationapi.azurewebsites.net/token the request will be as the image below:



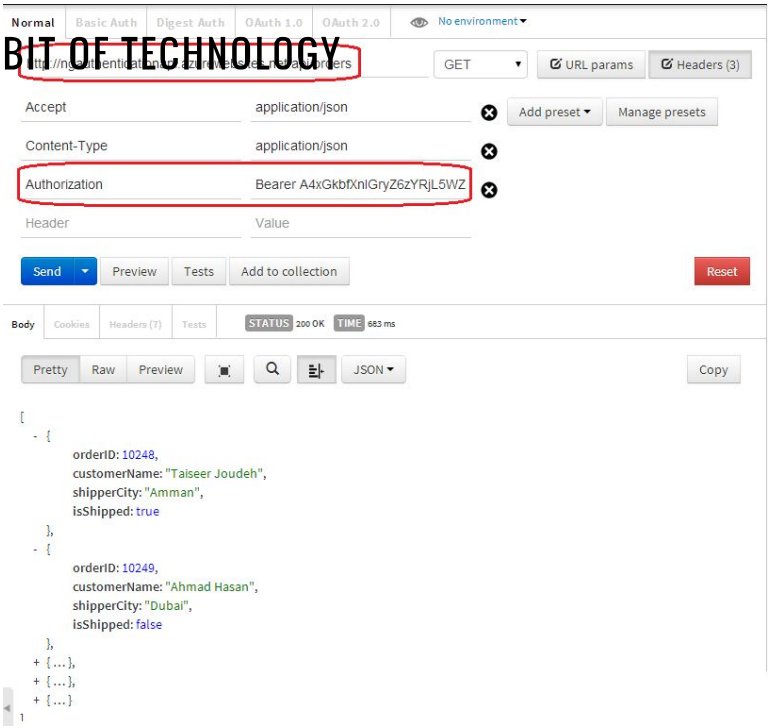
Notice that the content-type and payload type is "x-www-form-urlencoded" so the payload body will be on form (grant_type=password&username="Taiseer"&password="SuperPass"). If all is correct you'll notice that we've received signed token on the response.

As well the "grant_type" Indicates the type of grant being presented in exchange for an access token, in our case it is password.

Now we want to use this token to request the secure data using the end point http://ngauthenticationapi.azurewebsites.net/api/orders so we'll issue GET request to the end point and will pass the bearer token in the Authorization header, so for any secure end point we've to pass this bearer token along with each request to authenticate the user.

Note: that we are not transferring the username/password as the case of Basic authentication.

The GET request will be as the image below:



If all is correct we'll receive HTTP status 200 along with the secured data in the response body, if you try to change any character with signed token you directly receive HTTP status code 401 unauthorized.

Now our back-end API is ready to be consumed from any front end application or native mobile app.

Update (2014-08-11) Thanks for Attila Hajdrik for forking my repo and updating it to use MongoDb instead of Entity Framework, you can check it here.

You can check the demo application, play with the back-end API for learning purposes (http://ngauthenticationapi.azurewebsites.net), and check the source code on Github.

Follow me on Twitter @tjoudeh

References

- 10 Things You Should Know about Tokens by Matias Woloski
- SPA Authentication Example by David Antaramian

Related Posts

ASP.NET Web API Claims Authorization with ASP.NET Identity 2.1 – Part 5
ASP.NET Identity 2.1 Accounts Confirmation, and Password Policy Configuration – Part 2
ASP.NET Identity 2.1 with ASP.NET Web API 2.2 (Accounts Management) – Part 1

Comments

Kal Wolfson says
April 30, 2015 at 10:52 am

Hi Taiser

Sadly I have followed your tutorial to the T. And nothing seems to work. I'm not using SQL Express just SQL2008R2 so my connection string is
"

" Instead of

But everything else is a carbon copy of what you have done.

I'm using Postman for the requests

When I post both "http://localhost:1587/api/account/register" or "http://ngauthenticationapi.azurewebsites.net /api/account/register" I get status code of "500 internal Server Error"

When I issue a POST request to the endpoint "http://ngauthenticationapi.azurewebsites.net/token" I get a status code of "400 Bad Request" with a message of "{"error":"unsupported_grant_type"}".

When I do a GET Request for "http://ngauthenticationapi.azurewebsites.net/api/orders" I get a status code of"401 Unorthorized" with a message of "{"message":"Authorization has been denied for this request."}"

My Startup.cs is

```
[assembly:OwinStartup(typeof(AngularJSAuthentication.API.Startup))]
namespace AngularJSAuthentication.API
{
public class Startup
{
public void Configuration(IAppBuilder app)
{
var config = new HttpConfiguration();
```

```
app.UseCors(Microsoft.Owin.Cors.CorsOptions.AllowAll);
app.UseWebApi(config);

}

public void ConfigureOAuth(IAppBuilder app)
{
var OAuthServerOptions = new OAuthAuthorizationServerOptions()
{
AllowInsecureHttp = true,
TokenEndpointPath = new PathString("/token"),
AccessTokenExpireTimeSpan = TimeSpan.FromDays(1),
Provider = new SimpleAuthorizationServerProvider()
};

// Token Generation
app.UseOAuthAuthorizationServer(OAuthServerOptions);
app.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions());

}

public OAuthAuthorizationServerOptions OAuthServerOptions { get; set; }
}
}
```

Please help me find the problem?
I fear I might lose my job if I cant get these things working.

Reply

Taiseer Joudeh says
April 30, 2015 at 1:02 pm

I'm sad to hear that not doing a task will cause you job lose, if I were you I will leave my employer right now! Anyhow what the 500 error you are receiving? I think that you have issue when composing the request, are you using postman or something else?

Reply

víctor hugo garcía (@ivictorhugo) says
May 1, 2015 at 10:35 pm

Does the approach mentioned in the article, makes it to meets the requirements HIPAA compliance? Since the web server, I'm building is going to store medical and health information, that's why I'm looking for the best security approach to secure the web services, which will be consumed by native Android and iOS apps.

Reply

I will paste my answer here too for other users:

Regarding the HIPAA compliance, you need to Google this more as I can't guarantee this, I was looking at this post and it looks OAuth 2.0 is complaint.

Now the grant I'm using in my post is called "Resource owner password credential flow" and you need to make sure that you have full control on the source code written for the mobile application, in other words its very trusted mobile application that you are going to build or trusted third party will build it for you, think of it like the Official Twitter or Facebook application that you install it on your mobile and you present the Username/password in a native UI from the application, because if you do not trust this application you will not know how the App developer will use those credentials, maybe he will be evil enough to send them to another endpoint and harvest all your username/credentials.

Now if this is not the case you need to use the Implicit flow or Authorization code flow where the user enters his credentials in Web page provided by the Authorization Server so the mobile application never sees the credentials and just receive an access token.

For such critical application you are building, I recommend you to check as well the Regarding the HIPAA compliance, you need to Google this more as I can't guarantee this, I was looking at this post and it looks OAuth 2.0 is complaint.

Now the grant I'm using in my post is called "Resource owner password credential flow" and you need to make sure that you have full control on the source code written for the mobile application, in other words its very trusted mobile application that you are going to build or trusted third party will build it for you, think of it like the Official Twitter or Facebook application that you install it on your mobile and you present the Username/password in a native UI from the application, because if you do not trust this application you will not know how the App developer will use those credentials, maybe he will be evil enough to send them to another endpoint and harvest all your username/credentials.

Now if this is not the case you need to use the Implicit flow or Authorization code flow where the user enters his credentials in Web page provided by the Authorization Server so the mobile application never sees the credentials and just receive an access token.

For such critical application you are building, I recommend you to check as well the ThinkTecture Identity Server

Reply

Devon says
August 14, 2014 at 12:32 am

To be clear the Post works OK and a token is returned. It's when sending the get request with the bearer token returned from the POST request that the error occurs:

GET /token?Accept=application/json&Content-Type=application/x-www-form-urlencoded&
Authorization=Bearer mc0UaZiHM97ueb8xiEQqeZ5oCwMnF6ZoKgEHp0cd8ftSQ5MCAHkwo6HAcOPFQil3k7hjD-dB-tX5OHTzImbP8ncHUU6ZMt8a1OYVGhv59AzN1f7YiaNydHPnIrjbI-U74aEtAyFrmghtK0_JjIvBBGM3kxYW1hVFP_BDhljFA33LVhEfzqQFxGJIobIv0C6gHfoqeXjW9g857BUSA1xjk24PmaQK9KFORugmK6Jx68s HTTP/1.1

Reply

Taiseer Joudeh says
August 14, 2014 at 12:38 am

Hi Devon, you do not need to send Get request to /token endpoint again, you need to send it to the secured endpoint which is "/orders" and set the token. As well the content type should be application/json when sending this request. Hope this answers your question.

Reply

Devon says
August 14, 2014 at 12:39 am

Finally noticed I forgot to change my url on the get. Silly mistake. Also had accidentally left the URL params tab click in postman rather than Headers. Trying to work too quickly as the end of the day neared.

Reply

Hemant says
September 20, 2014 at 7:07 pm

I'm getting exception as "error": "unsupported_grant_type"

Image at http://1drv.ms/1ykj67F

Reply

Taiseer Joudeh says
September 20, 2014 at 7:18 pm

Build the request like this https://www.getpostman.com/collections/1ff438e088efac658be8 and you will be good.

Reply

Hemant says
September 23, 2014 at 9:56 pm

Tom says
January 23, 2015 at 6:16 pm

Hi Taiseer,
I'm getting "unsupported_grant_type", but what do you mean with "Build the request like this https://www.getpostman.com/collections/1ff438e088efac658be8 "

I have this request:

POST http://localhost:53701/token HTTP/1.1
User-Agent: Fiddler
Host: localhost:53701
Content-Length: 248
Accept: application/json
Content-Type: application/x-www-form-urlencoded

[
{
"key": "username",
"value": "Test",
"type": "text"
},
{
"key": "password",
"value": "MyPass",
"type": "text"
},
{
"key": "grant_type",
"value": "password",
"type": "text"
}
]

Reply

Taiseer Joudeh says
September 23, 2014 at 10:54 pm

You are welcome, happy to help Hemant.

Reply

do you have a suggestion for my problem below?

Thanks

Reply

Jiten says
September 26, 2014 at 10:47 am

Hey Taiseer ! Need your help.

I need to invoke /Token endpoint from OAuth service itself and generate access token and refresh token explicit once your change his password and pass tokens to client. I don't want user to login again.

Thanks,
Jiten

Reply

Aibol says
September 26, 2014 at 11:47 am

Dear Taiseer,

I faced the following problem:

XMLHttpRequest cannot load http://localhost:57606/token. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:32153' is therefore not allowed access.

The problem occurs in Chrome browser. In IE it's ok. Don't you know what the problem could be?

Reply

I believe you need to set the client allowed origin column to match your host i.e. "http://localhost:57606", CORS is not configured correctly in your case. Check this post on how to do it.
IE doesn't consider the port in URI as different URI that's why it is working correctly because it only checks the host-name "localhost", Chrome, Firefox take care of the port as well.

Reply

Taiseer Joudeh says
September 27, 2014 at 1:09 am

Hi Jiten,
With the current implementation you can't do this, what you can do is to use refresh tokens and once the user change his password you delete his refresh token. So when the client application wants to use this refresh token to obtain new access token it will receive 401 and you need to re-authenticate him by providing username and his new password.

Reply

Aibol says
September 27, 2014 at 6:35 am

Thank you Taiseer. Yes, you're correct! The problem was in not configured CORS, e.g. there weren't any rows in Client table.

Unfortunately, I faced another problem when FindClient method is invoked:

An exception of type 'System.Data.Entity.Core.ProviderIncompatibleException' occurred in EntityFramework.dll but was not handled in user code

Additional information: An error occurred accessing the database. This usually means that the connection to the database failed. Check that the connection string is correct and that the appropriate DbContext constructor is being used to specify it or find it in the application's config file. See http://go.microsoft.com/fwlink/?LinkId=386386 for information on DbContext and connections. See the inner exception for details of the failure.

Could you also to clarify what is the difference between the RefreshTokenLifeTime which is saved in Client table and AccessTokenExpireTimeSpan = TimeSpan.FromMinutes(30) in Startup.cs?

Reply

Taiseer Joudeh says
September 30, 2014 at 12:04 am

Hi Aibol, I've covered the difference in the post, but the AccessTokenExpireTimeSpan is used to determine the life

shimulsays says
November 19, 2014 at 9:56 pm

Hi.
Super cool post. Thanx.

Why i am getting this?

XMLHttpRequest cannot load http://localhost:26264/api/account/register. The 'Access-Control-Allow-Origin' header contains multiple values 'http://localhost:32176, *', but only one is allowed. Origin 'http://localhost:32176' is therefore not allowed access.

http://localhost:26264/token is okay.

Reply

Taiseer Joudeh says
November 20, 2014 at 3:05 am

Hi, can you try set the "Access-Control-Allow-Origin" value to "*" only and try again? Try to do this from PostMan or fiddler, it should be working correctly.

Reply

Jerry T says
January 5, 2015 at 1:05 am

I ran into this problem as well and used PostMan (a Chrome app) to send the JSON string to the url as a Post request and now it works. Caused me a bit of frustration till I realized this. Maybe you can add it to the blog post where you tell people to test the url so it's right next to the proper step?

And thanks for all the work done with these blog posts. I really need this info and it looks like this will be perfect for a new project I have to start.

Reply

Taiseer Joudeh says
January 24, 2015 at 1:08 am

Content-Type: application/x-www-form-urlencoded
Cache-Control: no-cache
Postman-Token: 3fc06f29-8f4f-54ed-10c2-3f3045dc8dfa

username=Taiseer&password=SuperPass&grant_type=password

Reply

Tom says
January 26, 2015 at 11:46 am

Hi all,
I have solved my problem whit "unsupported_grant_type" using this request:

POST http://localhost:53701/token HTTP/1.1
User-Agent: Fiddler
Accept: application/json
Content-Type: application/x-www-form-urlencoded
Host: localhost:53701
Content-Length: 51

grant_type=password&username=myuser&password=mypsw

Reply

« Older Comments

Trackbacks

AngularJS Token Authentication using ASP.NET Web API 2, Owin, and Identity - Bit of Technology says:
June 9, 2014 at 2:09 am
[…] Token Based Authentication using ASP.NET Web API 2, Owin middleware, and ASP.NET Identity – Pa… […]

Reply

Auntenticación por Tokens usando OWIN, WebAPI y AngularJs | JRos Code says:
September 5, 2014 at 1:20 pm
[…] AngularJS Token Authentication using ASP.NET Web API 2, Owin, and Identity […]

Reply

Reply

Sharing the Love – Zurb Foundation for Apps Scaffolding for .Net with Authentication | Sup-a-Dillie-O says:
January 20, 2015 at 3:41 pm
[…] core of the authentication piece deserves a HUGE hat tip to Taiseer Joudah, who wrote an amazing series about how to build an app with a .Net web service backend and use an AngularJS architecture on the […]

Reply

How to unit-test your OWIN-configured OAuth2 implementation | To kill a mocking bug says:
March 21, 2015 at 8:29 pm
[…] recently started implementing OAuth2 in a project by following this wonderful blog series by Taiseer Joudeh. So far everything is going great but one thing I'm missing is unit-tests to make sure […]

Reply

Leave a Reply

Enter your comment here...

Fill in your details below or click an icon to log in:

-
-
-
-

Email (required) (Address never made public)

Name (required)

Website

You are commenting using your WordPress.com account. (Log Out / Change)

ABOUT TAISEER

Father, MVP (ASP

BIT OF TECHNOLOGY

BIT OF TECHNOLOGY

RECENT POSTS

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

TAGS

AJAX AngularJS API API Versioning ASP.NET Attribute Routing Authentication

BIT OF TECHNOLOGY

Password Policy Configuration – Part 2

Interview with John about establishing a successful blog

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…