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

June 1, 2014 By Taiseer Joudeh — 1,360 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.

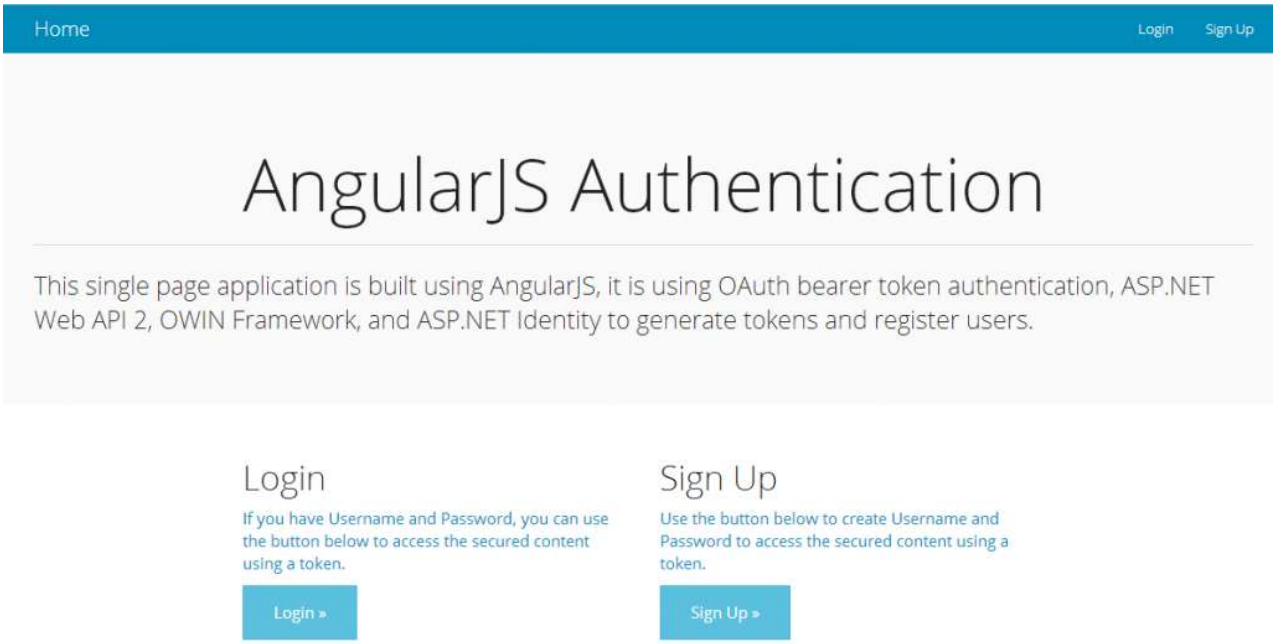
So I decided to compile mini tutorial of ~~three~~ five posts which covers and connects those topics. In this tutorial we'll build SPA using AngularJS for the front-end, and ASP.NET Web API 2, Owin middleware, and ASP.NET Identity for the back-end.

- 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 Aug-2016**: Secure ASP.NET Web API 2 Using Azure AD B2C (Business to Consumer).
- 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.

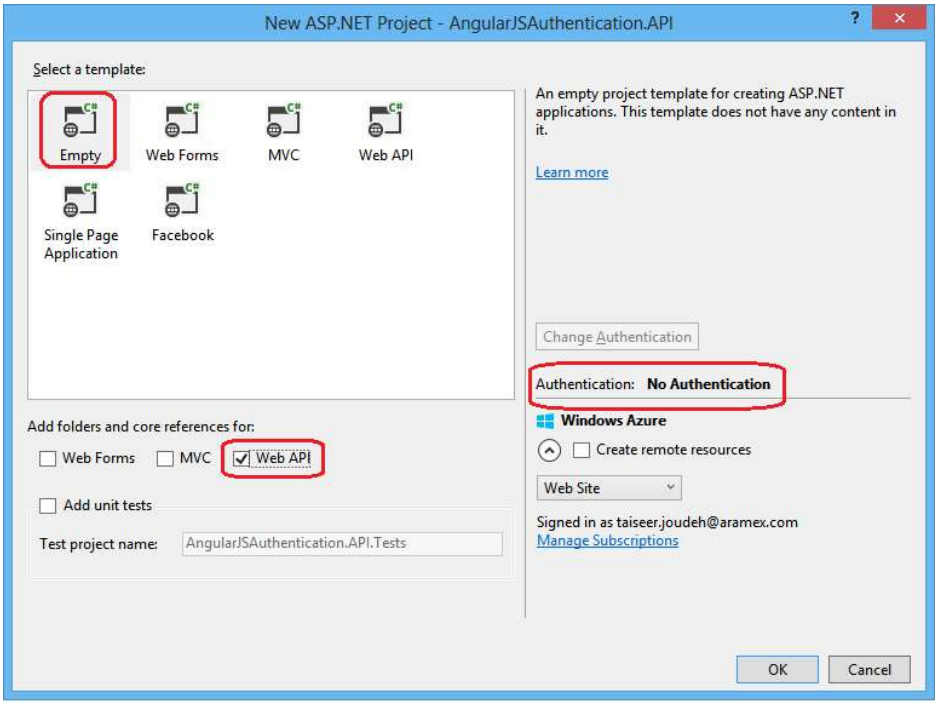
Enough theories let's get our hands dirty and start implementing the API!

## Building the Back-End API

### Step 1: Creating the Web API Project

In this tutorial I'm using Visual Studio 2013 and .Net framework 4.5, you can follow along using Visual Studio 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
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            HttpConfiguration config = new HttpConfiguration();
            WebApiConfig.Register(config);
            app.UseWebApi(config);
        }

    }
}
```

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:

```
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {

            // Web API routes
            config.MapHttpAttributeRoutes();

```

```
            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );

            var jsonFormatter = config.Formatters.OfType<JsonMediaTypeFormatter>().First();
            jsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();
        }
    }
```

## 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:

```
Install-Package Microsoft.AspNet.Identity.Owin -Version 2.0.1
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:

```
public class AuthContext : IdentityDbContext<IdentityUser>
    {
        public AuthContext()
            : base("AuthContext")
        {

        }
    }
```

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;
        }

        public async Task<IdentityUser> FindUser(string userName, string password)
        {
            IdentityUser user = await _userManager.FindAsync(userName, password);

            return user;
        }

        public void Dispose()
        {
            _ctx.Dispose();
            _userManager.Dispose();

        }
    }
```

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:

```
[RoutePrefix("api/Account")]
    public class AccountController : ApiController
    {
        private AuthRepository _repo = null;

        public AccountController()
        {
            _repo = new AuthRepository();
        }

        // POST api/Account/Register
        [AllowAnonymous]
        [Route("Register")]
        public async Task<IHttpActionResult> Register(UserModel userModel)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            IdentityResult result = await _repo.RegisterUser(userModel);

            IHttpActionResult errorResult = GetErrorResult(result);

            if (errorResult != null)
            {
                return errorResult;
            }

            return Ok();
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                _repo.Dispose();
            }

            base.Dispose(disposing);
        }

        private IHttpActionResult GetErrorResult(IdentityResult result)
        {
            if (result == null)
            {
                return InternalServerError();
            }

            if (!result.Succeeded)
            {
                if (result.Errors != null)
                {
                    foreach (string error in result.Errors)
                    {
                        ModelState.AddModelError("", error);
                    }
                }

                if (ModelState.IsValid)
                {
                    // No ModelState errors are available to send, so just return an empty BadRequest.
                    return BadRequest();
                }

                return BadRequest(ModelState);
            }

            return null;
        }
    }
```

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:

```
{
```

```
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},
28                 new Order {OrderID = 10250,CustomerName = "Tamer Yaser", ShipperCity = "Jeddah", IsShipped = false },
29                 new Order {OrderID = 10251,CustomerName = "Lina Majed", ShipperCity = "Abu Dhabi", IsShipped = false},
30                 new Order {OrderID = 10252,CustomerName = "Yasmeen Rami", ShipperCity = "Kuwait", IsShipped = true}
31             };
32
33             return OrderList;
34         }
35     }
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:

```
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:

```
public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            ConfigureOAuth(app);
        //Rest of code is here;
        }

        public void ConfigureOAuth(IAppBuilder app)
        {
            OAuthAuthorizationServerOptions 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());

        }
    }
```

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

Add new folder named "Providers" then add new class named "SimpleAuthorizationServerProvider", paste the code snippet below:

```
public class SimpleAuthorizationServerProvider : OAuthAuthorizationServerProvider
    {
        public override async Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)
        {
            context.Validated();
        }
```

```
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:

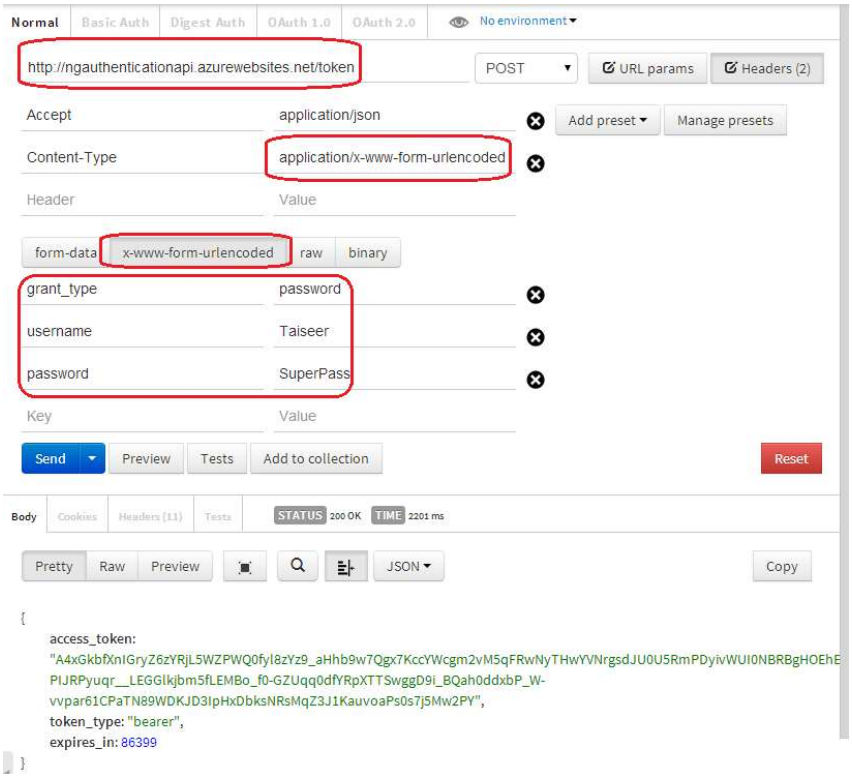
```
1   public void Configuration(IAppBuilder app)
2           {
3               HttpConfiguration config = new HttpConfiguration();
4
```

```
            ConfigureOAuth(app);

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

        }
```

## 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:



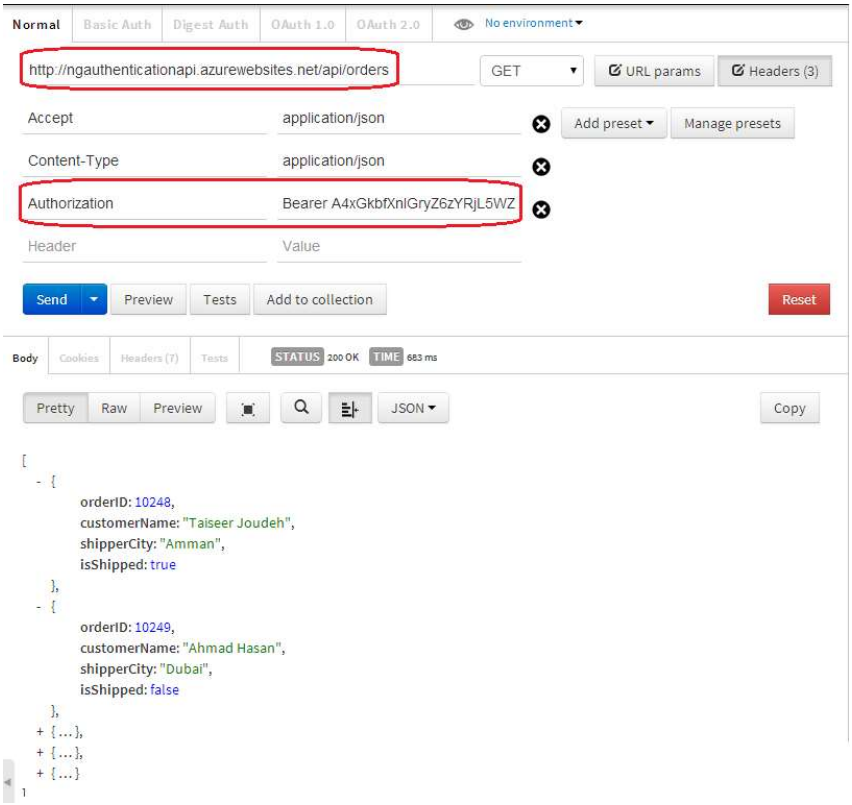

Notice that the content-type and payload type is "x-www-form-urlencoded" so the payload body will be on form (grant_type=password&username="Taiseer"&password="SuperPass"). If all is correct you'll notice that we've received signed token on the response.

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

Integrate Azure AD B2C with ASP.NET MVC Web App – Part 3
Secure ASP.NET Web API 2 using Azure AD B2C – Part 2
Azure Active Directory B2C Overview and Policies Management – Part 1

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

Filed Under: AngularJS, ASP.NET, ASP.NET Identity, ASP.Net Web API, RESTful API, Single Page Applications, Web API Tutorial
Tagged With: AngularJS, ASP.NET, SPA, Token Authentication, Tutorial, Web API

## Comments

Debbs says
November 16, 2017 at 2:54 pm

Simple and on point. Thanks

Reply

Naveen says
November 20, 2017 at 4:13 pm

Hi,
I followed your post and created application, I was able to register with user name and password, but I'm not able to generate token,
When I try to request for token in Postman, I'm getting following jSon data, Please help me to resolve this.

{
"message": "No HTTP resource was found that matches the request URI 'http://localhost:53227/api/token'.",
"messageDetail": "No type was found that matches the controller named 'token'."
}

Reply

Jose Guadalupe Rodriguez Flores says
February 13, 2018 at 8:07 pm

hi, your url is bad, if in startup configure TokenEndpointPath = new PathString("/token") then call http://localhost:53227/token no need "Api"

Reply

nandita says
February 28, 2018 at 11:34 pm

I faced the same issue. And it was due to two (very silly) reason.
1) AllowInsecureHttp = true was missing in the intialization of OAuthorizationServerOption
2) ConfigureoAuth(app) was missing in the Configuration(IAppBuilder app).

Hope it helps.

Reply

Nick says
March 7, 2018 at 1:17 am

You have probably realised by now that you should drop the api. http://localhost:53227/token'

Reply

Vinnie says
March 8, 2018 at 7:58 pm

This localhost:53227 should match with your current iis/express settings. Go to project properties, click on web tab on left and on right side, see whether you have set iis as local iis or iis express and replace your url in the code appropriately.

Hope that solves your issue.

Reply

Xibalba says
March 16, 2018 at 4:32 pm

Drop the api/ from your path.

Reply

Mukul says
April 24, 2018 at 4:43 am

You are trying to hit the wrong URL, it will be only http://localhost:*****/token instead of http://localhost:*****/api/token

Reply

Himanshu says
November 23, 2017 at 2:25 pm

Hi,

Thanks for this great article. Still useful in 2017. Ended up using a few other stack overflow answers to implement OWIN token based authentication for our software that does not use EntityFramework. (y)

Reply

Peter says
December 1, 2017 at 1:06 pm

Hello Taiseer,

Thanks, good article!

Reply

Alex says
December 6, 2017 at 2:06 am

Awesome tutorial!
Thank you for putting this together, this really filled in many holes in my knowledge of WebApi auth and identity.

Reply

Elior says
December 7, 2017 at 12:12 am

Hi,
Thank you very much, your tutorial is a great help for me!
I was able to register a user, but when I try to call the '/token' end point using the following js call, I get back the html of the default file in the 'data' variable ... any suggestions?
Thanks !
Elior

this is the js code:

$.ajax({url: “http://localhost:53118/token/”,data: {“username”:”Taiseer””password”:”SuperPass””grant_type”:”password”},cache: false,type: ‘post’}).done(function (data) {debugger;});

Reply

Elior says
December 7, 2017 at 12:44 pm

ok, it turns out that I should omit the last slash in the url parameter in the ajax call..

$.ajax({url: “http://localhost:53118/token”,data: {“username”:”Taiseer””password”:”SuperPass””grant_type”:”password”},cache: false,type: ‘post’}).done(function (data) {debugger;});

thanks for the great tutorial !

Reply

Tullio Tesorone says
December 10, 2017 at 11:57 am

Hello Taiseer , first chapeau.

Then , in my sample project ( exact duplicate of this ) the register method fails because usermodel is null.
At controller time it is already null.
But i am sure that json body is correctly received on the server side.
Any idea?

Thanks in advance
Tullio

Reply

Vinnie says
March 8, 2018 at 8:01 pm

If you are using postman, check the request params should exactly match with the model variable name, in this case, check the register user model and make sure, both matches.

Hope this solves your problem.

Reply

fabricio says
December 13, 2017 at 9:03 pm

Hi, nice tutorial! One question though, what if I want to add some properties to the user model?
For instance, I want to store address, country or whatever..

Reply

Vinnie says
March 8, 2018 at 8:02 pm

You can add without hesitation, it will work, let me know, in case it doesn't work.

Hope that solves your query.

Reply

Edison says
January 15, 2018 at 12:00 pm

Thank you very much Sir .....:)

Reply

Thierry says
January 21, 2018 at 6:48 am

Hi Taiseer,

Great for this very detailed article. Can't wait to start the next part with AngularJS!! 🙂

Quick question: Is there a way to achieve the same without using Entity Framework? I'm not a fan of it to be honest and prefer creating my own data layer using SqlConnection, etc...

Thanks.

Reply

Vinnie says
March 8, 2018 at 8:03 pm

You can very well replace entity framework with your own service that serves bridge between repository and data layer and it will work flawless. Let me know, in case, of any problems you face.

Hope that answers your query.

Reply

sujith says
February 5, 2018 at 7:50 pm

have one question about this Security token that you described in this lesson . i followed your method and created same security type token in my live project but if you look at the page source you can see the token .any hakers can take my token and skip login page authentication.

Can you please suggest some other approach so i can implemented in my project.

Reply

Vinnie says
March 8, 2018 at 8:05 pm

You can use JWT instead, which is very well described by Taiseer in subsequent tutorials. Let me know, if you need any help.

Hope that solves your query.

Reply

Ishwor says
February 10, 2018 at 5:07 am

I wonder do we need to set Mime type in the server as .json for file extension and application/json for file type because in my case it shows that 406 NotAcceptable and nothing happens.

Reply

Ishwor says
February 10, 2018 at 5:15 pm

Nice article. I tried all your way but used SimpleMemebrshipProvider instead Asp.Net Identity and I was able to access bearer access token using localhost but was unable in DevServer while testing on PostMan. I always get 500 status code. Can you please help me out ? Thank you.

Reply

Ronald Kincaid says
March 7, 2018 at 7:01 am

I have been fighting with this for some time and cannot get the app to run locally. I followed each of your steps and copied your code from the article, but I get a 400 Bad Request after ValidateClientAuthentication is called from authService.js This appears to be CORS-related, but I have verified that the app.UseCors(Microsoft.Owin.Cors.CorsOptions.AllowAll); is in Startup.cs and the context.OwinContext.Response.Headers.Add("Access-Control-Allow-Origin", new[] { "*" }); is in the override GrantResourceOwnerCredentials.

When I debug the code, the app calls the ValidateClientAuthentication method, which has only context.Validated(); in the method. It appears to succeed here, but immediately after, it returns to the client with the 400 message.
There is a second error as follows:

Failed to load http://localhost:46497/token: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:2464' is therefore not allowed access. The response had HTTP status code 400. These errors seem CORS-related, but I have followed your instructions and I have verified that there is no other CORS code in the app, as I have seen that this can occur if one enables CORS in multiple places

Reply

Greg Knierim says
March 15, 2018 at 5:54 pm

I have implemented this and after upgrading to the latest Nuget packages, it works great.

The only question is, is there a way to test the API either by navigating to the URL and entering credentials or from another application (other than the web application) to make sure the responses are as needed?

Thanks,
Greg

Reply

Bp says
May 25, 2018 at 5:44 am

you could use POSTMAN client / fiddler or similar open source tools to test as shown in step 12 by Taiseer

Reply

Alexandre Pereira says
March 20, 2018 at 3:01 pm

public override async Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)
{
context.Validated();
}

I am getting an error:
This async method lacks 'await' operators and will run syncchronously.....

Reply

vvr says
March 24, 2018 at 1:39 pm

Is it possible to use Windows authentication and Authorization using custom tables and then to generate a token?

Reply

Janos says
April 14, 2018 at 11:47 pm

When trying to access the current user's ID by User.Identity.GetUserId() I always get null. Why is that?

Reply

Janos says
April 15, 2018 at 1:59 am

To answer to the questions asked by myself, in SimpleAuthorizationServerProvider.GrantResourceOwnerCredentials I did the following:
public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
{

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

```
using (AuthRepository _repo = new AuthRepository())
{
IdentityUser user = await _repo.FindUser(context.UserName, context.Password);

if (user == null)
{
context.SetError("invalid_grant", "The user name or password is incorrect.");
return;
}
else
{
var identity = _repo.CreateIdentity(user, context.Options.AuthenticationType);
identity.AddClaim(new Claim("sub", context.UserName));
identity.AddClaim(new Claim("role", "user"));

context.Validated(identity);
}
}
}
```

And in AuthRepository:

```
public ClaimsIdentity CreateIdentity(IdentityUser user, string defaultAuthenticationType)
{
var userIdentity = _userManager.CreateIdentity(user, defaultAuthenticationType);

return userIdentity;
}
```

Reply

Harry Vu says
May 5, 2018 at 1:44 am

A great article even to this day (May 2018). Thank you very much.

Reply

Jake says
May 22, 2018 at 10:49 pm

Someoby copied your entire article without any attribution: http://www.geeksblood.com/token-based-authentication-using-asp-net-web-api-2-owin-identity/

Reply

Taiseer Joudeh says
May 23, 2018 at 2:37 pm

Thanks Jake for the heads-up, I do not know what to do with guys who steal other efforts, sad story 😕

Reply

Bp says
May 25, 2018 at 5:40 am

This is really great post. Thanks for breaking this concept.

I implemented this architecture for frontend MVC app (instead of Angular as demoed in this tutorial). It works great

I have a second mvc app which also uses the same web api for the authentication purposes, can SSO (single sign-on) be implemented on these 2 apps with this architecture and be deployed to azure

webapp1.azurewebsites.net -> uses authserver.azurewebsites.net (Web API) for authentication
webapp2.azurewebsites.net -> uses authserver.azurewebsites.net (Web API) for authentication

These both applications have same users and roles. Is SSO Possible?

Any pointers are greatly appreciated

Thanks

Reply

« Older Comments

## Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

POST COMMENT

## ABOUT TAISEER

Husband, Father, Consultant @ MSFT, Life Time Learner...
Read More...

Buy me a coffee

RECENT POSTS

Integrate Azure AD B2C with ASP.NET MVC Web App – Part 3

Secure ASP.NET Web API 2 using Azure AD B2C – Part 2

Azure Active Directory B2C Overview and Policies Management – Part 1

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

## BLOG ARCHIVES

Blog Archives

Select Month

## RECENT POSTS

Integrate Azure AD B2C with ASP.NET MVC Web App – Part 3

Secure ASP.NET Web API 2 using Azure AD B2C – Part 2

Azure Active Directory B2C Overview and Policies Management – Part 1

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

## TAGS

AJAX AngularJS API API Versioning ASP.NET Authentication Autherization Server Azure Active Directory B2C Azure AD B2C 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

## SEARCH

Search this website ...