Table of Contents

[Token based Authentication 1](#_Toc452983439)

[Steps to perform token based authentication: 1](#_Toc452983440)

[Step 1: Installing the needed nuget packages: 1](#_Toc452983441)

[Step 2: Add Provider class: 2](#_Toc452983442)

[Step 3: Add Formatter for token Generation: 3](#_Toc452983443)

[Step 4: Add custom authorization attribute: 4](#_Toc452983444)

[Step 5: Add Owin “Startup” Class and remove Global.asax: 5](#_Toc452983445)

[Workflow for Token Generation: 7](#_Toc452983446)

[Workflow for Token Consumption: 8](#_Toc452983447)

# Token based Authentication

In the token based authentication, 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.

## Steps to perform token based authentication:

So in this post we’ll cover step by step how to create token based authentication.

### Step 1: 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. Add all below packages from package manager console.

1) Install-Package Microsoft.AspNet.WebApi.Owin -Version 5.1.2

2) Install-Package Microsoft.Owin.Host.SystemWeb -Version 2.1.0

To Add support for registering and validating user credentials, include below packages.

3) Install-Package Microsoft.AspNet.Identity.Owin -Version 2.0.1

4) Install-Package Microsoft.AspNet.Identity.EntityFramework -Version 2.0.1

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.

5) Install-Package Microsoft.Owin.Security.OAuth -Version 2.1.0

To enable cross origin, install package

6) Install-Package Microsoft.Owin.Cors -Version 2.1.0

### Step 2: Add Provider class:

Add provider class named “OAuthProvider” which inherits from OAuthAuthorizationServerProvider and override GrantResourceOwnerCredentials method from it which is used to find user and generate token for the logeed in user.

Provider class will look like this

public class OAuthProvider : OAuthAuthorizationServerProvider

{

IUserRepository \_userRepo;

public OAuthProvider(IUserRepository userRepo)

{

\_userRepo = userRepo;

}

public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)

{

User user = \_userRepo.Find(context.UserName, context.Password);

if(user == null)

{

context.SetError("invalid\_grant", "The user name or password is incorrect.");

return;

}

ClaimsIdentity identity = new ClaimsIdentity(context.Options.AuthenticationType);

identity.AddClaim(new Claim(ClaimTypes.Name, context.UserName));

context.Validated(identity);

}

}

### Step 3: Add Formatter for token Generation:

Add JWT Access token Formatter class named “JWTAccessTokenFormater” which implements from ISecureDataFormat<AuthenticationTicket> and Protect method will create token and return it as a string.

Formatter will look like this,

public class JWTAccessTokenFormater : ISecureDataFormat<AuthenticationTicket>

{

private string \_issuer = string.Empty;

public JWTAccessTokenFormater(string issuer)

{

\_issuer = issuer;

}

public string Protect(AuthenticationTicket data)

{

if (data == null)

{

throw new ArgumentNullException("data");

}

//implement token format here.

var token = “Add token here”;

var handler = new JwtSecurityTokenHandler();

var jwt = handler.WriteToken(token);

return jwt;

}

public AuthenticationTicket Unprotect(string protectedText)

{

throw new NotImplementedException();

}

}

### Step 4: Add custom authorization attribute:

Now add custom authorize attribute which inherits from AuthorizeAttribute class and override OnAuthorizationAsync method to create custom authorization.

Custom authorization will look like this,

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, Inherited = true, AllowMultiple = true)]

public class CustomAuthorizeAttribute : System.Web.Http.AuthorizeAttribute

{

public CustomAuthorizeAttribute(params object[] roles)

{

if (roles.Any(r => r.GetType().BaseType != typeof(Enum)))

{

throw new ArgumentException("roles");

}

this.Roles = string.Join(",", roles.Select(r => Enum.GetName(r.GetType(), r)));

}

public override Task OnAuthorizationAsync(HttpActionContext actionContext, CancellationToken cancellationToken)

{

//add custom authorization code here.

return base.OnAuthorizationAsync(actionContext, cancellationToken);

}

}

### Step 5: Add Owin “Startup” Class and remove Global.asax:

Now we to need add new class named “Startup” and all registration, initialization and instance creation code will be added here instead of Global.asax. So, we will remove Global.asax file and now applications startup class will be “Startup.cs”.

We will add “<add key="owin:appStartup" value="BarcoderApp.RestApi.Startup" />” this code to appsettings tag of web.config file which specifies the startup class for the application.

In the startup class we will create instance of HttpConfiguration and use that instance to Configure Autofac, OAuthTokenGeneration, OAuthTokenConsumption, Web api routes and etc.

Demo for startup class is below,

public class Startup

{

public void Configuration(IAppBuilder app)

{

HttpConfiguration httpConfig = new HttpConfiguration();

ConfigureAutofac(app, httpConfig);

ConfigureOAuthTokenGeneration(app, httpConfig);

ConfigureOAuthTokenConsumption(app);

WebApiConfig.Register(httpConfig);

httpConfig.Services.Replace(typeof(IExceptionHandler), httpConfig.DependencyResolver.GetService(typeof(AppExceptionHandler)));

httpConfig.Services.Replace(typeof(IExceptionLogger), httpConfig.DependencyResolver.GetService(typeof(AppExceptionLogger)));

app.UseAutofacWebApi(httpConfig);

app.UseWebApi(httpConfig);

}

}

From above class you will see ConfigureOAuthTokenGeneration method is used to generate token and method will look like this.

private void ConfigureOAuthTokenGeneration(IAppBuilder app, HttpConfiguration config)

{

var OAuthServerOptions = new OAuthAuthorizationServerOptions()

{

AllowInsecureHttp = Convert.ToBoolean(ConfigurationManager.AppSettings["oauth:AllowInsecureHttp"]),

TokenEndpointPath = new Microsoft.Owin.PathString("/oauth/token"),

AccessTokenExpireTimeSpan = TimeSpan.FromMinutes(10),

Provider = (IOAuthAuthorizationServerProvider)config.DependencyResolver.GetService(typeof(OAuthProvider)),

AccessTokenFormat = new JWTAccessTokenFormater(“http://localhost”)

};

app.UseOAuthAuthorizationServer(OAuthServerOptions);

}

The ConfigureOAuthTokenConsumption method is used to consume the token from the request and verify the valid token.

private void ConfigureOAuthTokenConsumption(IAppBuilder app)

{

var issuer = ConfigurationManager.AppSettings["oauth:TokenIssuerUrl"];

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

byte[] audienceSecret = Microsoft.Owin.Security.DataHandler.Encoder.TextEncodings.Base64Url.Decode(ConfigurationManager.AppSettings["jwt:AudienceSecret"]);

// Api controllers with an [Authorize] attribute will be validated with JWT

app.UseJwtBearerAuthentication(new JwtBearerAuthenticationOptions

{

AuthenticationMode = Microsoft.Owin.Security.AuthenticationMode.Active,

AllowedAudiences = new string[] { audienceId },

IssuerSecurityTokenProviders = new IIssuerSecurityTokenProvider[] { new SymmetricKeyIssuerSecurityTokenProvider(issuer, audienceSecret) }

});

}

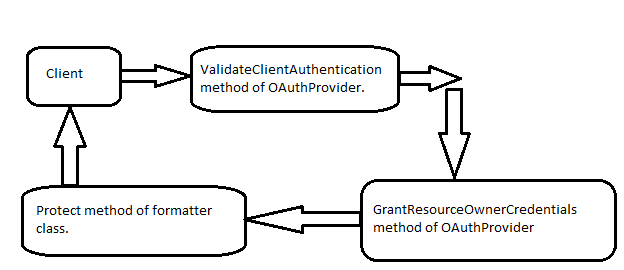
## Workflow for Token Generation:

When client will request for token with credentials, it will pass through ValidateClientAuthentication method which is used to validate the origin of the request is a registered client or not.

When the origin is validated then it will go through GrantResourceOwnerCredentials which is used to verify user credentials and generate token for the valid user.

When user is validated, request will pass now to Protect method of Formatter class which creates JWT token and return generated token to client.

Below is the flow for token generation,



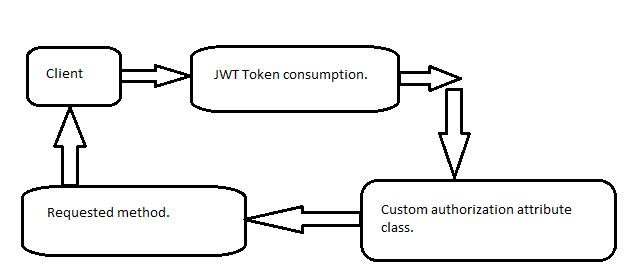
## Workflow for Token Consumption:

When client request with token for any action method, first request will pass through JWT token consumption class which is used to validate token and extract data from token.

When extracting data, it will pass through custom authorization class and it will check the authorization and will return to the requested action method.

When user is authorized, request will pass to requested action method.

Below is the flow for token consumption,



# Refresh Token Mechanism

After implementing token based authentication, now we need to add refresh token functionality.

The idea of using refresh token is to issue short lived access token at the first place then use the refresh token to obtain new access token and so on, so the user needs to authenticate him self by providing username and password along with client info (we’ll talk about clients later in this post), and if the information provided is valid a response contains a short lived access token is obtained along with long lived refresh token (This is not an access token, it is just identifier to the refresh token). Now once the access token expires we can use the refresh token identifier to try to obtain another short lived access token and so on.

## Steps to perform refresh token mechanism:

So in this section we will cover step by step how to add refresh token mechanism.

### Step 1: Add GrantRefreshToken Method in Provider Class:

Override GrantRefreshToken method in the OAuthProvider class which inherits from OAuthAuthorizationServerProvider class. This method is used to find user, create claims and returns token.

GrantRefreshToken method looks like this,

public override async Task GrantRefreshToken(OAuthGrantRefreshTokenContext context)

{

User user = \_userRepo.FindByName(context.Ticket.Identity.Name);

if (user == null)

{

context.SetError("invalid\_grant", "The user not exists.");

return;

}

ClaimsIdentity cliams = new ClaimsIdentity(context.Ticket.Identity);

context.Validated(cliams);

}

### Step 2: Add Refresh token Provider Class:

Add refresh token provider class named “JWTRefreshTokenProvider” which implements from IAuthenticationTokenProvider interface. This implementation contains create and receive functions.

### Step 3: Add Refresh token Provider configuration in Startup class:

Add refresh token provider class configuration in startup class in token generation method like below,

private void ConfigureOAuthTokenGeneration(IAppBuilder app, HttpConfiguration config)

{

var OAuthServerOptions = new OAuthAuthorizationServerOptions()

{

AllowInsecureHttp = Convert.ToBoolean(ConfigurationManager.AppSettings["oauth:AllowInsecureHttp"]),

TokenEndpointPath = new Microsoft.Owin.PathString("/oauth/token"),

AccessTokenExpireTimeSpan = TimeSpan.FromMinutes(Convert.ToInt32(ConfigurationManager.AppSettings["oauth:TokenExpireTime"])),

Provider = (IOAuthAuthorizationServerProvider)config.DependencyResolver.GetService(typeof(OAuthProvider)),

AccessTokenFormat = new JWTAccessTokenFormater(ConfigurationManager.AppSettings["oauth:TokenIssuerUrl"]),

RefreshTokenProvider = new JWTRefreshTokenProvider()

};

app.UseOAuthAuthorizationServer(OAuthServerOptions);

}

## Workflow for refresh token generation:

When client request for a new access token using refresh token, first request will pass through JWT token consumption class which is used to validate token and extract data from token.

After extracting data from jwt token, the request will go to custom authorization class and it will check the authorization.

When access token is expired or authorization fails, request will pass to ReceiveAsync or Receive method of refresh token provider class which checks the expiry, claims and identity of the refresh token.

After checking the refresh token, request will go through GrantRefreshToken method which is used to find user, validate user and gives new access token.

When user is valid it will send to Protect method of formatter class to generate formatted JWT access token.

After new access token generation, request will be passed to CreateAsync method of refresh token provider class which is used to generate new refresh token.

Below is the flow for refresh token generation,

