

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

March 31, 2015 By [Taiseer Joudeh](#) – 245 Comments

Be Sociable, Share!



This is the fifth 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 – Part 3.](#)
- [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 – (This Post)
- [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.

ASP.NET Web API Claims Authorization with ASP.NET Identity 2.1

In the [previous post](#) we have implemented a finer grained way to control authorization based on the Roles assigned for the authenticated user, this was done by assigning users to a predefined Roles in our system and then attributing the protected controllers or actions by the `[Authorize(Roles = "Role(s) Name")]` attribute.

Using Roles Based Authorization for controlling user access will be efficient in scenarios where your Roles do not change too much and the users permissions do not change frequently.

In some applications controlling user access on system resources is more complicated, and having users assigned to certain Roles is not enough for managing user access efficiently, you need more dynamic way to control access based on certain information related to the authenticated user, this will lead us to control user access using **Claims**, or in another word using **Claims Based Authorization**.

But before we dig into the implementation of Claims Based Authorization we need to understand what Claims are!


Note: It is not mandatory to use Claims for controlling user access, if you are happy with Roles Based Authorization and you have limited number of Roles then you can stick to this.

What is a Claim?

Claim is a statement about the user makes about itself, it can be user name, first name, last name, gender, phone, the roles user assigned to, etc... Yes the Roles we have been looking at are transformed to Claims at the end, and as we saw in the previous post; in ASP.NET Identity those Roles have their own manager (ApplicationRoleManager) and set of APIs to manage them, yet you can consider them as a Claim of type Role.

As we saw before, any authenticated user will receive a JSON Web Token (JWT) which contains a set of claims inside it, what we'll do now is to create a helper end point which returns the claims encoded in the JWT for an authenticated user.

To do this we will create a new controller named "ClaimsController" which will contain a single method responsible to unpack the claims in the JWT and return them, to do this add new controller named "ClaimsController" under folder Controllers and paste the code below:



```
1 [RoutePrefix("api/claims")]
2 public class ClaimsController : BaseApiController
3 {
4     [Authorize]
5     [Route("")]
6     public IHttpActionResult GetClaims()
7     {
8         var identity = User.Identity as ClaimsIdentity;
9
10        var claims = from c in identity.Claims
11                      select new
12                      {
13                          subject = c.Subject.Name,
14                          type = c.Type,
15                          value = c.Value
16                      };
17
18        return Ok(claims);
19    }
20
21 }
```

The code we have implemented above is straight forward, we are getting the Identity of the authenticated user by calling "User.Identity" which returns "ClaimsIdentity" object, then we are iterating over the IEnumerable Claims property and return three properties which they are (Subject, Type, and Value). To execute this endpoint we need to issue HTTP GET request to the end point "http://localhost/api/claims" and do not forget to pass a valid JWT in the Authorization header, the response for this end point will

contain the below JSON object:

[Click To Expand Code](#)

As you noticed from the response above, all the claims contain three properties, and those properties represents the below:

- **Subject:** Represents the identity which those claims belongs to, usually the value for the subject will contain the unique identifier for the user in the system (Username or Email).
- **Type:** Represents the type of the information contained in the claim.
- **Value:** Represents the claim value (information) about this claim.

Now to have better understanding of what type of those claims mean let's take a look the table below:

SUBJECT	TYPE	VALUE	NOTES
Hamza	nameidentifier	cd93945e-fe2c-49c1-b2bb-138a2dd52928	Unique User Id generated from Identity System
Hamza	name	Hamza	Unique Username
Hamza	identityprovider	ASP.NET Identity	How user has been authenticated using ASP.NET Identity
Hamza	SecurityStamp	a77594e2-ffa0-41bd-a048-7398c01c8948	Unique Id which stays the same until any security related attribute change, i.e. change user password
Hamza	iss	http://localhost:59822	Issuer of the Access Token (Authz Server)
Hamza	aud	414e1927a3884f68abc79f7283837fd1	For which system this token is generated
Hamza	exp	1427744352	Expiry time for this access token (Epoch)
Hamza	nbf	1427657952	When this token is issued (Epoch)

After we have briefly described what claims are, we want to see how we can use them to manage user assess, in this post I will demonstrate three ways of using the claims as the below:

1. Assigning claims to the user on the fly based on user information.
2. Creating custom Claims Authorization attribute.
3. Managing user claims by using the “ApplicationUserManager” APIs.

Method 1: Assigning claims to the user on the fly

Let's assume a fictional use case where our API will be used in an eCommerce website, where certain users have the ability to issue refunds for orders if there is incident happen and the customer is not happy.

So certain criteria should be met in order to grant our users the privileges to issue refunds, the users should have been working for the company for more than 90 days, and the user should be in “Admin”Role.

To implement this we need to create a new class which will be responsible to read authenticated user information, and based on the information read, it will create a single claim or set of claims and assign then to the user identity.

If you recall from the [first post](#) of this series, we have extended the “ApplicationUser” entity and added a property named “JoinDate” which represent the hiring date of the employee, based on the hiring date, we need to assign a new claim named “FTE” (Full Time Employee) for any user who has worked for more than 90 days. To start implementing this let's add a new class named “ExtendedClaimsProvider” under folder “Infrastructure” and paste the code below:

```
1 public static class ExtendedClaimsProvider
2 {
3     public static IEnumerable<Claim> GetClaims(ApplicationUser user)
4     {
5
6         List<Claim> claims = new List<Claim>();
7
8         var daysInWork = (DateTime.Now.Date - user.JoinDate).TotalDays;
9
10        if (daysInWork > 90)
11        {
12            claims.Add(CreateClaim("FTE", "1"));
13        }
14        else {
15            claims.Add(CreateClaim("FTE", "0"));
16        }
17
18        return claims;
19    }
20
21    public static Claim CreateClaim(string type, string value)
22    {
23        return new Claim(type, value, ClaimValueTypes.String);
24    }
25
26 }
27 }
```

The implementation is simple, the “GetClaims” method will take ApplicationUser object and returns a list of claims. Based on the “JoinDate” field it will add new claim named “FTE” and will assign a value of “1” if the user has been working for than 90 days, and a value of “0” if the user worked for less than this period. Notice how I'm using the method “CreateClaim” which returns a new instance of the claim.

This class can be used to enforce creating custom claims for the user based on the information related to her, you can add as many claims as you want here, but in our case we will add only a single claim.

Now we need to call the method “GetClaims” so the “FTE” claim will be associated with the authenticated user identity, to do this open class “CustomOAuthProvider” and in method “GrantResourceOwnerCredentials” add the highlighted line (line 7) as the code snippet below:

```
1 public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
2 {
3     //Code removed for brevity
4
5     ClaimsIdentity oAuthIdentity = await user.GenerateUserIdentityAsync(userManager, "JWT");
6
7     oAuthIdentity.AddClaims(ExtendedClaimsProvider.GetClaims(user));
8
9     var ticket = new AuthenticationTicket(oAuthIdentity, null);
10
11     context.Validated(ticket);
12
13 }
```

Notice how the established claims identity object “oAuthIdentity” has a method named “AddClaims” which accepts IEnumerable object of claims, now the new “FTE” claim is assigned to the authenticated user, but this is not enough to satisfy the criteria needed to issue the fictitious refund on orders, we need to make sure that the user is in “Admin” Role too.

To implement this we’ll create a new Role on the fly based on the claims assigned for the user, in other words we’ll create Roles from the Claims user assigned to, this Role will be named “IncidentResolvers”. And as we stated in the beginning of this post, the Roles eventually are considered as a Claim of type Role.

To do this add new class named “RolesFromClaims” under folder “Infrastructure” and paste the code below:

```
1 public class RolesFromClaims
2 {
3     public static IEnumerable<Claim> CreateRolesBasedOnClaims(ClaimsIdentity identity)
4     {
5         List<Claim> claims = new List<Claim>();
6
7         if (identity.HasClaim(c => c.Type == "FTE" && c.Value == "1") &&
8             identity.HasClaim(ClaimTypes.Role, "Admin"))
9         {
10             claims.Add(new Claim(ClaimTypes.Role, "IncidentResolvers"));
11         }
12
13         return claims;
14     }
15 }
```

The implementation is self explanatory, we have created a method named “CreateRolesBasedOnClaims” which accepts the established identity object and returns a list of claims.

Inside this method we will check that the established identity for the authenticated user has a claim of type “FTE” with value “1”, as well that the identity contains a claim of type “Role” with value “Admin”, if those 2 conditions are met then; we will create a new claim of Type “Role” and give it a value of “IncidentResolvers”.

Last thing we need to do here is to assign this new set of claims to the established identity, so to do this open class “CustomOAuthProvider” again and in method “GrantResourceOwnerCredentials” add the highlighted line (line 9) as the code snippet below:

```
1 public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
2 {
3     //Code removed for brevity
4
5     ClaimsIdentity oAuthIdentity = await user.GenerateUserIdentityAsync(userManager, "JWT");
6
7     oAuthIdentity.AddClaims(ExtendedClaimsProvider.GetClaims(user));
8
9     oAuthIdentity.AddClaims(RolesFromClaims.CreateRolesBasedOnClaims(oAuthIdentity));
10
11     var ticket = new AuthenticationTicket(oAuthIdentity, null);
12
13     context.Validated(ticket);
14
15 }
```

Now all the new claims which created on the fly are assigned to the established identity and once we call the method “context.Validated(ticket)”, all claims will get encoded in the JWT token, so to test this out let’s add fictitious controller named “OrdersController” under folder “Controllers” as the code below:

```
1 [RoutePrefix("api/orders")]
2 public class OrdersController : ApiController
3 {
4     [Authorize(Roles = "IncidentResolvers")]
5     [HttpPut]
6     [Route("refund/{orderId}")]
7     public IHttpActionResult RefundOrder([FromUri]string orderId)
8     {
9         return Ok();
10    }
11 }
```

Notice how we attribute the action “RefundOrder” with [Authorize(Roles = “IncidentResolvers”)] so only authenticated users with claim of type “Role” and has the value of “IncidentResolvers” can access this end point. To test this out you can issue HTTP PUT request to the URI “http://localhost/api/orders/refund/cxy-4456393” with an empty body.

As you noticed from the first method, we have depended on user information to create claims and kept the authorization more dynamic and flexible.

Keep in mind that you can add your access control business logic, and have finer grained control on authorization by implementing this logic into classes “ExtendedClaimsProvider” and “RolesFromClaims”.

Method 2: Creating custom Claims Authorization attribute

Another way to implement Claims Based Authorization is to create a custom authorization attribute which inherits from “AuthorizationFilterAttribute”, this authorize attribute will check directly the claims value and type for the established identity.

To do this let’s add new class named “ClaimsAuthorizationAttribute” under folder “Infrastructure” and paste the code below:

```
1 public class ClaimsAuthorizationAttribute : AuthorizationFilterAttribute
2 {
3     public string ClaimType { get; set; }
4     public string ClaimValue { get; set; }
5
6     public override Task OnAuthorizationAsync(HttpContext actionContext, System.Threading.CancellationToken
7     {
8         var principal = actionContext.RequestContext.Principal as ClaimsPrincipal;
9
10        if (!principal.Identity.IsAuthenticated)
11        {
12            actionContext.Response = actionContext.Request.CreateResponse(HttpStatusCode.Unauthorized);
13            return Task.FromResult<object>(null);
14        }
15
16        if (!(principal.HasClaim(x => x.Type == ClaimType && x.Value == ClaimValue)))
17        {
18            actionContext.Response = actionContext.Request.CreateResponse(HttpStatusCode.Unauthorized);
19            return Task.FromResult<object>(null);
20        }
21
22        //User is Authorized, complete execution
23        return Task.FromResult<object>(null);
24    }
25 }
26
27 }
```

What we've implemented here is the following:

- Created a new class named "ClaimsAuthorizationAttribute" which inherits from "AuthorizationFilterAttribute" and then override method "OnAuthorizationAsync".
- Defined 2 properties "ClaimType" & "ClaimValue" which will be used as a setters when we use this custom authorize attribute.
- Inside method "OnAuthorizationAsync" we are casting the object "actionContext.RequestContext.Principal" to "ClaimsPrincipal" object and check if the user is authenticated.
- If the user is authenticated we'll look into the claims established for this identity if it has the claim type and claim value.
- If the identity contains the same claim type and value; then we'll consider the request authentic and complete the execution, other wist we'll return 401 unauthorized status.

To test the new custom authorization attribute, we'll add new method to the "OrdersController" as the code below:

```
1 [ClaimsAuthorization(ClaimType="FTE", ClaimValue="1")]
2 [Route("")]
3 public IHttpActionResult Get()
4 {
5     return Ok();
6 }
```

Notice how we decorated the "Get()" method with the "[ClaimsAuthorization(ClaimType="FTE", ClaimValue="1")]" attribute, so any user has the claim "FTE" with value "1" can access this protected end point.

Method 3: Managing user claims by using the “ApplicationUserManager” APIs

The last method we want to explore here is to use the “ApplicationUserManager” claims related API to manage user claims and store them in ASP.NET Identity related tables “AspNetUserClaims”.

In the previous two methods we’ve created claims for the user on the fly, but in method 3 we will see how we can add/remove claims for a certain user.

The “ApplicationUserManager” class comes with a set of predefined APIs which makes dealing and managing claims simple, the APIs that we’ll use in this post are listed in the table below:

METHOD NAME	USAGE
AddClaimAsync(id, claim)	Create a new claim for specified user id
RemoveClaimAsync(id, claim)	Remove claim from specified user if claim type and value match
GetClaimsAsync(id)	Return IEnumerable of claims based on specified user id

To use those APIs let’s add 2 new methods to the “AccountsController”, the first method “AssignClaimsToUser” will be responsible to add new claims for specified user, and the second method “RemoveClaimsFromUser” will remove claims from a specified user as the code below:

```
1 [Authorize(Roles = "Admin")]
2 [Route("user/{id:guid}/assignclaims")]
3 [HttpPost]
4 public async Task<IHttpActionResult> AssignClaimsToUser([FromUri] string id, [FromBody] List<ClaimBindingModel> claims
5
6     if (!ModelState.IsValid)
7     {
8         return BadRequest(ModelState);
9     }
10
11     var appUser = await this.AppUserManager.FindByIdAsync(id);
12
13     if (appUser == null)
14     {
15         return NotFound();
16     }
17
18     foreach (ClaimBindingModel claimModel in claimsToAssign)
19     {
20         if (appUser.Claims.Any(c => c.ClaimType == claimModel.Type)) {
21             await this.AppUserManager.RemoveClaimAsync(id, ExtendedClaimsProvider.CreateClaim(claimModel.Type, claimMo
22         }
23
24         await this.AppUserManager.AddClaimAsync(id, ExtendedClaimsProvider.CreateClaim(claimModel.Type, claimModel.Val
25     }
26
27     return Ok();
28 }
29
30
31 [Authorize(Roles = "Admin")]
32 [Route("user/{id:guid}/removeclaims")]
33 [HttpPost]
34 public async Task<IHttpActionResult> RemoveClaimsFromUser([FromUri] string id, [FromBody] List<ClaimBindingModel> clai
```



```
35 {
36
37     if (!ModelState.IsValid)
38     {
39         return BadRequest(ModelState);
40     }
41
42     var appUser = await this.AppUserManager.FindByIdAsync(id);
43
44     if (appUser == null)
45     {
46         return NotFound();
47     }
48
49     foreach (ClaimBindingModel claimModel in claimsToRemove)
50     {
51         if (appUser.Claims.Any(c => c.ClaimType == claimModel.Type))
52         {
53             await this.AppUserManager.RemoveClaimAsync(id, ExtendedClaimsProvider.CreateClaim(claimModel.Type, claimMo
54         })
55     }
56
57     return Ok();
58 }
```

The implementation for both methods is very identical, as you noticed we are only allowing users in “Admin” role to access those endpoints, then we are specifying the UserId and a list of the claims that will be add or removed for this user.

Then we are making sure that user specified exists in our system before trying to do any operation on the user.

In case we are adding a new claim for the user, we will check if the user has the same claim type before trying to add it, add if it exists before we’ll remove this claim and add it again with the new claim value.

The same applies when we try to remove a claim from the user, notice that methods “AddClaimAsync” and “RemoveClaimAsync” will save the claims permanently in our SQL data-store in table “AspNetUserClaims”.

Do not forget to add the “ClaimBindingModel” under folder “Models” which acts as our POCO class when we are sending the claims from our front-end application, the class will contain the code below:

```
1 public class ClaimBindingModel
2 {
3     [Required]
4     [Display(Name = "Claim Type")]
5     public string Type { get; set; }
6
7     [Required]
8     [Display(Name = "Claim Value")]
9     public string Value { get; set; }
10 }
```

There is no extra steps needed in order to pull those claims from the SQL data-store when establishing the user identity, thanks for the method “CreateIdentityAsync” which is responsible to pull all the claims for the user. We have already implemented this and it can be checked by visiting the [highlighted LOC](#).

To test those methods all you need to do is to issue HTTP PUT request to the URI:

“http://localhost:59822/api/accounts/user/{UserId}/assignclaims” and
“http://localhost:59822/api/accounts/user/{UserId}/removeclaims” as the request images below:

Assign Claims to User

http://localhost:59822/api/accounts/user/f0f8d481-e24c-413a-bf84-a202780f8e50/assignclaims

Accept

application/json

Add preset ▼

Content-Type

application/json

Authorization

Bearer eyJ0eXAiOiJKV1QiLCJhbG

Header

Value

form-data

x-www-form-urlencoded

raw

binary

JSON (application/json) ▼

1

[

2

{

3

{"Type": "Phone", "Value": "96279535581"},

4

{"Type": "Gender", "Value": "Male"}

}

]

Assign Claims

Remove Claims from User

http://localhost:59822/api/accounts/user/f0f8d481-e24c-413a-bf84-a202780f8e50/removeclaims

Accept

application/json

Add prese

Content-Type

application/json

Authorization

Bearer eyJ0eXAiOiJKV1QiLCJhbG

Header

Value

form-data

x-www-form-urlencoded

raw

binary

JSON (application/json) ▼

BIT OF TECHNOLOGY

[ARCHIVE](#)[ABOUT ME](#)[SPEAKING](#)[CONTACT](#)

In the next post we'll build a simple AngularJS application which connects all those posts together, this post should be interesting 😊

The **source code** for this tutorial is available on [GitHub](#).

Follow me on Twitter [@tjoudeh](#)

References

- [An Introduction to Claims – MSDN Article](#)
- [Pro ASP.NET MVC 5 book by Adam Freeman – Chapter 15](#)
- [ASP.NET Web Api and Identity 2.0 – Customizing Identity Models by John Atten](#)
- [Featured Image Credit](#)

Be Sociable, Share!

18





19

**Like this:**

One blogger likes this.

Related Posts

[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](#)[ASP.NET Identity 2.1 with ASP.NET Web API 2.2 \(Accounts Management\) – Part 1](#)[AngularJS Authentication Using Azure Active Directory Authentication Library \(ADAL\)](#)Filed Under: [ASP.NET](#), [ASP.NET Identity](#), [ASP.Net Web API](#), [Uncategorized](#), [Web API Security](#), [Web API Tutorial](#)Tagged With: [Autherization Server](#), [Claims](#), [JWT](#), [OAuth](#), [Token Authentication](#)



Question
1/10
Your score
0

Do You Know Your Computer Viruses?

A chip implanted in a British scientist got a computer virus, making him the first human to be infected by one

TRUE ☒ FALSE ☐

Powered by *Carambola*

1 2 3 4 5 6 7 8 9 10

GREAT THINGS HAPPEN WHEN WE LIVE UNITED.

Ad Council

United Way

Donate or Volunteer

Comments

**Dew says**

January 23, 2016 at 10:38 pm

Hi Taiseer

Great useful and valuable post.

How can I add OrderController with CRUD pattern or Repository pattern?

I try to merge this post with <http://bitoftech.net/2013/11/25/applying-repository-pattern-data-access-layer>

but it not success.

Could you give me some advise?

Thank you,

[Reply](#)



Daniel Gunnarsson (@LoneTruckerG) says

January 31, 2016 at 4:24 pm

Great stuff these articles! Looking forward to more like this!

[Reply](#)



Taiseer Joudeh says

February 3, 2016 at 2:03 am

You are welcome, happy to hear this!

[Reply](#)



Quarenta says

February 11, 2016 at 12:13 am

Hi Taiseer. I've been following your tutorials and i must say i feel like i went back in time to found my favourite teacher... you explain very well.

I leave this reply because i've followed all your steps and everything worked well for me, until i moved the project to another directory. Now seems like my valid JWT token is not being recognized by the Identity middleware.

I tried several times and couldn't get the token to work, so i rolled back the project to the PART 4 version, where only Roles Based is implemented, but still getting the response Unauthorized with content {"message": "Authorization has been denied for this request."} when i try to consume "http://localhost/api/claims".

When i get another token it works fine consuming "http://localhost/api/accounts/users", but at the moment i try to consume "http://localhost/api/claims" and get as response Unauthorized {"message": "Authorization has been denied for this request."}, all following calls to other endpoints get Unauthorized.

Do you know if this is a bug?

[Reply](#)

Quarenta says

February 11, 2016 at 12:16 am

I would like to append that i implemented the ClaimsController, as the beginning of this tutorial.
Thanks.

[Reply](#)

Taiseer Joudeh says

February 11, 2016 at 6:57 pm

Hi, happy to hear that posts were clear to implement.
I do not think there is a bug, but I guess the issuer property is not matching the URL for your issuer after you moved it, try to decode the JWT token using jwt.io and check the claims inside it.

[Reply](#)

Quarenta says

February 11, 2016 at 10:35 pm

Sorry, i forgot to tell that i indeed used JWT.io and iss was ok. By the way i am running the project using inside Visual Studio IIS Express.
Any thoughts why AccountsController accept the JWT and ClaimsController rejects?

[Reply](#)

mvasilchenko says

February 15, 2016 at 9:41 am

Hello Taiseer, Thank you for your work, it really helped me to understand how Authentication works with WebApi.
And I believe not only me waiting for the last post ;).

Wish you inspiration for new articles.

[Reply](#)**Taiseer Joudeh** says

February 16, 2016 at 6:34 am

Thank you for your sweet comment, happy to help 😊

[Reply](#)**Ian** says

February 16, 2016 at 5:35 pm

Hi Taiseer,

thanks very much for this superb series that shows how to build the API for Claims Authorization with ASP.NET Identity. Everything works fine via PostMan, but now I want to be able to create an Asp.Net web solution that will be able to interface with this AspNetIdentity project to enable users to login (and change password etc.) and for me to then be able to control what is displayed in the aspx pages based on the user's roles.

I'm unsure as to how I can interact with the API (such as /oauth/token for example) via the code-behind an aspx login page, and how to get the User object as such, and the user's Roles.

Any help you can provide with this would be much appreciated.

With regards, Ian

[Reply](#)**Taiseer Joudeh** says

February 17, 2016 at 3:32 pm

Hi Ian,

Happy to hear that posts are useful, you need to use an Http Client so start building http requests and communicate with the Api, [this post](#) is really useful and shows how you can authenticate, get data, etc.. Hope this will help!

[Reply](#)



Stefano says

February 17, 2016 at 6:29 pm

Hi, compliments for all 5 posts, very very interesting. but....the last (part. 6)? I don't find it.. 😞

[Reply](#)



Taiseer Joudeh says

February 22, 2016 at 12:40 pm

Sorry about this 😞

[Reply](#)



Ashwin Kumar H says

February 18, 2016 at 4:21 pm

Hi Taiseer,

All the five parts are well explained with the required screenshots. It would be great if you could also show us how to implement the Angular application to consume the API.

Thanks,
Ashwin

[Reply](#)



Taiseer Joudeh says

February 22, 2016 at 12:39 pm

Thank you, I'm working on newest tutorial now to use the new ASP.Net Core 1.0

[Reply](#)



Jeff says

February 21, 2016 at 12:10 am

Hi Taiseer,

Truly this is a great series. I have been able to follow it step by step and indeed learnt a lot. Where I had challenges your responses on the comments came in handy. I have basic idea of how to create a client to communicate with the API but for a better understanding am hoping to see your final post on this.

Thanks a lot.

[Reply](#)



Taiseer Joudeh says

February 22, 2016 at 11:42 am

Hi Jeff, you are most welcome. Happy to hear that posts were useful and easy to understand.

[Reply](#)



Willem Luijk says

March 3, 2016 at 1:56 am

Hi Taiseer,

Is there a chance we will see the AngularJS front end in action with this backend?

[Reply](#)



Taiseer Joudeh says

March 3, 2016 at 1:19 pm

😞 I apologise for not posting this post yet, I will try my best to finalize it soon.

[Reply](#)



Ron says

March 8, 2016 at 8:33 pm

Hey Tarsier, thx for all the work in this very good tutorial. Looking forward to part 6! (can't wait... 😊)

[Reply](#)

Taiseer Joudeh says
March 10, 2016 at 12:02 pm

Hi Ron, you are most welcome. Thanks for your message.

[Reply](#)

Tomasz Jagusz says
March 11, 2016 at 12:48 pm

Hi Taiseer,
I'm reading Your parts of this tutorial and I must say it is amazing. It is complex, but You write about is in such way I can follow easily.
I've noticed that last part is missing, I'd really would like to see how You integrate with Angular.
Hopefully You'll be able to publish it soon 😊

[Reply](#)

Taiseer Joudeh says
March 15, 2016 at 1:05 pm

Hi Tom, thanks for your message I will do my best to publish it soon. Thank you

[Reply](#)

Roj says
March 11, 2016 at 2:18 pm

This tutorial is so great! I am so happy you decided to use ASP.NET core for part 6 since I am building a project using ASP.NET core too.
Are you going to use Angular2 too?

[Reply](#)**Taiseer Joudeh** says

March 15, 2016 at 1:04 pm

Hi Roj, I will be working on another series posts cover asp.net core 1.0 and Angular 2 soon.

[Reply](#)**Brent** says

March 15, 2016 at 11:20 pm

This was an amazing tutorial. I enjoyed it from part 1 to now. Very clear explanations. Keep up the great work!

[Reply](#)**Taiseer Joudeh** says

March 20, 2016 at 10:56 pm

You are welcome Brent, thanks for your message!

[Reply](#)**hisham** says

March 20, 2016 at 8:27 am

thank you,

i followed your post and when deployed webapi to azure i get issue that the webapi not grant the jwt token send in the request header and always give me "Authorization has been denied for this request"

i coded exp = 120 minutes and then i checked login by postman i always get new toke for every post

i don't know the reason for that , any help appreciated.

[Reply](#)



Taiseer Joudeh says

March 21, 2016 at 12:02 am

Check that client id, issuer, and secrets are matching. That is the only reason I can think of.

[Reply](#)



entilzha says

March 23, 2016 at 4:14 am

Is it possible to peak in on the magic? What I mean is I want to find out what happens between the time the http request arrives and when my claims attribute gets a crack at it. In the very first line of `OnAuthorizationAsync` immediately casts "`actionContext.RequestContext.Principal`" to a `ClaimsPrincipal` where we find out whether the user "`IsAuthorized`" and what the claims in the token are. So obviously the token has already been unpacked and the identity of the user has already been confirmed or denied. If the user was denied there is no way for us to learn why. The principal is mostly an empty object with `IsAuthorized` set to false. So, again, I ask...is there any way to see what happens to the token and why it may have resulted in a failure?

I should add I did find out once that this could happen because I had an `audience_id` mismatch. You can't expect the token be decrypted with a different `audience_id` than was used to encrypt it. That is not what is happening to me now and I have no idea how to `_discover_` more information. Any tips would be greatly appreciated! Marcus

[Reply](#)



entilzha says

March 24, 2016 at 12:21 am

Would you say it is not important to be able to see "behind the curtain" to reveal the code that decrypts the token and populates the identity principal? The only way this would not be necessary is if all the problems are traceable to inconsistent "client id, issuer, and secrets"?

[Reply](#)

[« Older Comments](#)

Leave a Reply

Enter your comment here...

ABOUT TAISEER



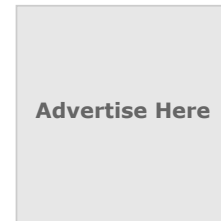
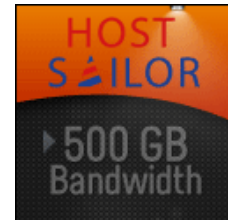
Father, MVP (ASP.NET/IIS), Scrum Master, Life Time Learner

CONNECT WITH ME



Advertise Here

Advertise Here



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

[Blog Archives](#)

LEAVE YOUR EMAIL AND KEEP TUNED!

Sign up to receive email updates on every new post!

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

Copyright © 2016 · eleven40 Pro Theme · Genesis Framework by StudioPress · WordPress · [Log in](#)