

ASP.NET Identity 2.1 with ASP.NET Web API 2.2 (Accounts Management) - Part 1

January 21, 2015 By [Taiseer Joudeh](#) – 148 Comments

Be Sociable, Share!



34



50



40



ASP.NET Identity 2.1 is the latest membership and identity management framework provided by Microsoft, this membership system can be plugged to any ASP.NET framework such as Web API, MVC, Web Forms, etc...

In this tutorial we'll cover how to integrate ASP.NET Identity system with ASP.NET Web API, so we can build a secure HTTP service which acts as back-end for SPA front-end built using AngularJS, I'll try to cover in a simple way different ASP.NET Identity 2.1 features such as: Accounts managements, roles management, email confirmations, change password, roles based authorization, claims based authorization, brute force protection, etc...



The AngularJS front-end application will use **bearer token based authentication** using **Json Web Tokens (JWTs)** format and should support roles based authorization and contains the basic features of any membership system. The SPA is not ready yet but hopefully it will sit on top of our HTTP service without the need to come again and modify the ASP.NET Web API logic.

I will follow step by step approach and I'll start from scratch without using any VS 2013 templates so we'll have better understanding of how the ASP.NET Identity 2.1 framework talks with ASP.NET Web API framework.

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

I broke down this series into multiple posts which I'll be posting gradually, posts are:

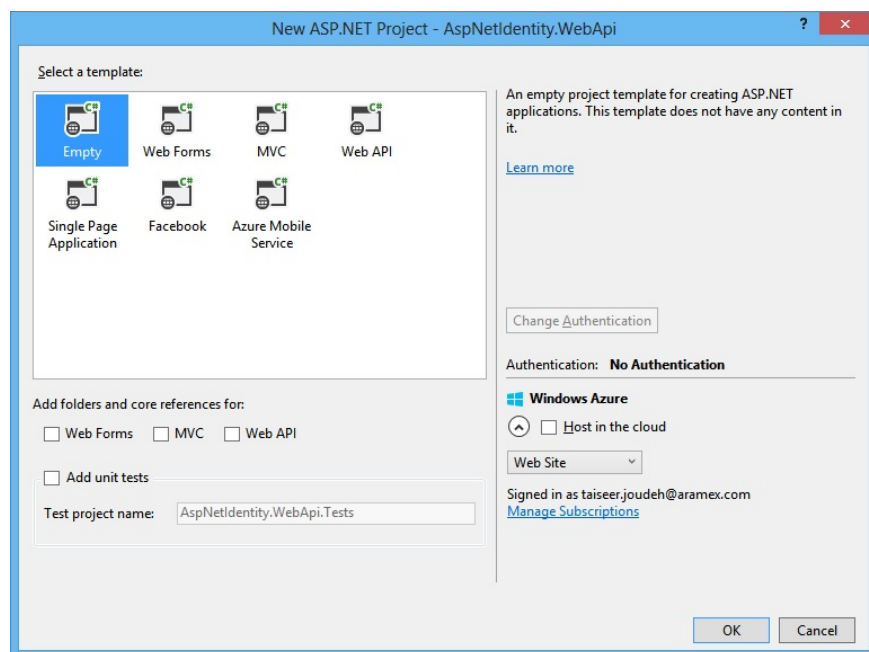
- Configure ASP.NET Identity with ASP.NET Web API (Accounts Management) – (This Post)
- [ASP.NET Identity 2.1 Accounts Confirmation, and Password/User Policy Configuration – Part 2](#)
- [Implement OAuth JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1 – Part 3](#)
- [ASP.NET Identity Role Based Authorization with ASP.NET Web API – Part 4](#)
- [ASP.NET Web API Claims Authorization with ASP.NET Identity 2.1 – Part 5](#)
- [AngularJS Authentication and Authorization with ASP.NET Web API and Identity – Part 6](#)

Configure ASP.NET Identity 2.1 with ASP.NET Web API 2.2 (Accounts Management)

Setting up the ASP.NET Identity 2.1

Step 1: Create the Web API Project

In this tutorial I'm using Visual Studio 2013 and .Net framework 4.5, now create an empty solution and name it "AspNetIdentity" then add new ASP.NET Web application named "AspNetIdentity.WebApi", we will select an empty template with no core dependencies at all, it will be as as the image below:



Step 2: Install the needed NuGet Packages:

We'll install all those NuGet packages to setup our Owin server and configure ASP.NET Web API to be hosted within an Owin server, as well we will install packages needed for ASP.NET Identity 2.1, if you would like to know more about the use of each package and what is the Owin server, please check this [post](#).

```
1 Install-Package Microsoft.AspNet.Identity.Owin -Version 2.1.0
2 Install-Package Microsoft.AspNet.Identity.EntityFramework -Version 2.1.0
```

```
3 Install-Package Microsoft.Owin.Host.SystemWeb -Version 3.0.0
4 Install-Package Microsoft.AspNet.WebApi.Owin -Version 5.2.2
5 Install-Package Microsoft.Owin.Security.OAuth -Version 3.0.0
6 Install-Package Microsoft.Owin.Cors -Version 3.0.0
```

Step 3: Add Application user class and Application Database Context:

Now we want to define our first custom entity framework class which is the “ApplicationUser” class, this class will represents a user wants to register in our membership system, as well we want to extend the default class in order to add application specific data properties for the user, data properties such as: First Name, Last Name, Level, JoinDate. Those properties will be converted to columns in table “AspNetUsers” as we’ll see on the next steps.

So to do this we need to create new class named “ApplicationUser” and derive from “Microsoft.AspNet.Identity.EntityFramework.IdentityUser” class.

Note: If you do not want to add any extra properties to this class, then there is no need to extend the default implementation and derive from “IdentityUser” class.

To do so add new folder named “Infrastructure” to our project then add new class named “ApplicationUser” and paste the code below:

```
1 public class ApplicationUser : IdentityUser
2 {
3     [Required]
4     [MaxLength(100)]
5     public string FirstName { get; set; }
6
7     [Required]
8     [MaxLength(100)]
9     public string LastName { get; set; }
10
11    [Required]
12    public byte Level { get; set; }
13
14    [Required]
15    public DateTime JoinDate { get; set; }
16
17 }
```

Now we need to add Database context class which will be responsible to communicate with our database, so add new class and name it “ApplicationDbContext” under folder “Infrastructure” then paste the code snippet below:

```
1 public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
2 {
3     public ApplicationDbContext()
4         : base("DefaultConnection", throwIfV1Schema: false)
5     {
6         Configuration.ProxyCreationEnabled = false;
7         Configuration.LazyLoadingEnabled = false;
8     }
9
10    public static ApplicationDbContext Create()
11    {
12        return new ApplicationDbContext();
13    }
14 }
```

```
14  
15 }
```

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, this default constructor takes the connection string name “DefaultConnection” as an argument, this connection string will be used point to the right server and database name to connect to.

The static method “Create” will be called from our Owin Startup class, more about this later.

Lastly we need to add a connection string which points to the database that will be created using code first approach, so open “Web.config” file and paste the connection string below:

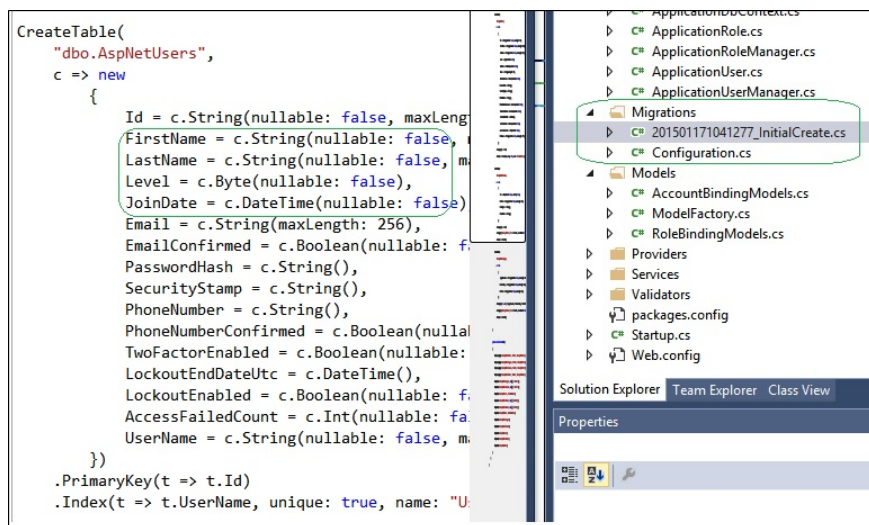
```
1 <connectionStrings>  
2 <add name="DefaultConnection" connectionString="Data Source=.\sqlexpress;Initial Catalog=AspNetIdentity;Integrated  
3 </connectionStrings>
```

Step 4: Create the Database and Enable DB migrations:

Now we want to enable EF code first migration feature which configures the code first to update the database schema instead of dropping and re-creating the database with each change on EF entities, to do so we need to open NuGet Package Manager Console and type the following commands:

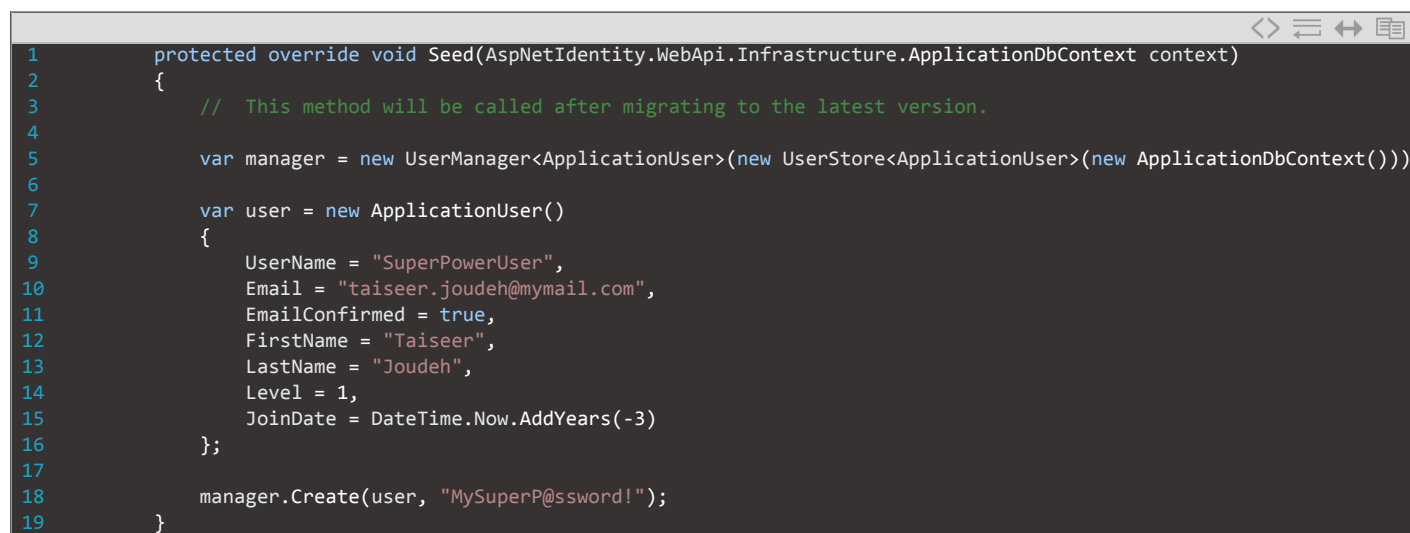
```
1 enable-migrations  
2 add-migration InitialCreate
```

The “enable-migrations” command creates a “Migrations” folder in the “AspNetIdentity.WebApi” project, and it creates a file named “Configuration”, this file contains method named “Seed” which is used to allow us to insert or update test/initial data after code first creates or updates the database. This method is called when the database is created for the first time and every time the database schema is updated after a data model change.



As well the “add-migration InitialCreate” command generates the code that creates the database from scratch. This code is also in the “Migrations” folder, in the file named “<timestamp>_InitialCreate.cs”. The “Up” method of the “InitialCreate” class creates the database tables that correspond to the data model entity sets, and the “Down” method deletes them. So in our case if you opened this class “201501171041277_InitialCreate” you will see the extended data properties we added in the “ApplicationUser” class in method “Up”.

Now back to the “Seed” method in class “Configuration”, open the class and replace the Seed method code with the code below:



```
1  protected override void Seed(AspNetIdentity.WebApi.Infrastructure.ApplicationDbContext context)
2  {
3      // This method will be called after migrating to the latest version.
4
5      var manager = new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new ApplicationDbContext()))
6
7      var user = new ApplicationUser()
8      {
9          UserName = "SuperPowerUser",
10         Email = "taiseer.joudeh@mymail.com",
11         EmailConfirmed = true,
12         FirstName = "Taiseer",
13         LastName = "Joudeh",
14         Level = 1,
15         JoinDate = DateTime.Now.AddYears(-3)
16     };
17
18     manager.Create(user, "MySuperP@ssword!");
19 }
```

This code basically creates a user once the database is created.

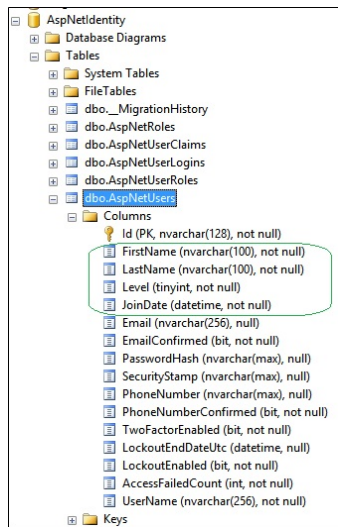
Now we are ready to trigger the event which will create the database on our SQL server based on the connection string we specified earlier, so open NuGet Package Manager Console and type the command:



```
1 update-database
```

The “update-database” command runs the “Up” method in the “Configuration” file and creates the database and then it runs the “Seed” method to populate the database and insert a user.

If all is fine, navigate to your SQL server instance and the database along with the additional fields in table “AspNetUsers” should be created as the image below:



Step 5: Add the User Manager Class:

The User Manager class will be responsible to manage instances of the user class, the class will derive from “`UserManager<T>`” where T will represent our “`ApplicationUser`” class, once it derives from the “`ApplicationUser`” class a set of methods will be available, those methods will facilitate managing users in our Identity system, some of the exposed methods we’ll use from the “`UserManager`” during this tutorial are:

| METHOD NAME | USAGE |
|--|---|
| <code>FindByIdAsync(id)</code> | Find user object based on its unique identifier |
| <code>Users</code> | Returns an enumeration of the users |
| <code>FindByNameAsync(Username)</code> | Find user based on its Username |
| <code>CreateAsync(User, Password)</code> | Creates a new user with a password |
| <code>GenerateEmailConfirmationTokenAsync(Id)</code> | Generate email confirmation token which is used in email confirmation |
| <code>SendEmailAsync(Id, Subject, Body)</code> | Send confirmation email to the newly registered user |
| <code>ConfirmEmailAsync(Id, token)</code> | Confirm the user email based on the received token |
| <code>ChangePasswordAsync(Id, OldPassword, NewPassword)</code> | Change user password |
| <code>DeleteAsync(User)</code> | Delete user |
| <code>IsInRole(Username, Rolename)</code> | Check if a user belongs to certain Role |

AddToRoleAsync(Username, RoleName) Assign user to a specific Role

RemoveFromRoleAsync(Username,
RoleName) Remove user from specific Role

Now to implement the “userManager” class, add new file named “ApplicationUserManager” under folder “Infrastructure” and paste the code below:

```
1  public class ApplicationUser : UserManager<ApplicationUser>
2  {
3      public ApplicationUser(IUserStore<ApplicationUser> store)
4          : base(store)
5      {
6      }
7
8      public static ApplicationUser Create(IdentityFactoryOptions<ApplicationUserManager> options, IOwinContext context)
9      {
10         var appDbContext = context.Get<ApplicationDbContext>();
11         var appUserManager = new ApplicationUser(new UserStore<ApplicationUser>(appDbContext));
12
13         return appUserManager;
14     }
15 }
```

As you notice from the code above the static method “Create” will be responsible to return an instance of the “ApplicationUserManager” class named “appUserManager”, the constructor of the “ApplicationUserManager” expects to receive an instance from the “UserStore”, as well the UserStore instance construct expects to receive an instance from our “ApplicationDbContext” defined earlier, currently we are reading this instance from the Owin context, but we didn’t add it yet to the Owin context, so let’s jump to the next step to add it.

Note: In the coming post we’ll apply different changes to the “ApplicationUserManager” class such as configuring email service, setting user and password policies.

Step 6: Add Owin “Startup” Class

Now we’ll add the Owin “Startup” class which will be fired once our server starts. 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, so add new file named “Startup” to the root of the project and paste the code below:

```
1  public class Startup
2  {
3
4      public void Configuration(IAppBuilder app)
5      {
6          HttpConfiguration httpConfig = new HttpConfiguration();
7
8          ConfigureOAuthTokenGeneration(app);
9
10         ConfigureWebApi(httpConfig);
11
12         app.UseCors(Microsoft.Owin.Cors.CorsOptions.AllowAll);
13
14         app.UseWebApi(httpConfig);
15     }
16 }
```

```
16     }
17
18     private void ConfigureOAuthTokenGeneration(IApplicationBuilder app)
19     {
20         // Configure the db context and user manager to use a single instance per request
21         app.CreatePerOwinContext(ApplicationDbContext.Create);
22         app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);
23
24         // Plugin the OAuth bearer JSON Web Token tokens generation and Consumption will be here
25     }
26
27     private void ConfigureWebApi(HttpConfiguration config)
28     {
29         config.MapHttpAttributeRoutes();
30
31         var jsonFormatter = config.Formatters.OfType<JsonMediaTypeFormatter>().First();
32         jsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();
33     }
34 }
35 }
```

What worth noting here is how we are creating a fresh instance from the “ApplicationDbContext” and “ApplicationUserManager” for each request and set it in the Owin context using the extension method “CreatePerOwinContext”. Both objects (ApplicationDbContext and ApplicationUserManager) will be available during the entire life of the request.

Note: I didn’t plug any kind of authentication here, we’ll visit this class again and add JWT Authentication in the next post, for now we’ll be fine accepting any request from any anonymous users.

Define Web API Controllers and Methods

Step 7: Create the “Accounts” Controller:

Now we’ll add our first controller named “AccountsController” which will be responsible to manage user accounts in our Identity system, to do so add new folder named “Controllers” then add new class named “AccountsController” and paste the code below:

```
1 [RoutePrefix("api/accounts")]
2 public class AccountsController : BaseApiController
3 {
4
5     [Route("users")]
6     public IHttpActionResult GetUsers()
7     {
8         return Ok(this.AppUserManager.Users.ToList().Select(u => this.TheModelFactory.Create(u)));
9     }
10
11     [Route("user/{id:guid}", Name = "GetUserById")]
12     public async Task<IHttpActionResult> GetUser(string Id)
13     {
14         var user = await this.AppUserManager.FindByIdAsync(Id);
15
16         if (user != null)
17         {
18             return Ok(this.TheModelFactory.Create(user));
19         }
20
21         return NotFound();
22     }
23
24
25     [Route("user/{username}")]
26     public async Task<IHttpActionResult> GetUserByName(string username)
```



```
27 {
28     var user = await this.AppUserManager.FindByNameAsync(username);
29
30     if (user != null)
31     {
32         return Ok(this.TheModelFactory.Create(user));
33     }
34
35     return NotFound();
36 }
37 }
38 }
```

What we have implemented above is the following:

- Our “AccountsController” inherits from base controller named “BaseApiController”, this base controller is not created yet, but it contains methods that will be reused among different controllers we’ll add during this tutorial, the methods which comes from “BaseApiController” are: “AppUserManager”, “TheModelFactory”, and “GetErrorResult”, we’ll see the implementation for this class in the next step.
- We have added 3 methods/actions so far in the “AccountsController”:
 - Method “GetUsers” will be responsible to return all the registered users in our system by calling the enumeration “Users” coming from “ApplicationUserManager” class.
 - Method “GetUser” will be responsible to return single user by providing it is unique identifier and calling the method “FindByIdAsync” coming from “ApplicationUserManager” class.
 - Method “GetUserByName” will be responsible to return single user by providing it is username and calling the method “FindByNameAsync” coming from “ApplicationUserManager” class.
 - The three methods send the user object to class named “TheModelFactory”, we’ll see in the next step the benefit of using this pattern to shape the object graph returned and how it will protect us from leaking any sensitive information about the user identity.
- **Note:** All methods can be accessed by any anonymous user, for now we are fine with this, but we’ll manage the access control for each method and who are the authorized identities that can perform those actions in the coming posts.

Step 8: Create the “BaseApiController” Controller:

As we stated before, this “BaseApiController” will act as a base class which other Web API controllers will inherit from, for now it will contain three basic methods, so add new class named “BaseApiController” under folder “Controllers” and paste the code below:

```
1 public class BaseApiController : ApiController
2 {
3
4     private ModelFactory _modelFactory;
5     private ApplicationUserManager _AppUserManager = null;
6
7     protected ApplicationUserManager AppUserManager
8     {
9         get
10         {
11             return _AppUserManager ?? Request.GetOwinContext().GetUserManager<ApplicationUserManager>();
12         }
13     }
14
15     public BaseApiController()
16     {
17     }
```

```
18
19     protected ModelFactory TheModelFactory
20     {
21         get
22         {
23             if (_modelFactory == null)
24             {
25                 _modelFactory = new ModelFactory(this.Request, this.AppUserManager);
26             }
27             return _modelFactory;
28         }
29     }
30
31     protected IHttpActionResult GetErrorResult(IdentityResult result)
32     {
33         if (result == null)
34         {
35             return InternalServerError();
36         }
37
38         if (!result.Succeeded)
39         {
40             if (result.Errors != null)
41             {
42                 foreach (string error in result.Errors)
43                 {
44                     ModelState.AddModelError("", error);
45                 }
46             }
47
48             if (ModelState.IsValid)
49             {
50                 // No ModelState errors are available to send, so just return an empty BadRequest.
51                 return BadRequest();
52             }
53
54             return BadRequest(ModelState);
55         }
56
57         return null;
58     }
59 }
```

What we have implemented above is the following:

- We have added read only property named “AppUserManager” which gets the instance of the “ApplicationUserManager” we already set in the “Startup” class, this instance will be initialized and ready to invoked.
- We have added another read only property named “TheModelFactory” which returns an instance of “ModelFactory” class, this factory pattern will help us in shaping and controlling the response returned to the client, so we will create a simplified model for some of our domain object model (Users, Roles, Claims, etc..) we have in the database. Shaping the response and building customized object graph is very important here; because we do not want to leak sensitive data such as “PasswordHash” to the client.
- We have added a function named “GetErrorResult” which takes “IdentityResult” as a constructor and formats the error messages returned to the client.

Step 8: Create the “ModelFactory” Class:

Now add new folder named “Models” and inside this folder create new class named “ModelFactory”, this class will contain all the functions needed to shape the response object and control the object graph returned to the client, so open the file and paste the code below:

```
1 public class ModelFactory
2 {
3     private UrlHelper _UrlHelper;
4     private ApplicationUserManager _AppUserManager;
5
6     public ModelFactory(HttpRequestMessage request, ApplicationUserManager appUserManager)
7     {
8         _UrlHelper = new UrlHelper(request);
9         _AppUserManager = appUserManager;
10    }
11
12    public UserReturnModel Create(ApplicationUser appUser)
13    {
14        return new UserReturnModel
15        {
16            Url = _UrlHelper.Link("GetUserById", new { id = appUser.Id }),
17            Id = appUser.Id,
18            UserName = appUser.UserName,
19            FullName = string.Format("{0} {1}", appUser.FirstName, appUser.LastName),
20            Email = appUser.Email,
21            EmailConfirmed = appUser.EmailConfirmed,
22            Level = appUser.Level,
23            JoinDate = appUser.JoinDate,
24            Roles = _AppUserManager.GetRolesAsync(appUser.Id).Result,
25            Claims = _AppUserManager.GetClaimsAsync(appUser.Id).Result
26        };
27    }
28 }
29
30 public class UserReturnModel
31 {
32     public string Url { get; set; }
33     public string Id { get; set; }
34     public string UserName { get; set; }
35     public string FullName { get; set; }
36     public string Email { get; set; }
37     public bool EmailConfirmed { get; set; }
38     public int Level { get; set; }
39     public DateTime JoinDate { get; set; }
40     public IList<string> Roles { get; set; }
41     public IList<System.Security.Claims.Claim> Claims { get; set; }
42 }
```

Notice how we included only the properties needed to return them in users object graph, for example there is no need to return the “PasswordHash” property so we didn’t include it.

Step 9: Add Method to Create Users in “AccountsController”:

It is time to add the method which allow us to register/create users in our Identity system, but before adding it, we need to add the request model object which contains the user data which will be sent from the client, so add new file named “AccountBindingModels” under folder “Models” and paste the code below:

```
1 public class CreateUserBindingModel
2 {
3     [Required]
4     [EmailAddress]
5     [Display(Name = "Email")]
6     public string Email { get; set; }
7
8     [Required]
9     [Display(Name = "Username")]
10    public string Username { get; set; }
11
12    [Required]
13    [Display(Name = "First Name")]
14    public string FirstName { get; set; }
15 }
```

```
16 [Required]
17 [Display(Name = "Last Name")]
18 public string LastName { get; set; }
19
20 [Display(Name = "Role Name")]
21 public string RoleName { get; set; }
22
23 [Required]
24 [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
25 [DataType(DataType.Password)]
26 [Display(Name = "Password")]
27 public string Password { get; set; }
28
29 [Required]
30 [DataType(DataType.Password)]
31 [Display(Name = "Confirm password")]
32 [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
33 public string ConfirmPassword { get; set; }
34 }
```

The class is very simple, it contains properties for the fields we want to send from the client to our API with some data annotation attributes which help us to validate the model before submitting it to the database, notice how we added property named “RoleName” which will not be used now, but it will be useful in the coming posts.

Now it is time to add the method which register/creates a user, open the controller named “AccountsController” and add new method named “CreateUser” and paste the code below:

```
1 [Route("create")]
2 public async Task<IHttpActionResult> CreateUser(CreateUserBindingModel createUserModel)
3 {
4     if (!ModelState.IsValid)
5     {
6         return BadRequest(ModelState);
7     }
8
9     var user = new ApplicationUser()
10    {
11        UserName = createUserModel.Username,
12        Email = createUserModel.Email,
13        FirstName = createUserModel.FirstName,
14        LastName = createUserModel.LastName,
15        Level = 3,
16        JoinDate = DateTime.Now.Date,
17    };
18
19    IdentityResult addUserResult = await this.AppUserManager.CreateAsync(user, createUserModel.Password);
20
21    if (!addUserResult.Succeeded)
22    {
23        return GetErrorResult(addUserResult);
24    }
25
26    Uri locationHeader = new Uri(Url.Link("GetUserById", new { id = user.Id }));
27
28    return Created(locationHeader, TheModelFactory.Create(user));
29 }
```

What we have implemented here is the following:

- We validated the request model based on the data annotations we introduced in class “AccountBindingModels”, if there is a field missing then the response will return HTTP 400 with proper error message.
- If the model is valid, we will use it to create new instance of class “ApplicationUser”, by default we’ll put

all the users in level 3.

- Then we call method “CreateAsync” in the “AppUserManager” which will do the heavy lifting for us, inside this method it will validate if the username, email is used before, and if the password matches our policy, etc.. if the request is valid then it will create new user and add to the “AspNetUsers” table and return success result. From this result and as good practice we should return the resource created in the location header and return 201 created status.

Notes:

- Sending a confirmation email for the user, and configuring user and password policy will be covered in the next post.
- As stated earlier, there is no authentication or authorization applied yet, any anonymous user can invoke any available method, but we will cover this authentication and authorization part in the coming posts.

Step 10: Test Methods in “AccountsController”:

Lastly it is time to test the methods added to the API, so fire your favorite REST client Fiddler or PostMan, in my case I prefer PostMan. So lets start testing the “Create” user method, so we need to issue HTTP Post to the URI: “http://localhost:59822/api/accounts/create” as the request below, if creating a user went good you will receive 201 response:

The screenshot shows the Postman interface for a 'Create User' request. The URL is 'http://localhost:59822/api/accounts/create' and the method is 'POST'. The 'Content-Type' and 'Accept' headers are both set to 'application/json'. The 'Body' tab is selected, showing a JSON payload with the following fields: Email, UserName, Password, ConfirmPassword, FirstName, and LastName. The 'JSON (application/json)' format is selected.

```
1 {  
2   "Email": "tayseer_joudeh@msn.com",  
3   "UserName": "tayseer.Joudeh",  
4   "Password": "MySimplePass**",  
5   "ConfirmPassword": "MySimplePass**",  
6   "FirstName": "Tayseer",  
7   "LastName": "Joudeh"  
8 }
```

Now to test the method “GetUsers” all you need to do is to issue HTTP GET to the URI: “http://localhost:59822/api/accounts/users” and the response graph will be as the below:

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

```
8      emailConfirmed: true,
9      "level": 1,
10     "joinDate": "2012-01-17T12:41:40.457",
11     "roles": [
12       "Admin",
13       "Users",
14       "SuperAdmin"
15     ],
16     "claims": [
17       {
18         "issuer": "LOCAL AUTHORITY",
19         "originalIssuer": "LOCAL AUTHORITY",
20         "properties": {},
21         "subject": null,
22         "type": "Phone",
23         "value": "123456782",
24         "valueType": "http://www.w3.org/2001/XMLSchema#string"
25       },
26       {
27         "issuer": "LOCAL AUTHORITY",
28         "originalIssuer": "LOCAL AUTHORITY",
29         "properties": {},
30         "subject": null,
31         "type": "Gender",
32         "value": "Male",
33         "valueType": "http://www.w3.org/2001/XMLSchema#string"
34       }
35     ]
36   },
37   {
38     "url": "http://localhost:59822/api/accounts/user/f0f8d481-e24c-413a-bf84-a202780f8e50",
39     "id": "f0f8d481-e24c-413a-bf84-a202780f8e50",
40     "userName": "tayseer.Joudeh",
41     "fullName": "Tayseer Joudeh",
42     "email": "tayseer_joudeh@hotmail.com",
43     "emailConfirmed": true,
44     "level": 3,
45     "joinDate": "2015-01-17T00:00:00",
46     "roles": [],
47     "claims": []
48   }
49 ]
```

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

In the **next post** we'll see how we'll configure our Identity service to start sending email confirmations, customize username and password polices, implement Json Web Token (JWTs) Authentication and manage the access for the methods.

Follow me on Twitter [@tjoudeh](#)

References

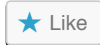
- [Code First Migrations and Deployment with the Entity Framework in an ASP.NET MVC](#)
- [The good, the bad and the ugly of ASP.NET Identity](#)
- [Introduction to ASP.NET Identity](#)

Be Sociable, Share!





Like this:



3 bloggers like this.

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](#)

[AngularJS Authentication Using Azure Active Directory Authentication Library \(ADAL\)](#)

[Two Factor Authentication in ASP.NET Web API & AngularJS using Google Authenticator](#)

[Secure ASP.NET Web API 2 using Azure Active Directory, Owin Middleware, and ADAL](#)

Filed Under: [ASP.NET](#), [ASP.NET Identity](#), [ASP.Net Web API](#), [Web API Tutorial](#)

Tagged With: [Token Authentication](#), [Tutorial](#), [Web API 2](#)

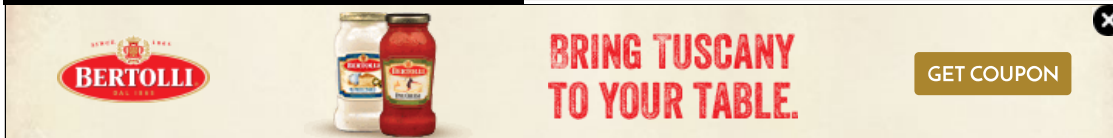


How Does Technology Help Breakups?

The DrunkDial app makes you pass a sobriety test (math problems) before it allows a phone call to the ex

TRUE ☒ FALSE ☐

1 2 3 4 5 6 7 8 9 10



Comments



capesean says

November 4, 2015 at 1:01 am

Hi Taiseer

Have you tried any of this with ASP.NET V5? I'm busy trying to convert my code but am running into issue after issue. I'm wondering if you've tried it yet?

From what I can tell, a lot of the assemblies have changed and are still changing, so maybe it's not a good idea to try using it now. Any thoughts?

Thanks
Sean

[Reply](#)



Taiseer Joudeh says

November 6, 2015 at 3:50 am

Hi,

Do not try, ASP.NET 5 is a totally different platform, many things have changed especially when it comes to authentication and authorization, as well there is no middle ware for generating tokens until now. Web Apis will understand bearer tokens.

If you are referring to ASP.NET identity 3.0 then I didn't play with it yet, so I have no comments yet.

[Reply](#)



Arooran (@Aroor) says

December 10, 2015 at 8:23 pm

Hi Taiseer,

Really thanks for your tutorials. Why didn't you use repository pattern approach for CURD operation in this post? you have implemented previously in Token Authentication post?

[Reply](#)



Taiseer Joudeh says

December 11, 2015 at 9:25 pm

Hi Arooran,

Thanks for your comment, there is really no reason, just for the sake of keeping the post simple, repository pattern is better way to go.

[Reply](#)



dotnetshadow says

December 14, 2015 at 12:43 pm

Hi Taiseer

Thanks for the tutorial...great help!

I've few questions here..

- 1) How can we configure third party caching for tokens ?
- 2) integrate AFDS/LDAP with Identity, Any sample available?

Reply



dotnetshadow says

December 15, 2015 at 12:25 pm

Adding one more question to the above..

How the token validation has been done in the above samples...

my use case is I want to cache the token and validate against to cache data....

Is asp.net Identity by default use cache for token(JWT) for further requests????

Reply



RR says

January 7, 2016 at 4:23 am

Thank you for the post.

In the Seed Method is there a reason for not using the context parameter to add the user to the database ?

Thanks.

Reply



Taiseer Joudeh says

January 10, 2016 at 1:24 am

No reason, you can use the context for seeding too.

Reply



Mark says

January 26, 2016 at 2:49 pm

What if we already have a user database?

Reply



Taiseer Joudeh says

January 27, 2016 at 7:50 am

What do you mean Mark? Can you elaborate more?

Reply



Ravindra says

January 29, 2016 at 6:39 am

Hi Taiseer,

What is the difference in the API between this series and the one in <http://bitoftech.net/2014/06/01/token-based-authentication-asp-net-web-api-2-owin-asp-net-identity/>

Thanks

Reply



stt106 says

January 29, 2016 at 5:39 pm

Hi Taiseer,

Any idea why in the Seed method, when I tried to initialise the database with two users e.g. call `manager.Create(user1, "password1"); manager.Create(user2, "password2");` it somehow only created the first user in the db?

Oh great series of posts by the way; this is exactly what I need though just get started trying myself.

Thanks.

Reply



Taiseer Joudeh says

February 3, 2016 at 2:10 am

Happy to hear that posts were useful, to be honest I have no clue why only one user has been created, did the second one throws an exception or just didn't create it?

Reply



stt106 says

January 30, 2016 at 12:20 am

Another thing is I think you meant to cache the ApplicationUserManager in the basecontroller class e.g. something like

```
protected ApplicationUserManager UserManager
{
    get { return _userManager ?? (_userManager = equest.GetOwinContext().GetUserManager()); }
}
```

Reply



mrtabzify says

February 26, 2016 at 8:04 pm

hi, i made step by step project for web api that you posted in this post, i am not able to hit controller methods in Account controller. do i have to add routing class or anything i am missing?

Reply



Taiseer Joudeh says

March 3, 2016 at 11:57 pm

Hi< please download the Repo from GitHub and compare with your project, there should be something missing or different.

[Reply](#)**Bob Mazzo (@BobMazzo)** says

March 1, 2016 at 7:41 pm

is part 6 available ?

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

March 3, 2016 at 1:21 pm

not yet, sorry about this!

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

March 2, 2016 at 7:21 pm

Namespaces would have been nice. Spent a lot of time looking for the proper namespace for using statements.

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

March 3, 2016 at 1:20 pm

Hi Mike, usually Namespaces will be imported in the top of the file, always refer to the GitHub codebase for complete working example.

[Reply](#)



Joe says

March 9, 2016 at 7:59 pm

Taiseer

Thanks for this tutorial; it's hard to find a good guide that explains all of this without relying on the premade templates in VS.

I noticed the GET request for all users at the end of part 1 lists claims and roles in your example, and the GitHub project has those controllers in place. Did I miss an important step in this part, or are those pieces included later?

Thanks again,

Joe

Reply



Taiseer Joudeh says

March 10, 2016 at 12:01 pm

Hi Joe,

Happy to hear that posts were comprehensive and useful.

Maybe I missed something in the steps, but always refer to the GitHub for the updated code.

Thanks

Reply



Abu Sufyan says

March 11, 2016 at 5:38 pm

Hi,

I had downloaded this project from Github its build successfully, when i send the request from PostMan my breakpoints hit on Account controller CreateUser method by using URL "http://localhost:59822/api/accounts/create", but my CreateUser Method parameter "createUserModel" is null and received following exceptions

```
{  
  "message": "An error has occurred.",  
  "exceptionMessage": "Object reference not set to an instance of an object.",  
  "exceptionType": "System.NullReferenceException",  
  "stackTrace": " at AspNetIdentity.WebApi.Controllers.AccountsController.d__3.MoveNext() in
```

```
C:\AspNetIdentity.WebApi-master\AspNetIdentity.WebApi\Controllers\AccountsController.cs:line 72\r\n
- End of stack trace from previous location where exception was thrown -\r\n at
System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\r\n at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)\r\n at
System.Runtime.CompilerServices.TaskAwaiter1.GetResult()\r\n at
System.Threading.Tasks.TaskHelpersExtensions.d__31.MoveNext()\r\n- End of stack trace from
previous location where exception was thrown -\r\n at
System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\r\n at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)\r\n at
System.Runtime.CompilerServices.TaskAwaiter1.GetResult()\r\n at
System.Web.Http.Controllers.ApiControllerActionInvoker.d__0.MoveNext()\r\n--- End of
stack trace from previous location where exception was thrown ---\r\n at
System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\r\n at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)\r\n at System.Runtime.CompilerServices.TaskAwaiter1.GetResult()\r\n at
System.Web.Http.Controllers.ActionFilterResult.d__2.MoveNext()\r\n- End of stack trace from previous location
where exception was thrown -\r\n at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task
task)\r\n at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)\r\n at System.Runtime.CompilerServices.TaskAwaiter1.GetResult()\r\n at
System.Web.Http.Dispatcher.HttpControllerDispatcher.d__1.MoveNext()
}
```

I did not see any fields on PostMan where i fill data. Is there i missing something?

Reply



Taiseer Joudah says

March 15, 2016 at 1:15 pm

Hi, your request model is empty, check the request fields you are sending and make sure it matches the User Model.

Reply



PANKAJ says

March 21, 2016 at 10:38 pm

Greet post, thanks!

How can I get token from Post? That's why my get call is failing 401

Reply



PANKAJ says

March 21, 2016 at 11:02 pm

Great Post. Thanks

But Get call is failing 401 [Authorization has been denied for this request]. How can get token from post?

Reply



Taiseer Joudeh says

March 23, 2016 at 7:00 pm

You are welcome, I really do not know what si wrong at your end, please download the repo and compare your code with it. It is working perfectly at the repo.

Reply



Mohammed says

March 24, 2016 at 12:29 pm

Greetings,

Thank you very much for this useful tutorial.

I followed this part of your tutorial step-by-step. However, I have tried testing the methods added to the API especially the 'Create' method using PostMan, but creating a user didn't go well. I am getting the following error and I don't know why.

```
{
  "message": "An error has occurred.",
  "exceptionMessage": "Object reference not set to an instance of an object.",
  "exceptionType": "System.NullReferenceException",
  "stackTrace": " at AspNetIdentity.WebApi.Controllers.AccountsController.d__3.MoveNext() in
D:\\Practice\\AspNetIdentity\\AspNetIdentity.WebApi\\AspNetIdentity.WebApi\\Controllers\\AccountsController.cs:line
59\\r\\n- End of stack trace from previous location where exception was thrown -\\r\\n at
System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\\r\\n at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)\\r\\n at
System.Threading.Tasks.TaskHelpersExtensions.d__3`1.MoveNext()\\r\\n- End of stack trace from previous
location where exception was thrown -\\r\\n at
System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\\r\\n at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)\\r\\n at
System.Web.Http.Controllers.ApiControllerActionInvoker.d__0.MoveNext()\\r\\n- End of stack trace from
previous location where exception was thrown -\\r\\n at
```

```
System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\r\n at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)\r\n at
System.Web.Http.Controllers.ActionFilterResult.d__2.MoveNext()\r\n- End of stack trace from previous location
where exception was thrown -\r\n at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task
task)\r\n at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)\r\n at System.Web.Http.Dispatcher.HttpControllerDispatcher.d__1.MoveNext()
}
```

Could you please tell me why I am still getting this error?

[Reply](#)



Taiseer Joudeh says

March 24, 2016 at 12:31 pm

Hi Mohammed,

Put a breakpoint inside the create method and check if the UserModel is filled correctly and it is not null, maybe it is not binding correctly when you set those properties.

[Reply](#)



Mohammed says

March 24, 2016 at 12:30 pm

One more thing, can I use this Web API with my ASP.NET WebForms project? I have an existing project and I would like to include this to it.

[Reply](#)



Taiseer Joudeh says

March 24, 2016 at 12:33 pm

Yup you can, you need to install the needed NuGet packages and you are good to go. Yet I don't recommend you going this path unless there is no other option and you have to mix both projects together.

[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

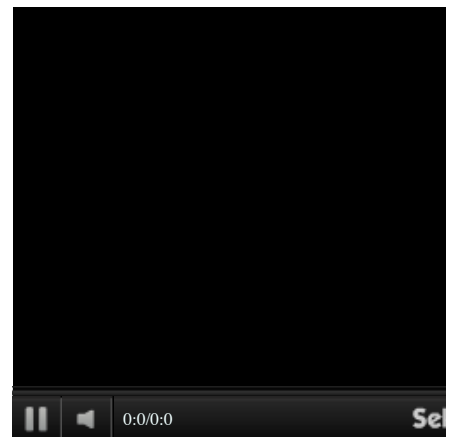


Windows Server workloads meet the cloud.


Experience enhanced performance for enterprise applications.

 Learn more

IBM
SOFT



JavaScript Diagrams



Click Here for

Advertise Here

JavaScript Diagrams



Click here for GoJS

HOST
SAILOR


Manage
Service

Advertise Here

Advertise Here

Windows Server workload
meet the cloud.

Experience enhanced
performance for enterprise
applications.

 Learn more

IBM

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

Select Month 

LEAVE YOUR EMAIL AND KEEP TUNED!

Sign up to receive email updates on every new post!

Email Address

SUBSCRIBE

RECENT POSTS

[ASP.NET Web API Claims Authorization with ASP.NET Identity 2.1 - Part 5](#)

[ASP.NET Identity 2.1 Roles Based Authorization with ASP.NET Web API - Part 4](#)

[Implement OAuth JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1 - Part 3](#)

[ASP.NET Identity 2.1 Accounts Confirmation, and Password Policy Configuration - Part 2](#)

[Interview with John about establishing a successful blog](#)

TAGS

[AJAX](#) [AngularJS](#) [API](#) [API Versioning](#)

[ASP.NET](#) [Attribute Routing](#) [Authentication](#)

[Autherization Server](#) [basic authentication](#) [C#](#) [CacheCow](#)

[Client Side Templating](#) [Code First](#) [Dependency Injection](#)

[Entity Framework](#) [ETag](#) [Foursquare](#) [API](#)

[HTTP Caching](#) [HTTP Verbs](#) [IMDB](#) [API](#) [IoC](#) [Javascript](#) [jQuery](#) [JSON](#)

[JSON Web Tokens](#) [JWT](#) [Model Factory](#) [Ninject](#) [OAuth](#)

[OData](#) [Pagination](#) [Resources Association](#) [Resource Server](#)

[REST](#) [RESTful](#) [Single Page](#)

[Applications](#) [SPA](#) [Token](#) [Authentication](#)

Tutorial Web API Web API 2 Web

API Security Web Service wordpress.com

wordpress.org

CONNECT WITH ME



SEARCH

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