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

January 21, 2015 By Taiseer Joudeh – 102 Comments

Be Sociable, Share!

Share 47 Share 36 Email

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:

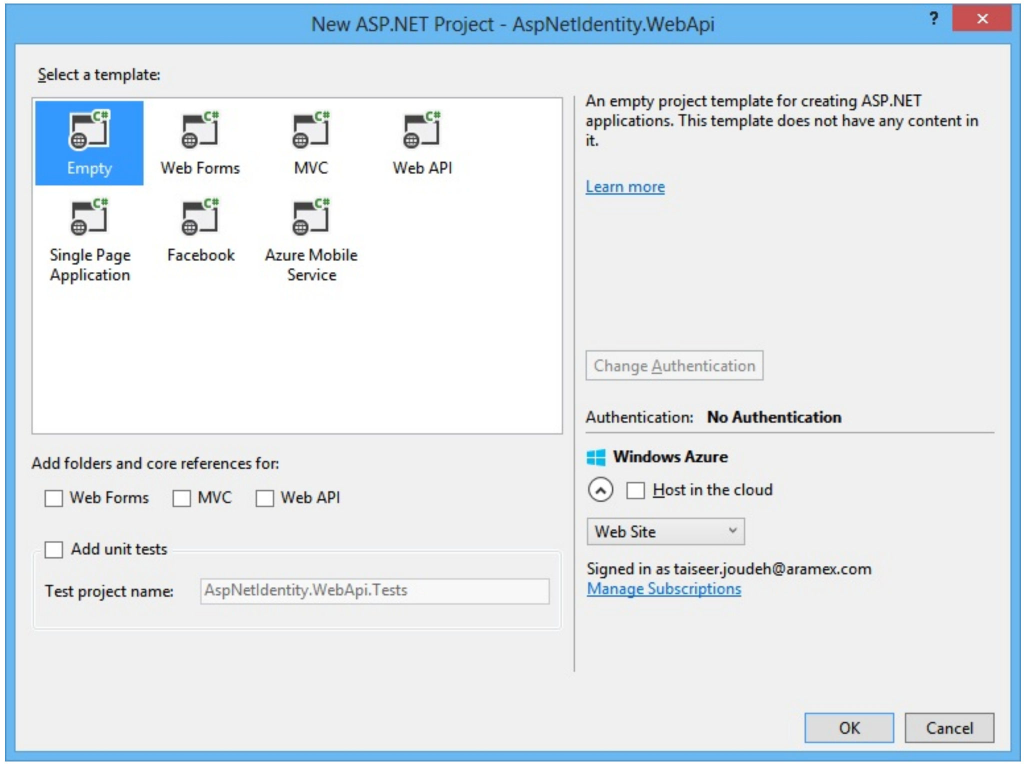
- Configure ASP.NET Identity with ASP.NET Web API (Accounts Management) – (This Post)
- ASP.NET Identity 2.1 Accounts Confirmation, and Password/User Policy Configuration – Part 2
- Implement OAuth JSON Web Tokens Authentication in ASP.NET Web API and Identity 2.1 – Part 3
- ASP.NET Identity Role Based 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:



New ASP.NET Project - AspNetIdentity.WebApi

Select a template: Empty

An empty project template for creating ASP.NET applications. This template does not have any content in it.

Authentication: No Authentication

Add folders and core references for: Web Forms ☐ MVC ☐ Web API ☐

Add unit tests ☐

Test project name: AspNetIdentity.WebApi.Tests

Windows Azure

Host in the cloud ☐

Web Site

Signed in as taiseer.joudeh@aramex.com

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.

```
Install-Package Microsoft.AspNet.Identity.Owin -Version 2.1.0
Install-Package Microsoft.AspNet.Identity.EntityFramework -Version 2.1.0
Install-Package Microsoft.Owin.Host.SystemWeb -Version 3.0.0
Install-Package Microsoft.AspNet.WebApi.Owin -Version 5.2.2
Install-Package Microsoft.Owin.Security.OAuth -Version 3.0.0
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:

```
public class ApplicationUser : IdentityUser
{
    [Required]
    [MaxLength(100)]
    public string FirstName { get; set; }

    [Required]
    [MaxLength(100)]
    public string LastName { get; set; }

    [Required]
    public byte Level { get; set; }

    [Required]
    public DateTime JoinDate { get; set; }

}
```

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:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext()
        : base("DefaultConnection", throwIfV1Schema: false)
    {
        Configuration.ProxyCreationEnabled = false;
        Configuration.LazyLoadingEnabled = false;
    }

    public static ApplicationDbContext Create()
    {
        return new ApplicationDbContext();
    }

}
```

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:

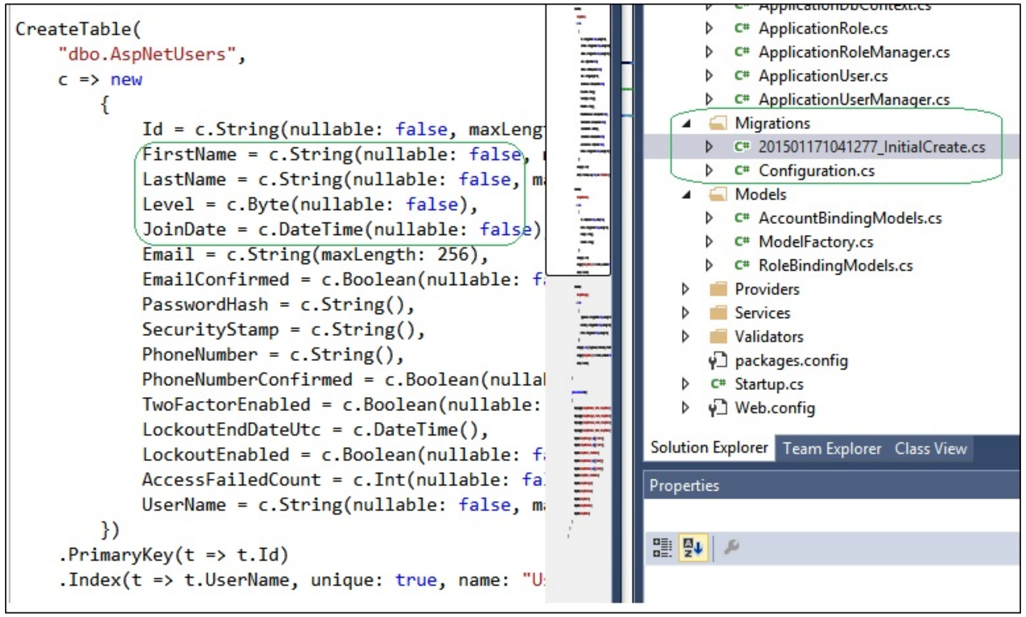
```
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data Source=.\sqlexpress;Initial Catalog=AspNetIdentity;Integrated
</connectionStrings>
```

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:

```
enable-migrations
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.



```
CreateTable(
    "dbo.AspNetUsers",
    c => new
        {
            Id = c.String(nullable: false, maxLeng
            FirstName = c.String(nullable: false,
            LastName = c.String(nullable: false, m
            Level = c.Byte(nullable: false),
            JoinDate = c.DateTime(nullable: false)
            Email = c.String(maxLength: 256),
            EmailConfirmed = c.Boolean(nullable: f
            PasswordHash = c.String(),
            SecurityStamp = c.String(),
            PhoneNumber = c.String(),
            PhoneNumberConfirmed = c.Boolean(nullal
            TwoFactorEnabled = c.Boolean(nullable:
            LockoutEndDateUtc = c.DateTime(),
            LockoutEnabled = c.Boolean(nullable: f
            AccessFailedCount = c.Int(nullable: fa
            UserName = c.String(nullable: false, m
        })
    .PrimaryKey(t => t.Id)
    .Index(t => t.UserName, unique: true, name: "U
```

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”

ɔde

```
        protected override void Seed(AspNetIdentity.WebApi.Infrastructure.ApplicationDbContext context)
        {
            //  This method will be called after migrating to the latest version.

            var manager = new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new ApplicationDbContext()))

            var user = new ApplicationUser()
            {
                UserName = "SuperPowerUser",
                Email = "taiseer.joudeh@mymail.com",
                EmailConfirmed = true,
                FirstName = "Taiseer",
                LastName = "Joudeh",
                Level = 1,
                JoinDate = DateTime.Now.AddYears(-3)
            };

            manager.Create(user, "MySuperP@ssword!");
        }
```

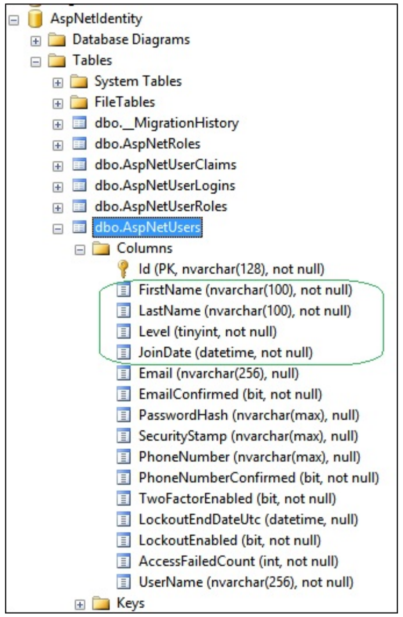
This code basically creates a user once the database is created.

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:

```
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:


```
AspNetIdentity
  Database Diagrams
  Tables
    System Tables
    FileTables
    dbo.__MigrationHistory
    dbo.AspNetRoles
    dbo.AspNetUserClaims
    dbo.AspNetUserLogins
    dbo.AspNetUserRoles
    dbo.AspNetUsers
      Columns
        Id (PK, nvarchar(128), not null)
        FirstName (nvarchar(100), not null)
        LastName (nvarchar(100), not null)
        Level (tinyint, not null)
        JoinDate (datetime, not null)
        Email (nvarchar(256), null)
        EmailConfirmed (bit, not null)
        PasswordHash (nvarchar(max), null)
        SecurityStamp (nvarchar(max), null)
        PhoneNumber (nvarchar(max), null)
        PhoneNumberConfirmed (bit, not null)
        TwoFactorEnabled (bit, not null)
        LockoutEndDateUtc (datetime, null)
        LockoutEnabled (bit, not null)
        AccessFailedCount (int, not null)
        UserName (nvarchar(256), not null)
      Keys
```

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
FindByIdAsync(id)	Find user object based on its unique identifier
Users	Returns an enumeration of the users
FindByNameAsync(Username)	Find user based on its Username

CreateAsync(User, Password	Creates a new user with a password
GenerateEmailConfirmationTokenAsync(Id)	Generate email confirmation token which is used in email confimration
SendEmailAsync(Id, Subject, Body)	Send confirmation email to the newly registered user
ConfirmEmailAsync(Id, token)	Confirm the user email based on the received token
ChangePasswordAsync(Id, OldPassword, NewPassword)	Change user password
DeleteAsync(User)	Delete user
IsInRole(Username, Rolename)	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:

```
    public class ApplicationUserManager : UserManager<ApplicationUser>
    {
        public ApplicationUserManager(IUserStore<ApplicationUser> store)
```

```
            : base(store)
        {
        }

        public static ApplicationUserManager Create(IdentityFactoryOptions<ApplicationUserManager> options, IOwinConte
        {
            var appDbContext = context.Get<ApplicationDbContext>();
            var appUserManager = new ApplicationUserManager(new UserStore<ApplicationUser>(appDbContext));

            return appUserManager;
        }
    }
```

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 polices.

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:

```
    public class Startup
    {

        public void Configuration(IAppBuilder app)
        {
            HttpConfiguration httpConfig = new HttpConfiguration();

            ConfigureOAuthTokenGeneration(app);

            ConfigureWebApi(httpConfig);

            app.UseCors(Microsoft.Owin.Cors.CorsOptions.AllowAll);

            app.UseWebApi(httpConfig);

        }

        private void ConfigureOAuthTokenGeneration(IAppBuilder app)
        {
            // Configure the db context and user manager to use a single instance per request
            app.CreatePerOwinContext(ApplicationDbContext.Create);
            app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);

        // Plugin the OAuth bearer JSON Web Token tokens generation and Consumption will be here

        }

        private void ConfigureWebApi(HttpConfiguration config)
        {
            config.MapHttpAttributeRoutes();

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

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 AplicationUserManager) 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:

```
[RoutePrefix("api/accounts")]
public class AccountsController : BaseApiController
{

    [Route("users")]
    public IHttpActionResult GetUsers()
    {
        return Ok(this.AppUserManager.Users.ToList().Select(u => this.TheModelFactory.Create(u)));
    }

    [Route("user/{id:guid}", Name = "GetUserById")]
    public async Task<IHttpActionResult> GetUser(string Id)
    {
        var user = await this.AppUserManager.FindByIdAsync(Id);

        if (user != null)
        {
            return Ok(this.TheModelFactory.Create(user));
        }

        return NotFound();

    }

    [Route("user/{username}")]
    public async Task<IHttpActionResult> GetUserByName(string username)
    {
        var user = await this.AppUserManager.FindByNameAsync(username);

        if (user != null)
        {
            return Ok(this.TheModelFactory.Create(user));
        }

        return NotFound();

    }
}
```

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:

```
    public class BaseApiController : ApiController
    {

        private ModelFactory _modelFactory;
        private ApplicationUserManager _AppUserManager = null;

        protected ApplicationUserManager AppUserManager
        {
            get
            {
                return _AppUserManager ?? Request.GetOwinContext().GetUserManager<ApplicationUserManager>();
            }
        }

        public BaseApiController()
        {
        }

        protected ModelFactory TheModelFactory
        {
            get
            {
                if (_modelFactory == null)
                {
                    _modelFactory = new ModelFactory(this.Request, this.AppUserManager);
                }
                return _modelFactory;
            }
        }

        protected 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;
        }
    }
```

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:

```
public class ModelFactory
{
    private UrlHelper _UrlHelper;
    private ApplicationUserManager _AppUserManager;

    public ModelFactory(HttpRequestMessage request, ApplicationUserManager appUserManager)
    {
        _UrlHelper = new UrlHelper(request);
        _AppUserManager = appUserManager;
    }

    public UserReturnModel Create(ApplicationUser appUser)
    {
        return new UserReturnModel
        {
            Url = _UrlHelper.Link("GetUserById", new { id = appUser.Id }),
            Id = appUser.Id,
            UserName = appUser.UserName,
            FullName = string.Format("{0} {1}", appUser.FirstName, appUser.LastName),
            Email = appUser.Email,
            EmailConfirmed = appUser.EmailConfirmed,
            Level = appUser.Level,
            JoinDate = appUser.JoinDate,
            Roles = _AppUserManager.GetRolesAsync(appUser.Id).Result,
            Claims = _AppUserManager.GetClaimsAsync(appUser.Id).Result
        };
    }
}

public class UserReturnModel
{
    public string Url { get; set; }
    public string Id { get; set; }
    public string UserName { get; set; }
    public string FullName { get; set; }
    public string Email { get; set; }
    public bool EmailConfirmed { get; set; }
    public int Level { get; set; }
    public DateTime JoinDate { get; set; }
    public IList<string> Roles { get; set; }
```

```
        public IList<System.Security.Claims.Claim> Claims { get; set; }
    }
```

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:

```
    public class CreateUserBindingModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        public string Email { get; set; }

        [Required]
        [Display(Name = "Username")]
        public string Username { get; set; }

        [Required]
        [Display(Name = "First Name")]
        public string FirstName { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }

        [Display(Name = "Role Name")]
        public string RoleName { 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; }

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

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:

```
[Route("create")]
public async Task<IHttpActionResult> CreateUser(CreateUserBindingModel createUserModel)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var user = new ApplicationUser()
    {
        UserName = createUserModel.Username,
        Email = createUserModel.Email,
        FirstName = createUserModel.FirstName,
        LastName = createUserModel.LastName,
        Level = 3,
        JoinDate = DateTime.Now.Date,
    };

    IdentityResult addUserResult = await this.AppUserManager.CreateAsync(user, createUserModel.Password);

    if (!addUserResult.Succeeded)
    {
        return GetErrorResult(addUserResult);
    }

    Uri locationHeader = new Uri(Url.Link("GetUserById", new { id = user.Id }));

    return Created(locationHeader, TheModelFactory.Create(user));
}
```

What we have implemented here is the following:

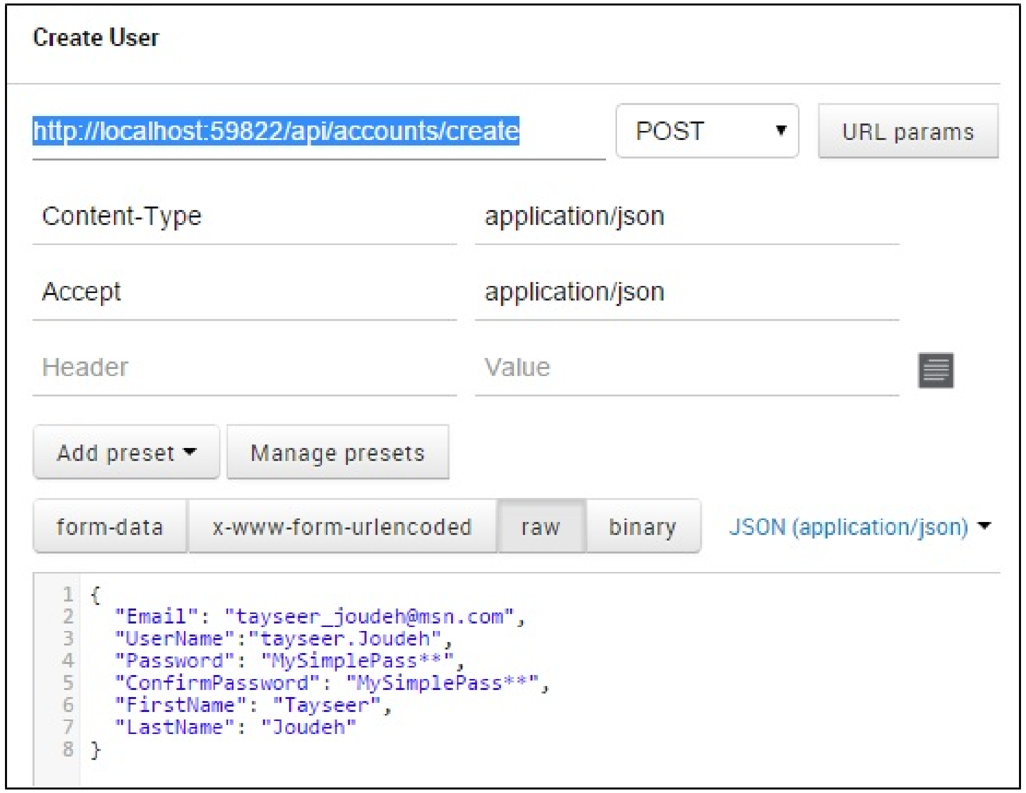
- We validated the request model based on the data annotations we introduced in class "AccountBindingModels", if there is a field missing then the response will return HTTP 400 with proper error message.
- If the model is valid, we will use it to create new instance of class "ApplicationUser", by default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



POST http://localhost:59822/api/accounts/create

Content-Type	application/json
Accept	application/json
Header	Value

```
{
  "Email": "tayseer_joudeh@msn.com",
  "UserName":"tayseer.Joudeh",
  "Password": "MySimplePass**",
  "ConfirmPassword": "MySimplePass**",
  "FirstName": "Tayseer",
  "LastName": "Joudeh"
}
```

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:

```
[
  {
    "url": "http://localhost:59822/api/accounts/user/29e21f3d-08e0-49b5-b523-3d68cf623fd5",
    "id": "29e21f3d-08e0-49b5-b523-3d68cf623fd5",
    "userName": "SuperPowerUser",
    "fullName": "Taiseer Joudeh",
    "email": "taiseer.joudeh@gmail.com",
    "emailConfirmed": true,
    "level": 1,
    "joinDate": "2012-01-17T12:41:40.457",
    "roles": [
      "Admin",
      "Users",
      "SuperAdmin"
    ],
    "claims": [
      {
        "issuer": "LOCAL AUTHORITY",
        "originalIssuer": "LOCAL AUTHORITY",
        "properties": {},
        "subject": null,
        "type": "Phone",
        "value": "123456782",
        "valueType": "http://www.w3.org/2001/XMLSchema#string"
      },
      {
        "issuer": "LOCAL AUTHORITY",
        "originalIssuer": "LOCAL AUTHORITY",
        "properties": {},
        "subject": null,
        "type": "Gender",
        "value": "Male",
        "valueType": "http://www.w3.org/2001/XMLSchema#string"
      }
    ]
  },
  {
    "url": "http://localhost:59822/api/accounts/user/f0f8d481-e24c-413a-bf84-a202780f8e50",
    "id": "f0f8d481-e24c-413a-bf84-a202780f8e50",
    "userName": "tayseer.Joudeh",
    "fullName": "Tayseer Joudeh",
    "email": "tayseer_joudeh@hotmail.com",
    "emailConfirmed": true,
    "level": 3,
    "joinDate": "2015-01-17T00:00:00",
    "roles": [],
    "claims": []
  }
]
```

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!

Share 47 Share 36 Email

Like this:

Loading...

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

Comments

Omar Qadan (@omarqdev) says
January 22, 2015 at 11:08 am

great post Taiseer.

i was trying to find a way for how to make aspnet identity integrate with my own database schema(to follow the DB team standards) and i actually hit the wall hard , now finally they released the code of this library ! here https://github.com/aspnet/Identity/.

Reply

Taiseer Joudeh says
January 24, 2015 at 1:15 am

Thanks Omar for your comment, I agree with you it was hard to integrate ASP.NET Identity with existing database.

Reply

Walter says
January 22, 2015 at 1:25 pm

This is excellent Taiseer. Keep up the great work. I can't wait to see the complete end-to-end with the AngularJS UI. Are you planning to illustrate with a mini CRUD app? Any plans on when your series will be complete? Thanks again. Kind Regards. Walter.

Reply

Taiseer Joudeh says
January 24, 2015 at 1:13 am

Hi Walter, well I'll try to blog 2/3 posts per month, so maybe by the end of Feb , hopefully 🙂 Yes it will be simple membership system.

Reply

vjenks says
January 26, 2015 at 7:13 pm

This is great, thank you! I only wish the entire tutorial was complete. I'm just starting out on a new project and this was exactly the approach I was hoping to use. More, more, please!

Reply

Taiseer Joudeh says
January 27, 2015 at 8:38 pm

Next post will be on Monday, thanks and keep tuned.

Reply

Walter says
January 28, 2015 at 12:17 pm

Excellent.

Reply

Andrew Ball says
January 30, 2015 at 8:27 am

Hi Taisser

Would it be possible to add prompting a user to change password on first login to your series. I implement auto registration
of accounts with a default password and want users to change to their own personalized credential when they first use the SPA. If not, what would be the best practice to add this functionality with ASPNET identity/OWIN.

Thanks so much for your clear content inspirational blogs. They are a pleasure to go through.Just need the light bulb to go off on the above and I'll feel in control for the first time 🙂 Mostly due to your blogs.

Cheers

Reply

Taiseer Joudeh says
January 30, 2015 at 1:27 pm

Hi Andrew,
Really happy to hear that posts are useful I'm currently writing the part 2, and the use case you asked for is not covered yet, but I guess it can be added easily, let me think about it, and if I didn't add it to the post we can take this offline later on. So Keep tuned!

Reply

Andrew Ball says
January 30, 2015 at 11:47 pm

Thanks Taisser. Certainly will keep tuned.

Cheers

Reply

keke says
February 7, 2015 at 1:24 pm

Can we consume that api from Android and iOS mobile applications too ?

Thank you!

Reply

Taiseer Joudeh says
February 7, 2015 at 2:48 pm

Absolutely, you can use this HTTP Service with any front-end client able to send HTTP requests.

Reply

keke says
February 7, 2015 at 9:01 pm

Thank you very much !

I 've already integrated User Authentication from your articles. Looking forward to Role Based Authorization. These are really helpful articles.

Very understandable , clear , easy and very useful.

Cheers ! Will keep tuned too:)

Reply

Bilal Pakka Insafian says
February 7, 2015 at 6:58 pm

Salaam Taiseer, thank you for the post, I am learning all this websites stuff 🙂 – just so I am clear model factory acts similar to Auto mapper for mapping Model to DTOs?

Reply

Taiseer Joudeh says
February 7, 2015 at 9:04 pm

Salam Bilal,
Thanks for your message, and yes you can use Auto Mapper instead of model factory, but I prefer this pattern which gives me more control over DTOs returned.

Reply

Bilal Pakka Insafian says

February 9, 2015 at 7:05 pm

Thank you 🙂

Reply

Daniël says
February 22, 2015 at 12:07 am

Hi, I'm new to your blog and it seems to be that there are two series of posts about the same topic (Asp.net Identity with AngularJS). Could you please explain the differences between the series? Could I implement the solution in your first series or should I wait for this one to be finished? Thanks a lot in advance!

Reply

Taiseer Joudeh says
February 22, 2015 at 12:43 pm

Hi, yes you can depend on the first tutorial, this one I'm building shows the features of ASP.NET Identity and covers different types of Authorization such as Roles and Claims.

Reply

pushpendra says
April 5, 2015 at 10:15 am

Hi
Taiseer, the blog is very useful to me , I also wanted to know the difference between the first series and this one ? Both are token based and both use OWIN.

Reply

Taiseer Joudeh says
April 5, 2015 at 10:29 am

Great to hear it is useful, this one focus more on Asp.Net Identity features and APIs.

Reply

mahavir Jadhav says
March 4, 2015 at 9:05 am

Hi Taiseer,
Please can you give example of Database first approch.

Reply

Taiseer Joudeh says
March 5, 2015 at 10:14 pm

Hi Mahavir,
You mean adding Asp.Net Identity to existing database? You can check this post. I do not have something written for this usecase, still you can use Asp.Net Identity with existing database using Code First approach.

Reply

neil says
March 6, 2015 at 2:29 pm

Great tutorial. However, I did run into a problem

Error 1 'System.Net.Http.HttpRequestMessage' does not contain a definition for 'GetOwinContext' and no extension method 'GetOwinContext' accepting a first argument of type 'System.Net.Http.HttpRequestMessage' could be found (are you missing a using directive or an assembly reference?) C:\Projects\AspNetIdentity\AaspNetIdentity.WebApi\Controllers\BaseApiController.cs 20 51 AaspNetIdentity.WebApi

Reply

neil says
March 6, 2015 at 3:46 pm

Found the reason: I had to reference System.Net.Http and the System.Web.Http from Microsoft.AspNet.WebApi.Owin

Reply

Taiseer Joudeh says
March 7, 2015 at 12:42 am

Glad that it worked 😀

Reply

neil says
March 10, 2015 at 12:12 pm

Unfortunately it does not 🙁

Reply

Jacob Cheriathundam says
March 28, 2015 at 7:23 pm

Try changing "Request" to "HttpContext.Current".

Also note, that if you are using WebApi2.2, the location of the extension method "GetUserManager" has changed to: Microsoft.AspNet.Identity.Owin library.

Blerim Jegeni says
April 4, 2015 at 3:34 pm

I had the same issue and resolved it using these as using statements:

using AspNetIdentity.Infrastructure;
using System.Web;
using System.Web.Http;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using AspNetIdentity.Models;

Marcin says
April 5, 2015 at 4:34 pm

Hi! Great post. Unfortunately I'm facing the same issue with missing definition for GetOwinContext(). Would you have a solution for this?

Ben says
April 26, 2015 at 12:47 pm

The source code repo has these in the following using statements.
using AspNetIdentity.WebApi.Infrastructure;
using AspNetIdentity.WebApi.Models;
using Microsoft.AspNet.Identity;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Net.Http;
using Microsoft.AspNet.Identity.Owin;

Johan says

March 12, 2015 at 12:14 pm

Hi, I've followed this post (using visual studio 2013) but cannot get the IIS express to find the controllers. I keep getting a 404 when I try to do a GET with postman. http://localhost:63486/api/accounts/users. I have configured application startup to be this project in my solution and I am running the web api application on http://localhost:63486. My code is identical to yours except that I've mapped my users db-table to another table than AspnetUsers.

How should I be hosting owin in IIS, I guess following the steps mentioned above should be good enough or am I missing something?

Reply

groovycoder says

April 5, 2015 at 10:26 pm

Hey, I had the same problem. You probably forgot to add [RoutePrefix("api/accounts")] attribute above the AccountsController.

Reply

kjsmith3 says
March 15, 2015 at 1:57 am

What a great post!

Dying to see the AngularJS post

Reply

Zaman says
March 15, 2015 at 2:51 pm

Thanks for this great article Taiseer…
Can i use (LocalDb)\v 11.0 instead of default <add name="DefaultConnection" connectionString="Data Source=.\sqlexpress;Initial Catalog=AspNetIdentity; in web.config.
I actually tried to change the data source to localdb but i dont know why database cant be created with this technique..Could you please help me on that?

Reply

Taiseer Joudeh says
March 16, 2015 at 5:10 pm

Hi, it should work without issues, please check this URL and double check that you are not receiving any errors on Package Manager Console when you type “update-database”

Reply

E. Timothy Uy says
March 17, 2015 at 4:39 am

These posts are fantastic, as are the first series on this topic. In this particular one, my last test did not reveal that my power user had any claims, unlike the example.

Reply

francescoimbesi says
March 20, 2015 at 1:24 pm

hi

I tried yuor project but don’t work..

I did this :

Download project
Build it and publish on my azure account

Change the url in //var serviceBase = 'http://ngauthenticationapi.azurewebsites.net/'; with my new url
Start the project web and tried the registration user but don't work I obtain this error "Failed to register user due to: "

Where is the error?

You can hel me?

br

Reply

Taiseer Joudeh says
March 20, 2015 at 10:46 pm

Hi,
I replied you by email, please read the post carefully as all the steps implemented successfully and thousand of devs implemented it successfully.

Reply

Raffaele Garofalo says
March 21, 2015 at 1:30 pm

Dude your series simply rocks. I just follow the tutorial step by step, and in exactly 30″ I have a web api OWIN authentication server up and running. Amazing, great job, eager to read the next steps!!

Reply

Taiseer Joudeh says
March 21, 2015 at 3:46 pm

Glad it was easy to follow and implement, keep tuned, the next part is coming soon.

Reply

Bilal™ (@akaMBS) says
March 23, 2015 at 6:21 pm

Salaan Taiseer, I am trying to add some abstraction layer but a bit stuck would be able to answer a question http://stackoverflow.com/questions/29195043/null-base-unitofwork-entityframework-with-repository-pattern-usermanager ?

Reply

Phil Martin says
March 27, 2015 at 2:17 am

Thanks for the great series of articles Taiseer. They are very informative reads.

May I ask a couple of questions please?

In the Seed() method, what is the benefit of creating a new ApplicationDbContext instead of using the one provided as a parameter?

Wehn you test the GetUsers method via a HTTP GET, you get a response where SuperPowerUser has a couple of claims. How did the claims get in there?

Reply

Taiseer Joudeh says
March 27, 2015 at 3:25 pm

Hi Phil, glad that posts are useful,
For the first question you are right, we can use the context in the parameter, I should fix this but it seems that I took this code from a unit test function I used to test the migration.
Good catch in the the second point 😃 I added them manually in the table to make sure that identity system is returning claims correctly, I'm currently working on next post which will cover claims in more depth. You can ignore those claims for now.
Regards,
Taiseer

Reply

Renato Lucas Chitolina says
March 27, 2015 at 4:46 am

Hello Taiseer, how are you?

I would like to congratulate you on the magnificent work that you have done with Web API, Asp.NET Identity, Angularjs, etc.
Great Job!!! Thanks for share your knowledge with us

Reply

Taiseer Joudeh says
March 27, 2015 at 3:19 pm

You are welcome, glad to know that posts are useful!

Reply

Bob says
April 3, 2015 at 10:38 am

Thanks for the tutorial. I followed the steps, and when testing with PostMan to create an account, I got error:

{"message":"The request entity's media type 'multipart/form-data' is not supported for this resource.","exceptionMessage":"No MediaTypeFormatter is available to read an object of type 'AccountBindingModels' from content with media type 'multipart/form-data'.","exceptionType":"System.Net.Http.UnsupportedMediaTypeException","stackTrace":" at System.Net.Http.HttpContentExtensions.ReadAsAsync[T](HttpContent content, Type type, IEnumerable1 `formatters, IFormatterLogger formatterLogger, CancellationToken cancellationToken)\r\n at System.Net.Http.HttpContentExtensions.ReadAsAsync(HttpContent content, Type type, IEnumerable`1 formatters, IFormatterLogger formatterLogger, CancellationToken cancellationToken)\r\n at System.Web.Http.ModelBinding.FormatterParameterBinding.ReadContentAsync(HttpRequestMessage request, Type type, IEnumerable`1 formatters, IFormatterLogger formatterLogger, CancellationToken cancellationToken)"}

Reply

Bob says
April 3, 2015 at 11:02 am

Never mind, forgot to put header there.

Reply

Taiseer Joudeh says
April 3, 2015 at 7:01 pm

Glad it is working!

craig snyder says
April 13, 2015 at 7:59 pm

Great article! Can’t wait for the next part. I am currently running into a problem and was hoping you could point me in the right direction –

I have downloaded the supplied code (using visual studio 2013) but cannot get the IIS express to find the controllers. I keep getting a 404 when I try to contact the controller regardless of the api call I make (create or get users) with postman. I have looked as IIS express and can see it running. The only difference between the downloaded coded and what I am running is that I am using a different DB running under SQL Server. How do I go about debugging this?

Reply

Taiseer Joudeh says
April 15, 2015 at 11:22 am

Hi Carig, apologies for the late reply. Well that is strange, the DB should do nothing with hitting the API endpoints. You still facing this issue or you were able to resolve it?

Reply

craig snyder says
April 15, 2015 at 6:44 pm

Hi Taiseer – thanks for the reply as I know your busy with part 6. I was able to figure it out, the problem was with PostMan. I must have missed your blog on how to use the tool. Now i just need to know how to connect a Web Client to the API, especially dealing with logging in.

Reply

Trevor de Koekkoek (@tdekoekkoek) says
April 15, 2015 at 10:04 am

Hi Taiseer,

As always, great posts on identity. I think I asked this already, but didn't see the comment come up. How is this implementation different from your previous series (June 2014) on implementing Identity with ASP.NET Web API? Just trying to understand what the differences are to see what's best to use for my own project. Thanks!

Reply

Taiseer Joudeh says
April 15, 2015 at 12:11 pm

Hi Trevor, thanks for your comment. This part focus more on ASP.NET Identity features and using JWT instead of default access token format, I recommend you to use JWT but the other is the same.

Reply

Luc says
April 20, 2015 at 6:34 pm

Hi Taiseer,

I'm following this tutorial, and getting this error when submiting this url "api/accounts/create" (with postman) :
{
"message": "An error has occurred.",
"exceptionMessage": "La référence d'objet n'est pas définie à une instance d'un objet.",
"exceptionType": "System.NullReferenceException",
"stackTrace": " à AspNetIdentity.WebApi.Controllers.AccountsController.d__a.MoveNext()
(…)
}
It's an Internal Server Error (500).

La référence d'objet n'est pas définie à une instance d'un objet == Object reference not set to an instance of an object
(I don't know how to translate this)

Reply

Taiseer Joudeh says
April 21, 2015 at 11:55 am

Make sure that the model for "Create User" has no null properties.

Reply

Luc says
April 22, 2015 at 8:42 am

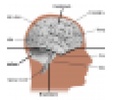
Thank you for your quick answer.

You were right, the "CreateUser" funtion was called with a null property.
I cheked it :

```
public async Task CreateUser(CreateUserBindingModel createUserModel)
{
if (createUserModel == null)
{
return Ok("CreateUserModel is null");
}
```

// it stops here.
I have to change this value, but i don't know where…
Maybe it's unfinded ?
What can i do ?

Reply



randi2160emlall says
June 16, 2015 at 11:34 pm

i am also getting the same …. please help resolve>?
"message": "An error has occurred.",
"exceptionMessage": "Object reference not set to an instance of an object.",
"exceptionType": "System.NullReferenceException",
"stackTrace": " at Learning.Web.Models.ModelFactory.Create(ApplicationUser appUser) in c:\\Applications\\eLearning.WebAPI-master\\eLearning.WebAPI-master\\Learning.Web\\Models\\ModelFactory.cs:line 36\r\n at Learning.Web.Controllers.AccountsController.d__6.MoveNext() in c:\\Applications\\eLearning.WebAPI-master\\eLearning.WebAPI-master\\Learning.Web\\Controllers\\AccountsController.cs:line 75\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.TaskAwaiter
```
1.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.TaskAwaiter`1.GetResult()\r\n at
System.Web.Http.Dispatcher.HttpControllerDispatcher.d__1.MoveNext()"
}issue

Reply

nino2015 says
April 21, 2015 at 7:22 pm

What about logging out a user? Would I use the RemoveLoginAsync and pass in the appropriate information?

Reply

nino2015 says
April 21, 2015 at 7:55 pm

Never mind, I just realized RemoveLoginAsync is part of the UserManager and don't want delete the user. I believe the best approach would be to have my Angular application clear out the token when the user logs out, correct?

Reply

Taiseer Joudeh says
April 22, 2015 at 9:33 am

This is for removing social identity account link, and there is no way to remove/expire self contained access tokens. All you need to do is to clear it from your client application and leave the token until it expires.

Reply

djpeluca says
April 26, 2015 at 2:43 am

Hi!

First of all thanks for this great tutorial, it's very helpful, but I have a problem that I couldn't resolve and maybe you can point me in the right direction.

My problem is with the step 4, because my DB is already created, I mean, I have another architecture, and I would like to us that and not the one proposed or auto-generated by your suggested code.

So, how could we adapt this to use our existing db structure and stored procedures? Is that possible?

Thanks and Regards

Reply

Taiseer Joudeh says
April 29, 2015 at 7:16 pm

Hi, you are welcome, glad that post is useful
Well you need to take a look at how to add ASP.NET Idenety to existing DB, maybe this video help, but I never try it my self yet.

Reply

Dew says
May 2, 2015 at 12:05 pm

Could you please provide sample code for “Log Out”?

Reply

Lester says
May 29, 2015 at 1:08 am

Just 1 question. Do I need to enable something in order to not allow or to throw an error when trying to create a user with an email address that already exists in the Db? Right now I am able to create user with same email address that already exists in the Db. Though, it doesn’t allow for same Username, without me enabling something.

Thank you for sharing your brain to us.

-Lester

Reply

Taiseer Joudeh says
May 29, 2015 at 8:34 am

I guess you need to configure under UserValidator property in class ApplicationUserManager as here

https://github.com/tjoudeh/AspNetIdentity.WebApi/blob/master/AspNetIdentity.WebApi/Infrastructure/ApplicationUserManager.cs
It's already configured to require unique username and email.

Reply

Lester says
May 29, 2015 at 10:56 pm

Thank you Taiseer. Yes, read it from Part 2. I guess default for Username is unique while Email is non-unique. Thumbs up!

Reply

hicode says
June 8, 2015 at 4:04 pm

Hi. the code is working on my machine. however got this error message while querying against http://beewestwebapi2.azurewebsites.net/oauth/token.

error:
XMLHttpRequest cannot load http://beewestwebapi2.azurewebsites.net/oauth/token. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:50453' is therefore not allowed access. The response had HTTP status code 400.

client:

```
$scope.login = function () {
//alert($scope.userName + ” “+ $scope.password);

var req = {
method:’POST’,
url: ‘http://beewestwebapi2.azurewebsites.net/oauth/token’,
//url:’http://localhost:55651/oauth/token’,
header: {
// ‘Content-Type': ‘application/x-www-form-urlencoded’
},

data: {
username: $scope.userName,
password: $scope.password,
grant_type:’password’
}
};
$http(req)
.success(function (data,status) {
$scope.info = data+” “+status;
})
.error(function (data, status) {
$scope.info = data + ” ” + status;
});
};
```

any idea pls?
Thanks

Reply

hicode says
June 8, 2015 at 4:44 pm

resolved. the corrected query should be:

```
var req = {
method:'POST',
url: 'http://beewestwebapi2.azurewebsites.net/oauth/token',
//url:'http://localhost:55651/oauth/token',
headers: {
'Content-Type': 'application/x-www-form-urlencoded'
},
data:'username=xxx%40gmail.com&password=yyy&grant_type=password'
//data: $.param({
// username: $scope.userName,
// password: $scope.password,
// grant_type:'password'
//})
};
```

Reply

Taiseer Joudeh says
June 13, 2015 at 2:53 am

You didn't configure CORS correctly so you can't it from JS app, it should be working from PostMan, so try to read the article again and follow how I\m configuring CORS.

Reply

Debashish says
June 13, 2015 at 6:51 pm

How do I write the tests?

Reply

Derek Rivers says
June 16, 2015 at 12:32 pm

This is an excellent article, very much appreciated.

Reply

Taiseer Joudeh says
June 18, 2015 at 2:17 am

Glad to hear it was useful 😃

Reply

Chris says
June 17, 2015 at 7:26 am

Awesome posts, Taiseer. I'm running into an issue that probably many others have had before me. On my windows 8.1 machine, I've installed Postman (now a standalone) and the interceptor (still in Chrome). Looked for your blog entry on using that but haven't found it yet. I'm assuming I have to run the owin server before I can access the identity app with Postman. So to begin in VS I start the app using the Chrome client. What Chrome gets back is a 402.14 Forbidden response. The boilerplate solution provided is to go to the IIS Express directory and run appcmd. But when I use %systemroot%\system32\inetserv\appcmd in that directory windows says appcmd is not recognized as an internal or external command. Of course if I try to run it without the path it says the same thing.
The error seems to be thrown in AccountController when it tries to create a new ApplicationUser. Not sure why it would do that _before_ a login screen was displayed.
Any thought readers have would be appreciated.

Reply

Taiseer Joudeh says
June 18, 2015 at 2:15 am

Hi Chris,
I'm not sure if I got your question correctly, PostMan is REST client like Fiddler where you can compose HTTP requests from, it is not related to Web API or Owin at all.
You need run this project on IIS express or normal IIS and there is no UI at all for the API, once you deploy the application to IIS you start use PostMan, fiddler, etc.. to issue HTTP requests to the API and get results from it.
Hope this clarifies your concern.

Reply

RT says
June 23, 2015 at 10:30 pm

Thanks for these sample solutions – these are helping a lot.

I had a question on how to extend the ASP.NET Identity the right way for adding an organizational structure.

So, we do have accounts, roles etc. however they also need to belong to organizations (or groups). It is a SaaS application with multiple tenants that we are hosting on azure. So, we can either:
1) Structure the organization to accounts in the "Resource Server" as opposed to the "Auth/Identity Server" OR
2) Use AD
3) Extend the "Auth/Identity Server" to have Organizations/Groups in them.

I'd be curious to hear thoughts on the above options.

Thanks in advance.

Reply

TC says
June 29, 2015 at 9:10 pm

Hi Taiseer,

Great post and series in general. I found it very helpful.

I'm interested in using ASP.NET Identity / WebApi from an Angular client.

Any update on when you might have the final part of your series posted?

Best,

TC

Reply

Dhanilan says
July 8, 2015 at 1:58 pm

There is a CORS exception when I try to sign up in the demo application http://ngauthenticationweb.azurewebsites.net/#/signup . This happens for me as well when I deploy Web Api to Azure . In local it works fine. Please help.

Reply

Taiseer Joudeh says
July 9, 2015 at 1:10 am

I'm not facing any issue when I try to register from the demo website. Can you try again and let me know if there is issue happening.

Reply

Dhanilan says
July 10, 2015 at 7:28 am

Hi Taiseer I have this issue only in my office network, that too across all browsers and all of my colleagues system. but doesn't happen at home. Very strange (and confused),any guess?

Reply

Dhanilan says
July 10, 2015 at 7:38 am

gotcha Firewall causes the issue. http://jeffmcmahan.info/blog/firewall-causes-cors-to-fail/

Reply

Dasun Sameera Weerasinghe says
July 22, 2015 at 4:30 pm

I wonder is _AppUserManager on BaseApiController have ever initialized? Or It just wait to be implement in future?

Anyway thanks again for this nice tutorial series.

Reply

Jaime says
July 26, 2015 at 9:02 am

I followed your tutorial a couple of times but still I get a 404 when I try to test requests. I see that the update-database command created the database without any errors and the seed method was called(I see the first user record in the database). However, I continuously see 404 and the application can't find my api requests. I even tried by starting with a web api project.

The api MVC routes work but not WebApi routes. Also the owin startup class is called as expected and I have decorated the correct RoutePrefix for class and Route attributes for the Web Api methods in the

AccountController, but still no dice.

I have tried to get the application to work both using Local IIS as well as IISExpress.

On my last try I got an error:
"ExceptionMessage": "Value cannot be null.\\\r\\\nParameter name: context"

So, I stopped in the debugger and found that "Request.GetOwinContext()" returns null. Hopefully, this is enough information for someone to lend a hand.

Thanks!

Reply

Taiseer Joudeh says
July 27, 2015 at 4:21 am

Hi Jamie,
I feel that you are mixing between Owin middleware and the traditional routes registration, that's why your Web Api routes are not configured correctly and not being registered. Could you please just download the sample code and try to run it directly without adding MVC components, once you make sure that the Api is working correctly you can add Mvc NuGet packages. It is hard to tell where is the issue happening because there is lot of moving parts.

Reply

Agostino Cinelli says
July 29, 2015 at 3:29 pm

Hi, what about OpenId? Can it be used with or in the place of ASP.NET Identity?

Reply

Julius Tetteh says
August 5, 2015 at 12:31 pm

Hi, when is "AngularJS Authentication and Authorization with ASP.NET Web API and Identity – Part 6″ coming up?
I just cant wait?
Thanks

Reply

davidetrotta says
August 7, 2015 at 12:06 pm

Compliments for the post.

I have just one question:
How is possible, in your opinion, secure the method CreateUser that currently it [AllowAnonymous]

[AllowAnonymous]
[Route("create")]
public async Task CreateUser(CreateUserBindingModel createUserModel)

Of course AllowAnonymous is correct, because the user still doesn't exist in our database.
But I am thinking that everyone can "call" this method infinite time in a loop destroying for instance our database.
With cookie session we have more "protection" for these kind of attacks.

Is there any best practice for prevent these situations?

Thank you,

Davide

Reply

Taiseer Joudeh says
August 8, 2015 at 7:45 am

Hi Davide,
Even if you are using cookies not tokens, you will not have a cookie for anonymous users, so this scenario will happen too, you can think of using captcha or Api rate limiter to make sure that user is not sending for example more than 2 requests per 5 seconds.

Reply

Davidetrotta says
August 8, 2015 at 9:35 am

Thank you for your reply. Great article great work.
Cookie. Vs tokens and eventually single sign on are the hot topic in London. Your article as far I can see is one of the milestone for understanding tokens and web.api

Reply

Carlos Ribeiro says
August 7, 2015 at 12:45 pm

Hi Taiseer, I would like to thank you for this great article!
You have no idea how much you´ve helped me!
I have followed it up to Part 5 and every thing works smoothly.
I would just like to add a footnote for those starting: Turn off you firewall, because many of the problems that I read above seem to me to be related to it, as it happened to me.

Reply

Taiseer Joudeh says
August 8, 2015 at 7:45 am

Thanks Carlos for your comment, always happy to help, and thanks for your note!

Reply

Mahi says
August 7, 2015 at 2:30 pm

hey.. really nice article. i followed ur tutorial. but wen i test it through fiddler m receiving an empty object in createUser method.

Reply

Taiseer Joudeh says
August 8, 2015 at 7:47 am

Hi, make sure your request body is identical to mine, if not that's why the serializer is not able to parse the request payload and cast it to CLR object.

Reply

Alderan says
August 19, 2015 at 12:48 pm

Would this still work with MVC6 and ASP.NET5? Now that WebAPI and MVC are merged, I'm wondering if any of this (great) series of article is still valid?

Reply

dagilleland says
August 22, 2015 at 1:21 pm

Just a little FYI for your readers if the Seed() method does not appear to add the user to the database:

If you attempt to create the user with a password that is too short or otherwise does not meet the minimum password requirements, then the Seed() method will silently fail to generate the user (even when using the -verbose flag on update-database). The simple solution is to ensure that your password is long enough.

E.g.: This will fail to create the user:
manager.Create(user, "demo");

This, however, will work:
manager.Create(user, "demoPassword");

Reply

Taiseer Joudeh says
August 24, 2015 at 10:55 pm

That is right, you have to set a password which meets the password policy defined in the ApplicationUserManager class. Thanks for sharing this.

Reply

Apollo says
September 17, 2015 at 3:23 pm

Cheers Taiseer

Reply

mbeddedsoft says
September 20, 2015 at 6:57 pm

hello I enjoy your article. It appears to be concise and exactly what I was searching for.
However, I cannot simply execute 'enable-migrations' in PM Console.
I get the error:

System.ArgumentException: The parameter is incorrect. (Exception from HRESULT: 0x80070057 (E_INVALIDARG))

is there something I missed?

thanks,

Reply

mbeddedsoft says
September 20, 2015 at 7:28 pm

Figured it out. I mistakenly created a 'web site' and not a 'web application'.
Thanks again for the great article.

m.

Reply

Taiseer Joudeh says
September 21, 2015 at 2:47 pm

Glad to hear it is working now.

Reply

Taiseer Joudeh says
September 21, 2015 at 2:46 pm

I'm not sure what is this error is, did you try searching it on stack overflow?

Reply

Trackbacks

The Morning Brew - Chris Alcock » The Morning Brew #1784 says:
January 23, 2015 at 11:17 am
[…] ASP.NET Identity 2.1 with ASP.NET Web API 2.2 (Accounts Management) – Part 1 – Taiseer Joudeh […]

Reply

The Morning Brew - Chris Alcock » The Morning Brew #1785 says:
January 26, 2015 at 1:56 pm
[…] ASP.NET Identity 2.1 with ASP.NET Web API 2.2 (Accounts Management) – Part 1 – Taiseer Joudeh […]

Reply

Leave a Reply

ABOUT TAISEER

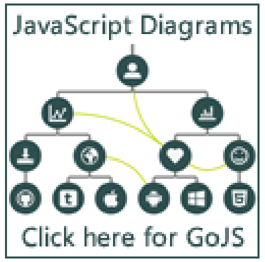
Father, MVP (ASP

.NET/IIS), Scrum Master, Life Time Learner

CONNECT WITH ME

Microsof
Most Valuab
Professional

Advertise Here



Advertise H

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

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

Search this website…