Customizing Token Based Authentication (OAuth) in ASP.NET Web API with Existing User Database

12 March 2017 C#, ASP.NET, Web API, OAuth, REST

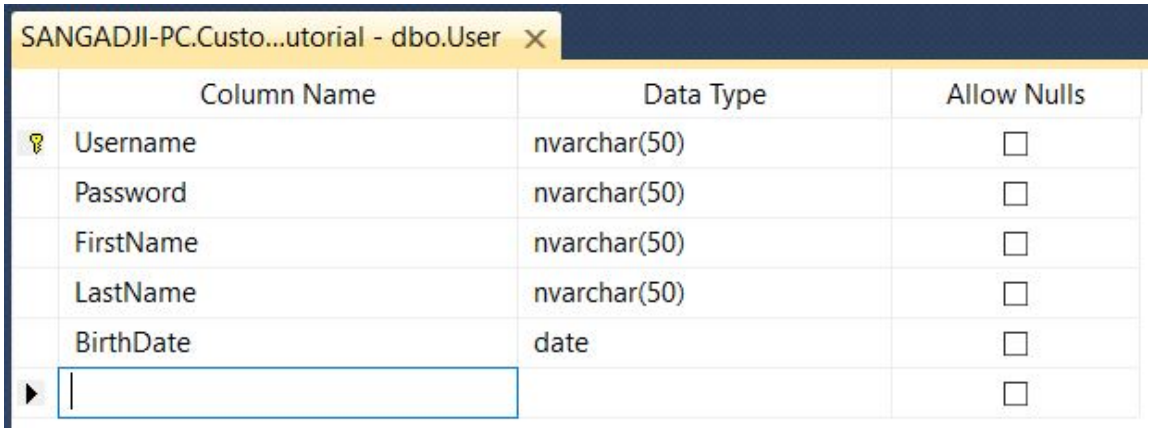
ASP.NET Web API project provides built-in OAuth provider to authorize and authenticate users by using access tokens. By default, the user information is stored using OWIN Middleware in the ASP.NET Identity system. The ASP.NET Identity system stores all the user information in a database where it utilizes Entity Framework Code First to implement all of its persistence mechanism. However, sometimes we were faced with a situation where we want to customize our web service to use another or existing database such as ASP.NET Membership, or perhaps our custom made user table. This tutorial will explain the step-by-step procedure to customize built-in token based authentication in ASP.NET Web API to our needs.

Software Requirements

- Visual Studio 2015
- Microsoft SQL Server 2014
- SQL Server Management Studio 2014
- Telerik Fiddler

Database Structure

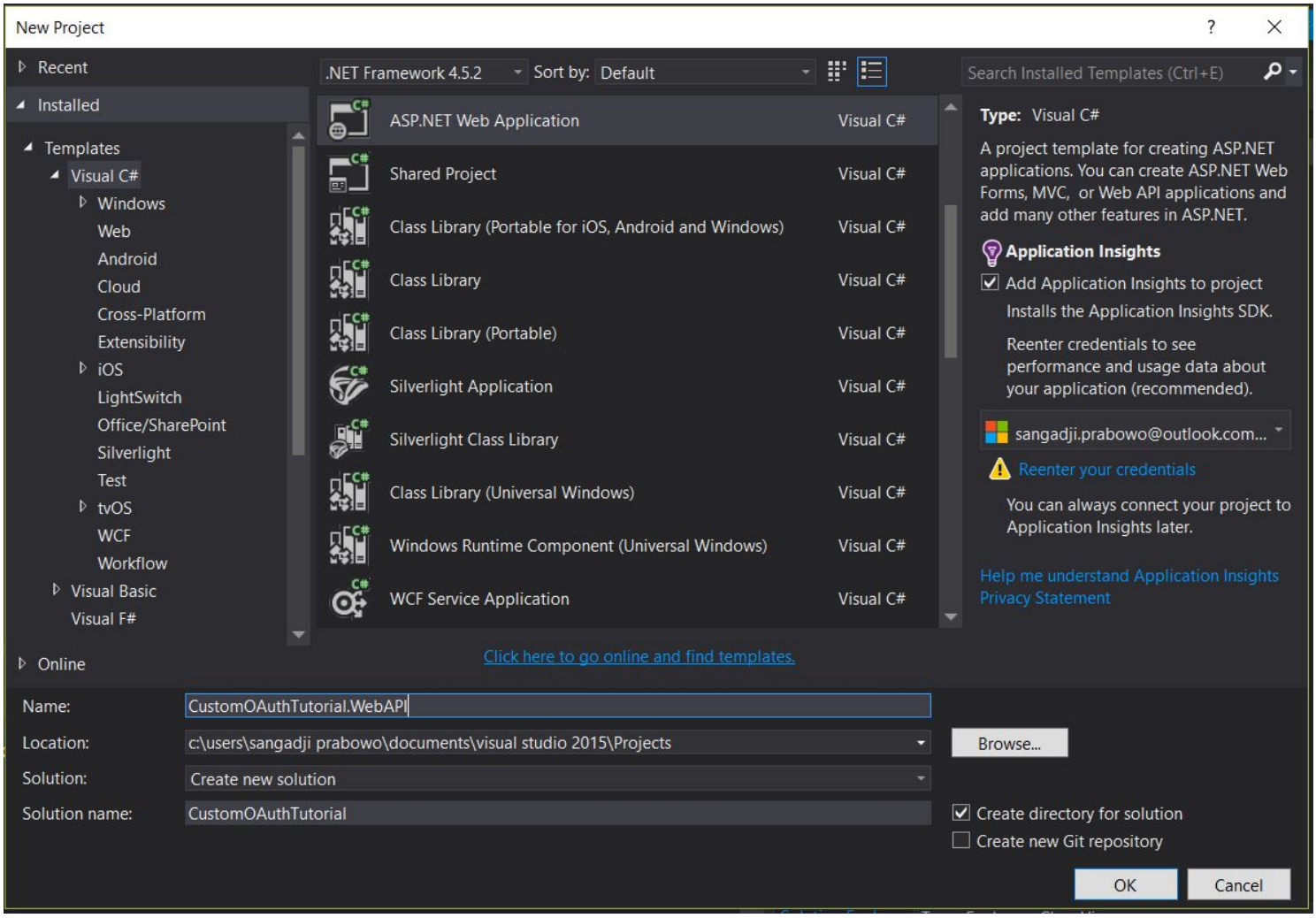
For this tutorial, we are going to use a custom user table in a SQL Server database. The table contains following attributes: username (PK), hashed password, first name, last name, and birth date. Every attribute is not allowed to have null values. For simplicity reason, the passwords will be hashed using MD5 hashing.



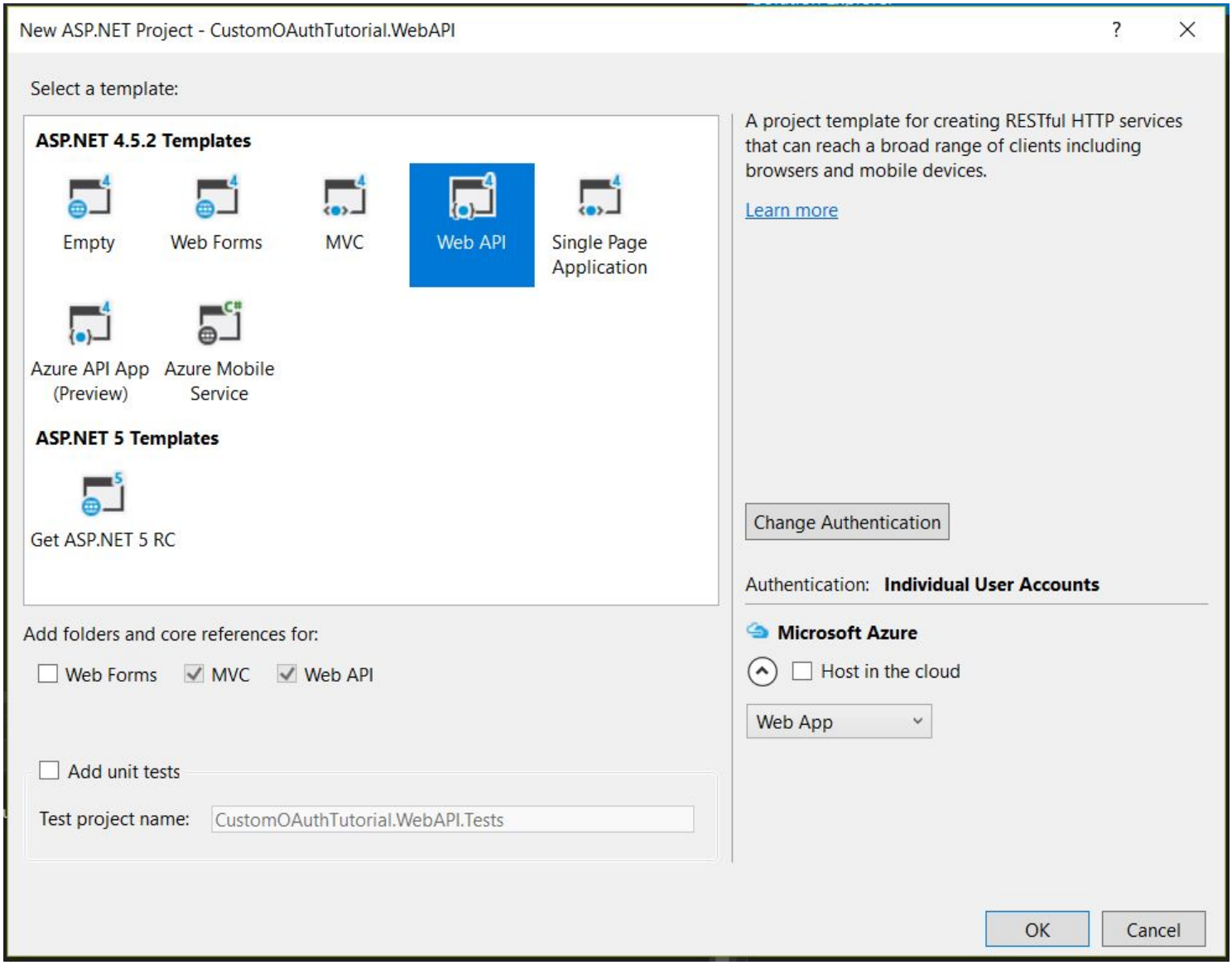
> “ It is important to note that in the real-world scenario MD5 hashing is considered to be not secure anymore. Use SHA2 hashing along with a salt key to securely store passwords.

Step 1: Create ASP.NET Web API Project

First, we need to create an ASP.NET Web API project (this tutorial uses .NET 4.5 version). On the Visual Studio window select **File > New > Project > ASP.NET Web Application**.



Select **Web API** template on the new ASP.NET project template selection window.



Step 2: Add Connection String on the Web.config File

Add a connection string to the web.config file. For example, I added a connection string named "OAuthContext" where it stores a user table in a localhost database named CustomOAuthTutorial.

```
<configuration>
   <connectionStrings>
      <add name="OAuthContext" connectionString="Data Source=.;Initial Catalog=Cust
   </connectionStrings>
</configuration>
```

Step 3: Create The DB Context

Create the data context class using the connection string. This data context will be used later to create the repository class for the user crud operations.

Create **OAuthContext.cs** in the **Models** folder.

```
public class OAuthContext: IdentityDbContext<IdentityUser>
{
     public AuthContext()
         : base("OAuthContext", throwIfV1Schema: false)
     {
     }

     public static OAuthContext Create()
     {
         return new OAuthContext();
     }
}
```

Step 4: Create The User Model

Create a model class for the users. The class should contain the attributes declared on the user table. You could also add data validation using data annotations in this class.

Create **UserModel.cs** in the **Models** folder.

```
public class UserModel
{
    [Required]
    [Display(Name = "User name")]
    public string UserName { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters lon
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters lon
    public string FirstName { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters lon
    public string LastName { get; set; }

    [Required]
```

```
22      public DateTime BirthDate { get; set; }
23  }
```

Step 5: Create a Repository Class

Create a repository class for persistence operations on the AuthContext. This class will be responsible for registering and finding a username in the database.

Create **AuthRepository.cs** in a new folder called **Utilities.**

```
1   public class AuthRepository : IDisposable
2       {
3           private AuthContext authContext;
4           private UserManager<IdentityUser> userManager;
5
6           public AuthRepository()
7           {
8               authContext = new AuthContext();
9               userManager = new UserManager<IdentityUser>(new UserStore<IdentityUser
10          }
11
12          public async Task<IdentityResult> RegisterUser(UserModel userModel)
13          {
14              IdentityUser user = new IdentityUser
15              {
16                  UserName = userModel.UserName
17              };
18
19              var result = await userManager.CreateAsync(user, userModel.Password);
20              return result;
21          }
22
23          public async Task<IdentityUser> FindUser(string userName, string password)
24          {
25              return await userManager.FindAsync(userName, password);
26          }
27
28          public void Dispose()
29          {
30              authContext.Dispose();
31              userManager.Dispose();
32          }
33      }
```

Step 6: Create an OAuthProvider Class

If you look at the **Providers** folder in the solution explorer, there is a built-in class there named **ApplicationOAuthProvider.cs**. This class purpose is to validate client authentication and granting the client access to resources by assigning claims. You can learn more about claims via this link.

We will replace this class with our custom OAuthProvider so that we can use our custom user model and database to authenticate a client.

Create **CustomOAuthProvider.cs** in the **Providers** folder.

```
public class CustomOAuthProvider : OAuthAuthorizationServerProvider
{
    public override Task ValidateClientAuthentication(OAuthValidateClientAuthentic
    {
        // Resource owner password credentials does not provide a client ID.
        if (context.ClientId == null)
        {
            context.Validated();
        }

        return Task.FromResult<object>(null);
    }

    public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwn
    {
        context.OwinContext.Response.Headers.Add("Access-Control-Allow-Origin", ne

        using (AuthRepository authRepo = new AuthRepository())
        {
            IdentityUser user = await authRepo.FindUser(context.UserName, context.

            if (user == null)
            {
                context.SetError("invalid_grant", "The user name or password is in
                return;
            }
        }

        var identity = new ClaimsIdentity(context.Options.AuthenticationType);
        identity.AddClaim(new Claim(ClaimTypes.Name, context.UserName));
        // Optional : You can add a role based claim by uncommenting the line belo
        // identity.AddClaim(new Claim(ClaimTypes.Role, "Administrator"));

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

Step 7: (Optional) Create Password Hasher

It is strongly advised to never store the actual password on the user table. Storing users actual password in the database can pose a security threat thus violating the user's privacy. You can implement a password hasher by creating a class that implements **IPasswordHasher**.

Create **MD5PasswordHasher.cs** in the **Utilities** folder.

```
public class MD5PasswordHasher : IPasswordHasher
{
     public string HashPassword(string password)
     {
         // Use input string to calculate MD5 hash
         using (System.Security.Cryptography.MD5 md5 = System.Security.Cryptograph
         {
             byte[] inputBytes = System.Text.Encoding.ASCII.GetBytes(password);
             byte[] hashBytes = md5.ComputeHash(inputBytes);

             // Convert the byte array to hexadecimal string
             StringBuilder sb = new StringBuilder();
             for (int i = 0; i < hashBytes.Length; i++)
             {
                 sb.Append(hashBytes[i].ToString("X2"));
             }
             return sb.ToString();
         }
     }

     public PasswordVerificationResult VerifyHashedPassword
                     (string hashedPassword, string providedPassword)
     {
         if (hashedPassword == HashPassword(providedPassword))
             return PasswordVerificationResult.Success;
         else
             return PasswordVerificationResult.Failed;
     }
 }
```

Step 8: Configure Authentication Options

Finally, the last thing we need to do is to configure the auth options in the **Startup.Auth.cs**. Replace the **ConfigureAuth** method with the following lines of code.

```
public void ConfigureAuth(IAppBuilder app)
{
    // Configure the application for OAuth based flow
    OAuthOptions = new OAuthAuthorizationServerOptions
    {
        TokenEndpointPath = new PathString("/Token"),
        Provider = new CustomOAuthProvider(),
        AuthorizeEndpointPath = new PathString("/api/Account/ExternalLogin"),
        AccessTokenExpireTimeSpan = TimeSpan.FromDays(10),
```

```
            // In production mode set AllowInsecureHttp = false
            AllowInsecureHttp = true
        };

        // Enable the application to use bearer tokens to authenticate users
        app.UseOAuthAuthorizationServer(OAuthOptions);
        app.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions());
    }
```

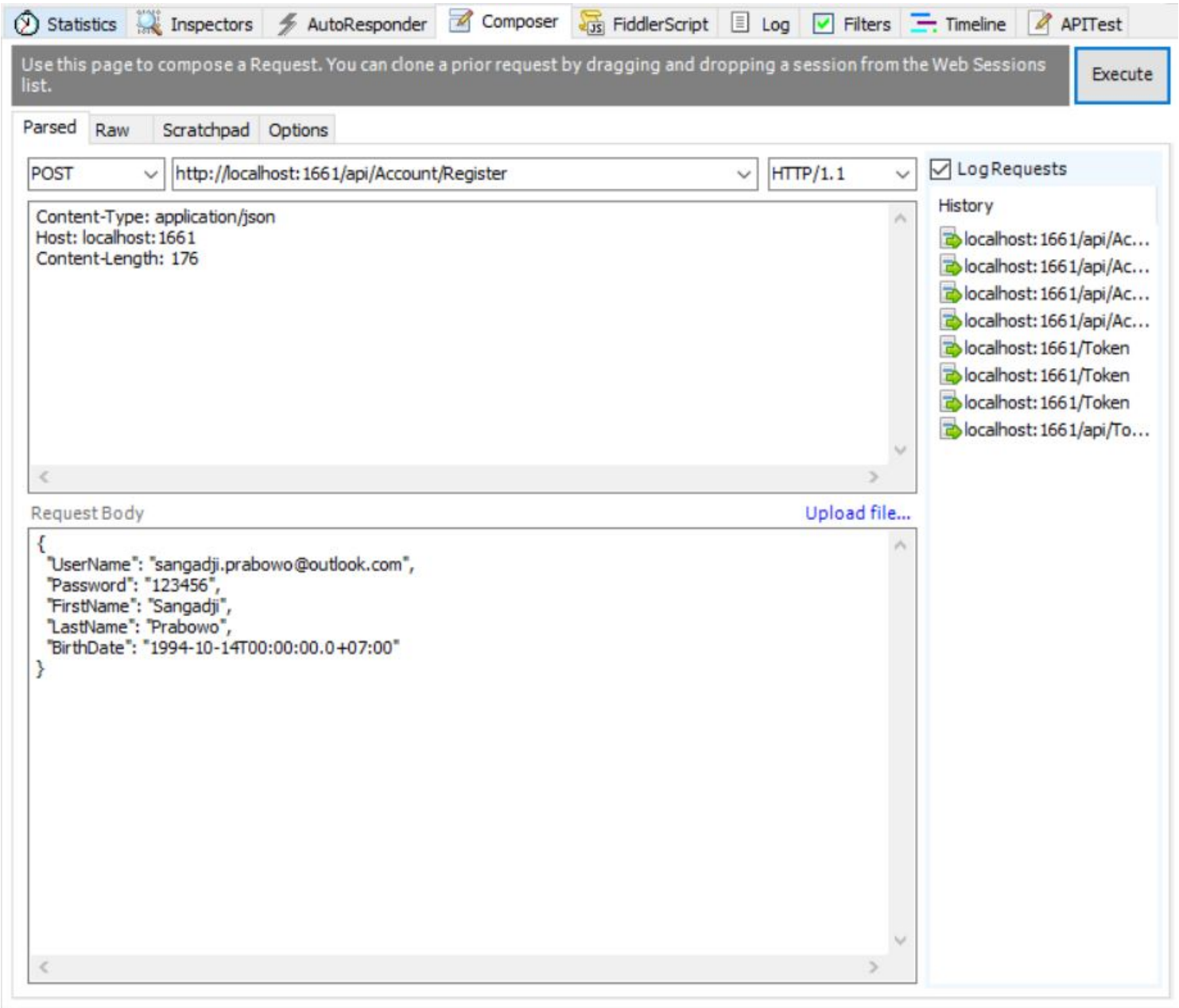
Testing The Web API

To test the Web API, first we need to modify the **Register** method on the **AccountController.cs** class in the **Controllers** folder to the following code.

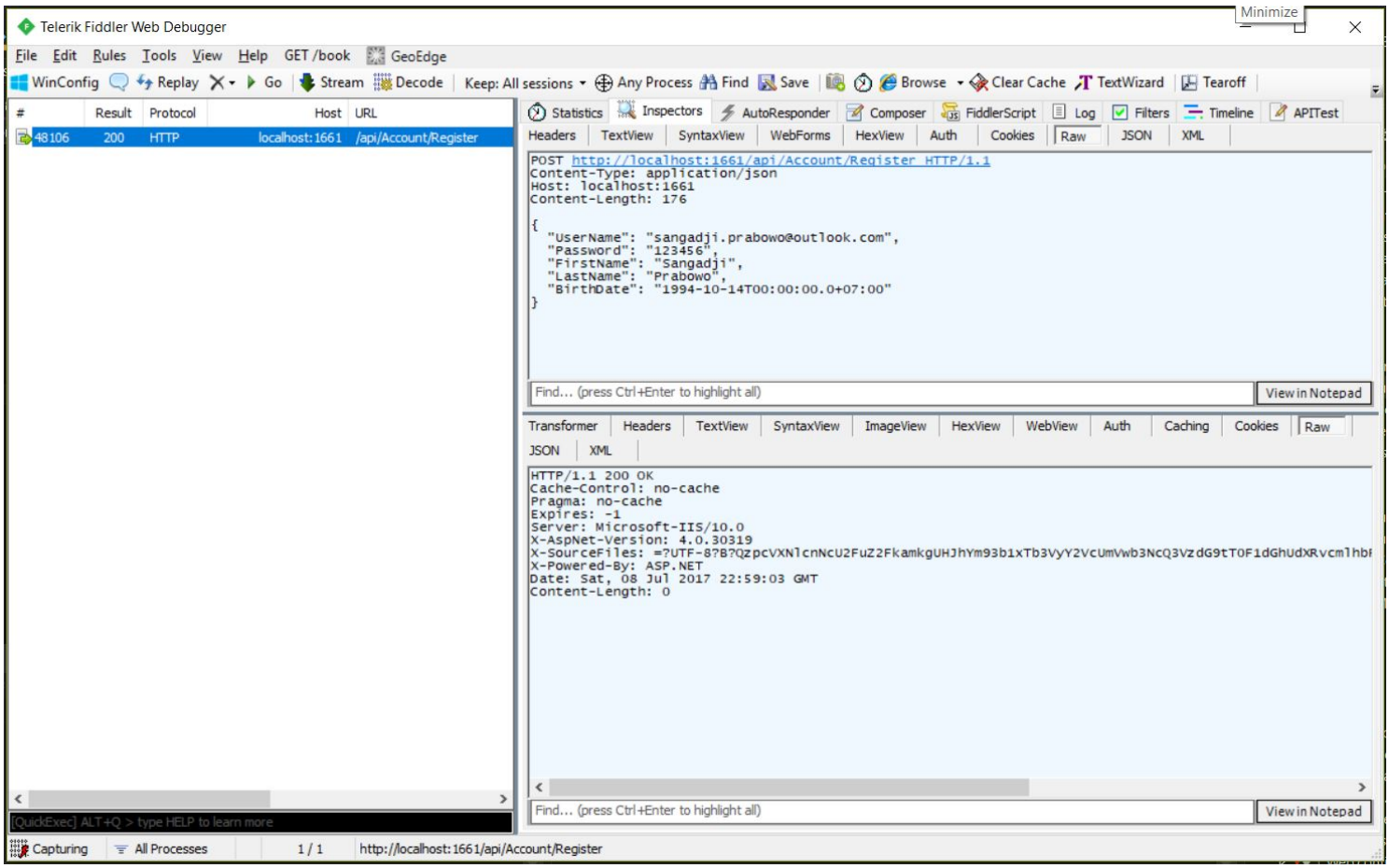
```
[AllowAnonymous]
[Route("Register")]
public async Task<IHttpActionResult> Register(UserModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    using (var repository = new AuthRepository())
    {
        IdentityResult result = await repository.RegisterUser(model);

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

        return Ok();
    }
}
```

Next, we will perform a register and authentication operation on the Web API. I love to use Telerik Fiddler for sending HTTP requests and receiving HTTP responses to a Web API. Before we do this we need to compile and run the project on the localhost.

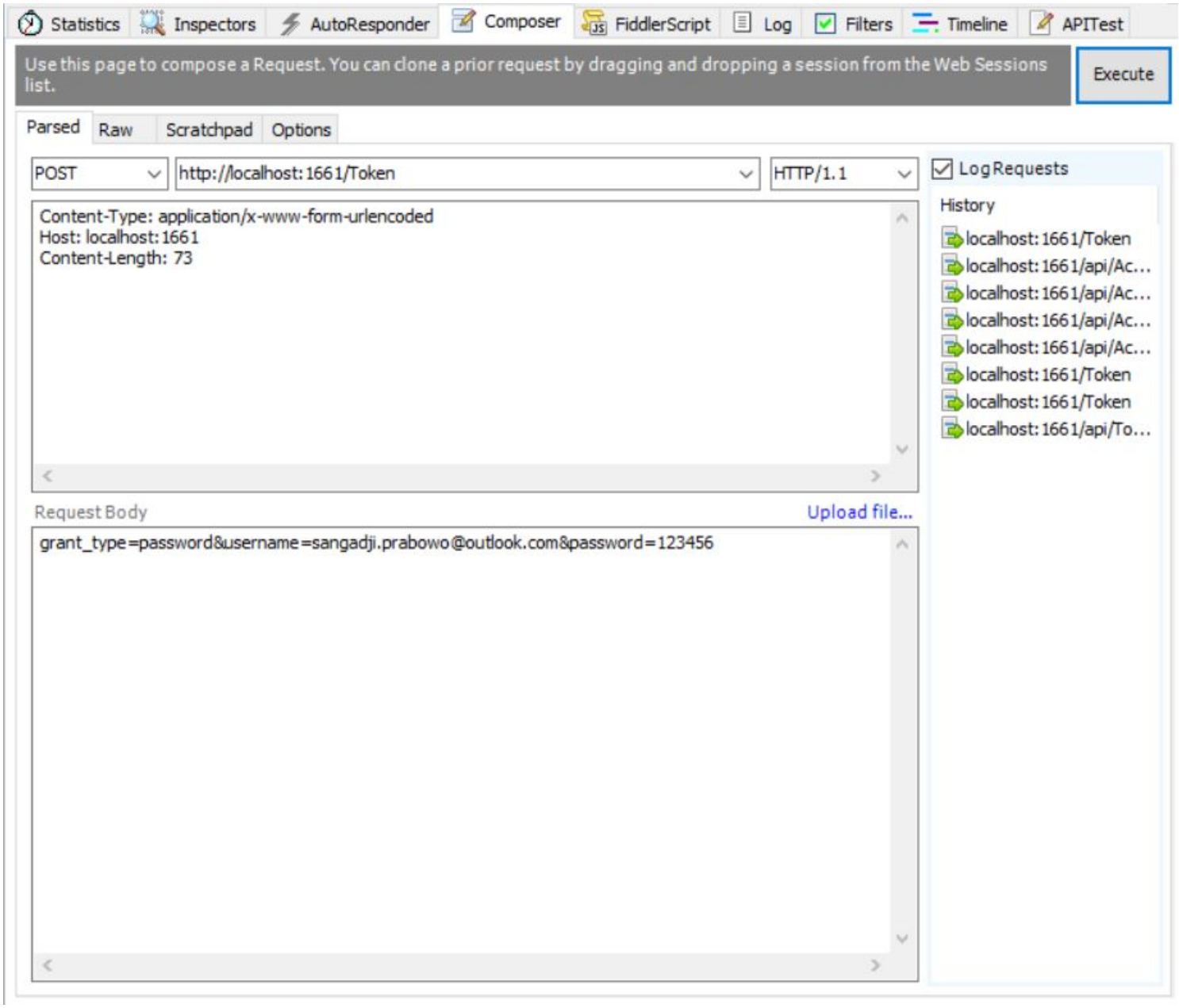
Send an HTTP POST request to the register URL with the following request header and body.



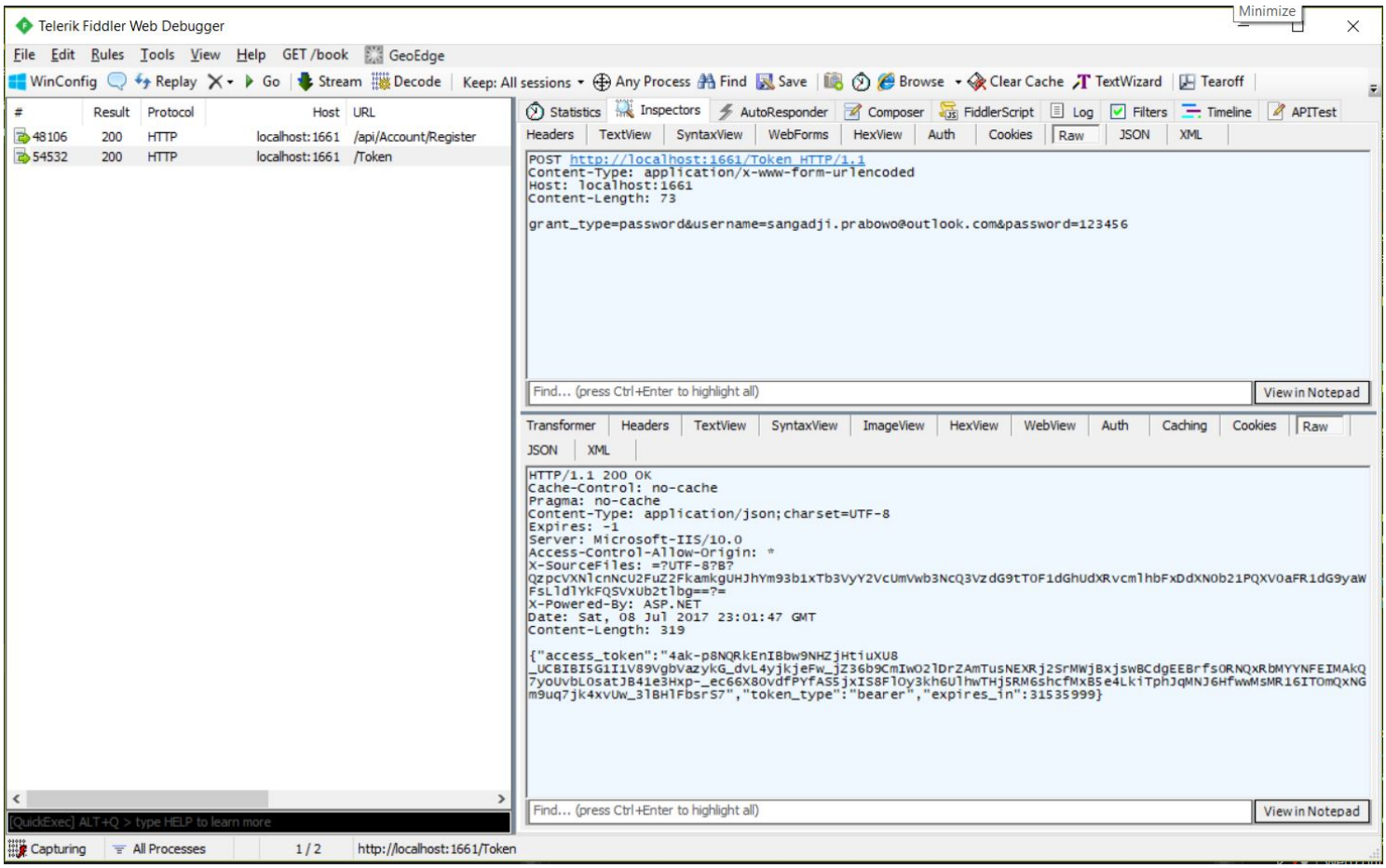
If the account creation was successful, you will get a response with **status code 200 (OK).**



Then you could try to authenticate with the previously created account by initiating a POST request with the following request header and body.



If the authentication process was successful, you will get a response with **status code 200 (OK)** along with the access token and the expiration information.



Download Sample Project

You can download the finished project in https://github.com/sangadji/CustomOAuthTutorial.

Tags : C#, ASP.NET, Web API, OAuth, REST

Currently rated 4.5 by 4 people

Related posts

- Creating Simple FAQ Chatbot using QnA Maker API
- Azure Resources Naming Conventions
- Comparing Asynchronous vs Synchronous Program Execution Time in .NET

7 Comments Sort by **Oldest**

Add a comment...

Adeem Rajpoot

i thnk this can solve my problem

Like · Reply · ٤٤w

Abhishek Raj

can you please tell me in which table the data after registration is stored?

Like · Reply · 1 · ٤٣w

Adeem Rajpoot

Abhishek Raj aspnetUsers

Like · Reply · ٤٣w

Adeem Rajpoot

but i dont want to store in this table can help me

Like · Reply · ٤٣w

Sándor Hatvani

Thank you for article but when I want to validate against role in Active Directory it always returns 'true' value regardless of role. Can you help me, please?

Like · Reply · ٣٢w

Victoria Del Castillo

I keep on getting this error when testing my API :

{"Message":"The request is invalid.","ModelState":{"model.UserName" "The Username field is required."],"model.Password" "The Password field is required."]}}

I'm not sure how to fix this. My syntax is fine.... I have tried both XML and JSON.

Like · Reply · ٣١w

Sowkath Iqbal

Token generated successfully and using bearer token , able to do GET and POST using Postman.
But from Angular using bearer token causing 401 unauthorized exception. can you help us how to do GET or POST from external application like angular

Like · Reply · ٢٥w

Ali Adravi

Nice and quite detailed article.
For Role based authentication with custom database see
http://www.advancesharp.com/.../oauth-web-api-2-bearer...

Like · Reply · ٢٣w

Dorababu Meka

How to achieve the same with MVC

Like · Reply · ٢٢w

Arpit Rai

How can I authorize through my refresh token. I have client ID is in encrypted format. I have searched everywhere but could not find anything like my question.
Thanks in advance.

Like · Reply · ٢d

Facebook Comments Plugin

Posts

Latest

Categories

Microsoft Azure

Universal Windows Platform

Xamarin

Visual Studio

Office 365

Pages

Blog

Archive

Contact