Why an OpenID Connect server?

Adding an OpenID Connect server to your application allows you to support token authentication. It also allows you to manage all your users using local password or an external identity provider (e.g. Facebook or Google) for all your applications in one central place, with the power to control who can access your API and the information that is exposed to each client.

[OpenIddict](https://github.com/openiddict/openiddict-core) is an open source framework for ASP.NET Core which allows you to easily implement an [OpenID Connect](https://openid.net/connect/)server.

JWT Middleware + OpenIddict/IDS4.

You will need to have some sort of OpenID Connect authorization server such as OpenIddict or IDS4 (or a commercial offering like Auth0, Okta, etc) which will be used to authorize the users and issue an id\_token + access\_token. These tokens can be stored in local storage in the browser.

Your Angular app will then need to pass the access\_token in the Authorization header with requests being made to your ASP.NET Core API. Inside the ASP.NET Core API you can use JWT middleware to authorize the request by ensuring the access\_token is valid.

This is assuming that the access\_token is a JWT, since OpenIddict, for example, can generate access tokens as either a JWT or it's own proprietary format - in which case you need to use their OAuth introspection middleware instead of the JWT middleware

[IdentityServer4](https://github.com/IdentityServer/IdentityServer4):

an OpenID Connect and OAuth 2.0 framework for ASP.NET Core 2.

[AspNet.Security.OpenIdConnect.Server](https://github.com/aspnet-contrib/AspNet.Security.OpenIdConnect.Server):

is an advanced OAuth2/OpenID Connect server framework for both ASP.NET Core 1.x/2.x and OWIN/Katana 3.x/4.x, designed to offer a low-level, protocol-first approach.

[OpenIddict](https://github.com/openiddict/openiddict-core):

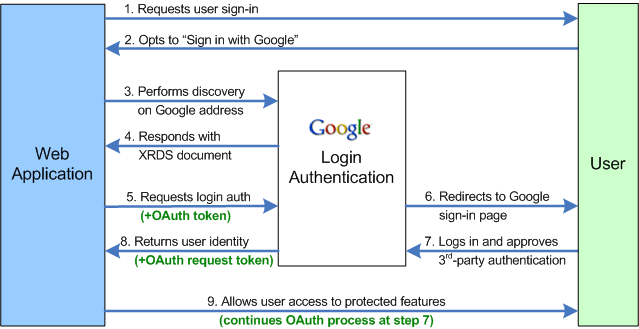
OpenIddict aims at providing a simple and easy-to-use solution to implement an OpenID Connect server in any ASP.NET Core 1.x or 2.x application.

OpenIddict is based on AspNet.Security.OpenIdConnect.Server to control the OpenID Connect authentication flow and can be used with any membership stack, including ASP.NET Core Identity.

Also have checked that all of them use well the [ASP.NET Core Identity](https://github.com/aspnet/Identity) as a membership system.

And so my current understanding is that IdentityServer4 and OpenIdConnect.Server are two alternative frameworks that solve the same problem. The main difference is the list of supported ASP.NET Core versions.

Regarding Openiddict - it is a kind of extension to simplify server creation based on AspNet.Security.OpenIdConnect.Server.



ID Tokens vs Access Tokens. The ID Token is a security token granted by the OpenID Provider that contains information about an End-User. ... Access tokens, on the other hand, are not intended to carry information about the user. They simply allow access to certain defined server resources

<https://learncode.net/2019/02/05/3-create-your-own-asp-net-core-react-redux-boilerplate-setup-authentication/>

Enable Entity Framework and add dbcontext to the project

If you already added Entity Framework to the project you can skip “Enable Entity Framework and add dbcontext to the project” section.

Entity Framework (EF) Core is a lightweight, extensible version of Entity framework data access technology. To add EF Core to the project, we should install Microsoft.EntityFrameworkCorepackage from nuget package manager. But as we can see Microsoft.AspNetCore.App package (check our “.csproj” file) is installed in our project and it includes EntityframeworeCore, so there is no need to install EF Core separately.

|  |  |
| --- | --- |
| 1  2  3  4 | <ItemGroup>      <PackageReference Include="Microsoft.AspNetCore.App" />      <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0" PrivateAssets="All" />    </ItemGroup> |

Otherwise, we should run the below command to install EF Core:

|  |  |
| --- | --- |
| 1  2 | dotnet add package Microsoft.EntityFrameworkCore --version 2.2.0  dotnet restore |

Let create our first entity to keep users information named ApplicationUser. It is “Best Practice” to create our entities under Core\Models folder.

|  |  |
| --- | --- |
| 1  2  3  4  5 | …> mkdir Core  …> cd Core  …/Core> mkdir Models  …/Core> cd Models  …/Core/Models> echo.> ApplicationUser.cs |

Open the ApplicationUser.cs file and modify it as below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | using Microsoft.AspNetCore.Identity;    namespace MyApp.Core.Models  {      public class ApplicationUser : IdentityUser      {      }  } |

It is “Best Practice” to create Dbcontext in a separate folder from Models, so let’s create Persistence folder and add ApplicationDbContext.cs file to it:

|  |  |
| --- | --- |
| 1  2  3 | …>mkdir Persistence  …>cd Persistence  ../Persistence> echo.> ApplicationDbContext.cs |

Modify ApplicationDbContext.cs file as below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Identity.EntityFrameworkCore;  using Microsoft.EntityFrameworkCore;  using MyApp.Core.Models;    namespace MyApp.Persistence  {    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>    {      public ApplicationDbContext(DbContextOptions options)      : base(options)      {      }    }  } |

Open the appsettings.json file and add ConnectionStrings to it (I have SQL Server express localdb 2014 installed on my pc, feel free to change the ConnectionStrings if you have another version of SQL Server installed)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {    "ConnectionStrings": {      "Default": "server=(LocalDb)\\MSSQLLocalDB; database=MyAppDbContext; Integrated Security=SSPI"    },    "Logging": {      "IncludeScopes": false,      "LogLevel": {        "Default": "Warning"      }    }  } |

To be able to run EF Core Commands we need one more package named Entity Framework Core .NET Command Line Tools. Run below commands:

|  |  |
| --- | --- |
| 1  2 | dotnet add package Microsoft.EntityFrameworkCore.Tools.DotNet --version 2.0.3  dotnet restore |

Now that EF Core is installed, let Configure EF Core in ConfigureServices of Startup.cs. so open startup file and add below code to it:

|  |
| --- |
| using MyApp.Persistence;  using Microsoft.EntityFrameworkCore;  .  .  .  public void ConfigureServices(IServiceCollection services)          {              services.AddMvc();                 services.AddDbContext<ApplicationDbContext>(o =>              {                  o.UseSqlServer(Configuration.GetConnectionString("Default"));              });          } |

Now, we should create our first migration.

|  |  |
| --- | --- |
| 1 | dotnet ef migrations add init |

If you check you could see migration folder and its files in your root directory. To update the database run below command:

|  |  |
| --- | --- |
| 1 | dotnet ef database update |

Add OpenIddict to the project

Open MyApp.csproj and add the below code to it then restore the dotnet project.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <ItemGroup>       <PackageReference Include="OpenIddict" Version="2.0.0-\*" />       <PackageReference Include="OpenIddict.EntityFrameworkCore" Version="2.0.0-\*" />       <PackageReference Include="OpenIddict.Mvc" Version="2.0.0-\*" />  </ItemGroup> |

Then run the below command:

|  |  |
| --- | --- |
| 1 | dotnet restore |

Create authentication controllers

We need two controllers:

Authorization controller

We are going to implement our authorization endpoint in this controller. We will create one action called Exchange which will handle /connect/token route which we specified as the authorization endpoint for OpenIddict. To keep token creating logic separated, we will create one another method called CreateTicketAsync.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class AuthController : Controller  {      [HttpGet("~/connect/token"), Produces("application/json")]      public async Task<IActionResult> Exchange([ModelBinder(typeof(OpenIddictMvcBinder))] OpenIdConnectRequest request)      {      }        private async Task<AuthenticationTicket> CreateTicketAsync(OpenIdConnectRequest request,          ApplicationUser user, AuthenticationProperties properties = null)          {          }  } |

So, create the AuthController.cs file under Controllers folder as below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211 | using System.Collections.Generic;  using System.Diagnostics;  using System.Linq;  using System.Threading.Tasks;  using AspNet.Security.OpenIdConnect.Extensions;  using AspNet.Security.OpenIdConnect.Primitives;  using AspNet.Security.OpenIdConnect.Server;  using Microsoft.AspNetCore.Authentication;  using Microsoft.AspNetCore.Identity;  using Microsoft.AspNetCore.Mvc;  using Microsoft.Extensions.Options;  using MyApp.Core.Models;  using OpenIddict.Abstractions;    namespace MyApp.Controllers  {      public class AuthController : Controller      {          private readonly IOptions<IdentityOptions> \_identityOptions;          private readonly SignInManager<ApplicationUser> \_signInManager;          private readonly UserManager<ApplicationUser> \_userManager;            public AuthController(              IOptions<IdentityOptions> identityOptions,              SignInManager<ApplicationUser> signInManager,              UserManager<ApplicationUser> userManager)          {              \_identityOptions = identityOptions;              \_signInManager = signInManager;              \_userManager = userManager;          }            [HttpPost("~/connect/token"), Produces("application/json")]          public async Task<IActionResult> Exchange([FromBody] OpenIdConnectRequest request)          {              Debug.Assert(request.IsTokenRequest(),                  "The OpenIddict binder for ASP.NET Core MVC is not registered. " +                  "Make sure services.AddOpenIddict().AddMvcBinders() is correctly called.");                if (request.IsPasswordGrantType())              {                  var user = await \_userManager.FindByNameAsync(request.Username);                  if (user == null)                  {                      return BadRequest(new OpenIdConnectResponse                      {                          Error = OpenIdConnectConstants.Errors.InvalidGrant,                          ErrorDescription = "The username/password couple is invalid."                      });                  }                    // Ensure the user is allowed to sign in.                  if (!await \_signInManager.CanSignInAsync(user))                  {                      return BadRequest(new OpenIdConnectResponse                      {                          Error = OpenIdConnectConstants.Errors.InvalidGrant,                          ErrorDescription = "The specified user is not allowed to sign in."                      });                  }                    // Reject the token request if two-factor authentication has been enabled by the user.                  if (\_userManager.SupportsUserTwoFactor && await \_userManager.GetTwoFactorEnabledAsync(user))                  {                      return BadRequest(new OpenIdConnectResponse                      {                          Error = OpenIdConnectConstants.Errors.InvalidGrant,                          ErrorDescription = "The specified user is not allowed to sign in."                      });                  }                    // Ensure the user is not already locked out.                  if (\_userManager.SupportsUserLockout && await \_userManager.IsLockedOutAsync(user))                  {                      return BadRequest(new OpenIdConnectResponse                      {                          Error = OpenIdConnectConstants.Errors.InvalidGrant,                          ErrorDescription = "The username/password couple is invalid."                      });                  }                    // Ensure the password is valid.                  if (!await \_userManager.CheckPasswordAsync(user, request.Password))                  {                      if (\_userManager.SupportsUserLockout)                      {                          await \_userManager.AccessFailedAsync(user);                      }                        return BadRequest(new OpenIdConnectResponse                      {                          Error = OpenIdConnectConstants.Errors.InvalidGrant,                          ErrorDescription = "The username/password couple is invalid."                      });                  }                    if (\_userManager.SupportsUserLockout)                  {                      await \_userManager.ResetAccessFailedCountAsync(user);                  }                    // Create a new authentication ticket.                  var ticket = await CreateTicketAsync(request, user);                  var result = SignIn(ticket.Principal, ticket.Properties, ticket.AuthenticationScheme);                  return result;              }              else if (request.IsRefreshTokenGrantType())              {                  // Retrieve the claims principal stored in the refresh token.                  var info = await HttpContext.AuthenticateAsync(OpenIdConnectServerDefaults.AuthenticationScheme);                    // Retrieve the user profile corresponding to the refresh token.                  // Note: if you want to automatically invalidate the refresh token                  // when the user password/roles change, use the following line instead:                  // var user = \_signInManager.ValidateSecurityStampAsync(info.Principal);                  var user = await \_userManager.GetUserAsync(info.Principal);                  if (user == null)                  {                      return BadRequest(new OpenIdConnectResponse                      {                          Error = OpenIdConnectConstants.Errors.InvalidGrant,                          ErrorDescription = "The refresh token is no longer valid."                      });                  }                    // Ensure the user is still allowed to sign in.                  if (!await \_signInManager.CanSignInAsync(user))                  {                      return BadRequest(new OpenIdConnectResponse                      {                          Error = OpenIdConnectConstants.Errors.InvalidGrant,                          ErrorDescription = "The user is no longer allowed to sign in."                      });                  }                    // Create a new authentication ticket, but reuse the properties stored                  // in the refresh token, including the scopes originally granted.                  var ticket = await CreateTicketAsync(request, user, info.Properties);                    return SignIn(ticket.Principal, ticket.Properties, ticket.AuthenticationScheme);              }                return BadRequest(new OpenIdConnectResponse              {                  Error = OpenIdConnectConstants.Errors.UnsupportedGrantType,                  ErrorDescription = "The specified grant type is not supported."              });          }            private async Task<AuthenticationTicket> CreateTicketAsync(OpenIdConnectRequest request,          ApplicationUser user,          AuthenticationProperties properties = null)          {              // Create a new ClaimsPrincipal containing the claims that              // will be used to create an id\_token, a token or a code.              var principal = await \_signInManager.CreateUserPrincipalAsync(user);                // Create a new authentication ticket holding the user identity.              var ticket = new AuthenticationTicket(principal,                  properties,                  OpenIdConnectServerDefaults.AuthenticationScheme);                if (!request.IsRefreshTokenGrantType())              {                  // Set the list of scopes granted to the client application.                  ticket.SetScopes(new[]                  {                      OpenIdConnectConstants.Scopes.OpenId,                      OpenIdConnectConstants.Scopes.Email,                      OpenIdConnectConstants.Scopes.Profile,                      OpenIdConnectConstants.Scopes.OfflineAccess,                      OpenIddictConstants.Scopes.Roles                  }.Intersect(request.GetScopes()));              }                ticket.SetResources("http://localhost:5000");                // Note: by default, claims are NOT automatically included in the access and identity tokens.              // To allow OpenIddict to serialize them, you must attach them a destination, that specifies              // whether they should be included in access tokens, in identity tokens or in both.              // principal.Claims.Append(new Claim ("test", "narbe", OpenIdConnectConstants.Destinations.AccessToken));               foreach (var claim in principal.Claims)              {                  // Never include the security stamp in the access and identity tokens, as it's a secret value.                  if (claim.Type == \_identityOptions.Value.ClaimsIdentity.SecurityStampClaimType)                  {                      continue;                  }                    var destinations = new List<string>                  {                      OpenIdConnectConstants.Destinations.AccessToken                  };                    // Only add the iterated claim to the id\_token if the corresponding scope was granted to the client application.                  // The other claims will only be added to the access\_token, which is encrypted when using the default format.                  if ((claim.Type == OpenIdConnectConstants.Claims.Name && ticket.HasScope(OpenIdConnectConstants.Scopes.Profile)) ||                      (claim.Type == OpenIdConnectConstants.Claims.Email && ticket.HasScope(OpenIdConnectConstants.Scopes.Email)) ||                      (claim.Type == OpenIdConnectConstants.Claims.Role && ticket.HasScope(OpenIddictConstants.Claims.Roles)))                  {                      destinations.Add(OpenIdConnectConstants.Destinations.IdentityToken);                  }                    claim.SetDestinations(destinations);              }                return ticket;          }      }  } |

Account Controller

There is one other controller which manages users and do works like register, change password and so on. Before implementing this controller let first create a folder called Resources under controllers folder. We will create our API resources in this folder, let’s create Register, ForgotPassword, ChangePassword, ResetPassword Resources.

|  |  |
| --- | --- |
| 1  2  3  4  5 | …> cd Resources  …/Controllers/Resources> echo.> RegisterResource.cs  …/Controllers/Resources> echo.> ForgotPasswordResource.cs  …/Controllers/Resources> echo.> ResetPasswordResource.cs  …/Controllers/Resources> echo.> ChangePasswordResource.cs |

RegisterResource

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using System.ComponentModel.DataAnnotations;    namespace MyApp.ViewModels  {      public class RegisterViewModel      {          [Required]          [EmailAddress]          public string Email { get; set; }            [Required]          [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]          [DataType(DataType.Password)]          public string Password { get; set; }              [Required]          [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]          [DataType(DataType.Password)]          public string ConfirmPassword { get; set; }      }  } |

ForgotPasswordResource

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | namespace MyApp.Controllers.Resources  {      public class ForgotPasswordResource      {          public string Email { get; set; }      }  } |

ChangePasswordResource

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System.ComponentModel.DataAnnotations;    namespace MyApp.Controllers.Resources  {      public class ChangePasswordResource      {          [Required]          public string OldPassword { get; set; }          [Required]          public string NewPassword { get; set; }          [Required]          public string ConfirmPassword { get; set; }      }  } |

ResetPasswordResource

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | using System.ComponentModel.DataAnnotations;    namespace MyApp.Controllers.Resources  {      public class ResetPasswordResource      {            public bool HasError { get; set; }            [Required]          [EmailAddress]          public string Email { get; set; }              [Required]          [StringLength(4)]          [RegularExpression(@"\d{4}")]          public string EmployeeNo { get; set; }            [Required]          [StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.", MinimumLength = 6)]          [DataType(DataType.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; }            [Required]          public string Code { get; set; }      }  } |

We also need to implement a fake EmailSender, so let’s create a Services folder and create IEmailSender as below:

|  |  |
| --- | --- |
| 1  2  3 | …> mkdir Services  …> cd Services  …/Services> echo.> IEmailSender.cs |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | using System.Threading.Tasks;    namespace MyApp.Services  {      public interface IEmailSender      {        Task SendEmailAsync(string email, string subject, string message);      }  } |

Then let’s create a FileEmailSender to save emails in a file for now.

|  |  |
| --- | --- |
| 1 | …/Services> echo.> FileEmailSender.cs |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | using System.IO;  using System.Threading.Tasks;    namespace MyApp.Services  {    public class FileEmailSender : IEmailSender    {      public Task SendEmailAsync(string email, string subject, string message)      {        var emailMessage = $"To: {email}\nSubject: {subject}\nMessage: {message}\n\n";                File.AppendAllText("emails.html", emailMessage);                return Task.FromResult(0);      }    }  } |

and Then add it to our services configuration in Startup.cs

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using MyApp.Services;    public class Startup  {    public void ConfigureServices(IServiceCollection services)    {     .     .     .     services.AddTransient<IEmailSender, FileEmailSender>();     .     .     .    }  } |

Now, we are ready to implement AccountController.cs

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182 | using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Authorization;  using Microsoft.AspNetCore.Identity;  using Microsoft.AspNetCore.Mvc;  using Microsoft.Extensions.Options;  using MyApp.Core.Models;  using MyApp.Persistence;  using MyApp.ViewModels;    namespace MyApp.Controllers  {      public class AccountController : Controller      {          private readonly UserManager<ApplicationUser> \_userManager;          private readonly SignInManager<ApplicationUser> \_signInManager;          private readonly IOptions<IdentityOptions> \_identityOptions;          private readonly ApplicationDbContext \_applicationDbContext;          private static bool \_databaseChecked;            public AccountController(              UserManager<ApplicationUser> userManager,              IOptions<IdentityOptions> identityOptions,              SignInManager<ApplicationUser> signInManager,              ApplicationDbContext applicationDbContext              )          {              \_userManager = userManager;              \_identityOptions = identityOptions;              \_signInManager = signInManager;              \_applicationDbContext = applicationDbContext;          }            [AllowAnonymous]          [HttpPost("~/api/auth/register")]          public async Task<IActionResult> Register([FromBody] RegisterViewModel model)          {              EnsureDatabaseCreated(\_applicationDbContext);                if (ModelState.IsValid)              {                  if(model.Password != model.ConfirmPassword)                  {                      return BadRequest(new { general = new[] {"Password and Confirm Password not match"} });                  }                    var user = new ApplicationUser                  {                      UserName = model.Email,                      Email = model.Email                  };                    var result = await \_userManager.CreateAsync(user, model.Password);                    if (result.Succeeded)                  {                      // we may need to send confirmation email after resigter succeed                      // Send an email with this link                      //var code = await \_userManager.GenerateEmailConfirmationTokenAsync(user);                      //var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code = code }, protocol: Context.Request.Scheme);                      //await \_emailSender.SendEmailAsync(model.Email, "Confirm your account",                      //    "Please confirm your account by clicking this link: <a href=\"" + callbackUrl + "\">link</a>");                      //===========================================================================                      //but we do nothing now just return ok                      return Ok();                  }                  else                  {                      return BadRequest(new { general = result.Errors.Select(x => x.Description).ToArray() });                  }              }              else              {                  return BadRequest(new { general = ModelState.SelectMany(x => x.Value.Errors)                      .Select(x => x.ErrorMessage).ToArray() });              }                // If we got this far, something failed, redisplay form               //return BadRequest(new { errors = "Bad Request".ToArray() });          }            [AllowAnonymous]          [HttpPost("~/api/auth/forgotPassword")]          public async Task<IActionResult> ForgotPassword([FromBody] ForgotPasswordResource model)          {              if(model == null || model.Email == null)            {              return BadRequest("Email not specified");            }              var user = await \_userManager.FindByEmailAsync(model.Email);            if(user == null)            {                BadRequest($"Unable to load user with ID '{model.Email}'.");            }              // For more information on how to enable account confirmation and password reset please visit http://go.microsoft.com/fwlink/?LinkID=532713            // Send an email with this link            var code = await \_userManager.GeneratePasswordResetTokenAsync(user);              var resetPasswordUrl = "https://localhost:5001/resetPassword/userId/" + user.Id + "/code/" + code;              await \_emailSender.SendEmailAsync(user.Email, "Reset Password",            "Please reset your password by clicking this link: <a href=\"" + resetPasswordUrl + "\">Reset Password</a>"            );              return Ok();            }              [HttpPost("~/api/auth/resetPassword")]          [AllowAnonymous]          public async Task<IActionResult> ResetPassword(ResetPasswordResource model)          {                NaravanUser user = await \_userManager.FindByEmailAsync(model.Email);              if (user == null)              {                      // Don't reveal that the user does not exist                      return BadRequest("the user does not exist");              }                var code = model.Code.Replace(" ", "+");                var resetResult = await \_userManager.ResetPasswordAsync(user, code, model.Password);              if (resetResult.Succeeded)              {                  return Ok();              }                return BadRequest(resetResult);          }              [HttpPost("~/api/auth/changePassword")]          [Authorize(AuthenticationSchemes = "Bearer")]          public async Task<IActionResult> ChangePassword(ChangePasswordResource model)          {              var claimsIdentity = User.Identity as ClaimsIdentity;              string username = claimsIdentity.Name;                NaravanUser user = await \_userManager.FindByNameAsync(username);                    if (user == null)                  {                      // Don't reveal that the user does not exist                      return BadRequest("the user does not exist");                  }                  if(model.NewPassword != model.ConfirmPassword)                  {                      return BadRequest("the password and confirm password not match.");                  }                  var resetResult = await  \_userManager.ChangePasswordAsync(user, model.OldPassword, model.NewPassword);              if (resetResult.Succeeded)              {                    return Ok();              }                return BadRequest(resetResult);          }             // The following code creates the database and schema if they don't exist.          // This is a temporary workaround since deploying database through EF migrations is          // not yet supported in this release.          // Please see this http://go.microsoft.com/fwlink/?LinkID=615859 for more information on how to do deploy the database          // when publishing your application.          private static void EnsureDatabaseCreated(ApplicationDbContext context)          {              if (!\_databaseChecked)              {                  \_databaseChecked = true;                  context.Database.EnsureCreated();              }          }      }  } |

Configure Authentication Services

Now we have our API endpoints for register and login users. Next step is to configure services to enable authentication. Open Startup.cs file and change ConfigureServices method as below:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79 | public void ConfigureServices(IServiceCollection services)          {              services.AddMvc();                services.AddDbContext<ApplicationDbContext>(o =>              {                  o.UseSqlServer(Configuration.GetConnectionString("Default"));                                  o.UseOpenIddict();              });                services.AddIdentity<ApplicationUser, IdentityRole>(                  options => {                      options.Password.RequiredLength = 4;                      options.Password.RequireUppercase = false;                      options.Password.RequireDigit = false;                      options.Password.RequireLowercase =false;                      options.Password.RequireNonAlphanumeric = false;                      options.Password.RequiredUniqueChars = 0;                  }              )              .AddEntityFrameworkStores<ApplicationDbContext>()              .AddDefaultTokenProviders();                services.Configure<IdentityOptions>(options =>                         {                             options.ClaimsIdentity.UserNameClaimType = OpenIdConnectConstants.Claims.Name;                             options.ClaimsIdentity.UserIdClaimType = OpenIdConnectConstants.Claims.Subject;                             options.ClaimsIdentity.RoleClaimType = OpenIdConnectConstants.Claims.Role;                         });                      // Register the OpenIddict services.              // Note: use the generic overload if you need              // to replace the default OpenIddict entities.              services.AddOpenIddict()              // Register the OpenIddict core services.              .AddCore(options =>              {                //options.                  // Register the Entity Framework stores and models.                  options.UseEntityFrameworkCore()                          .UseDbContext<ApplicationDbContext>();                })              .AddServer              (options =>              {                    options.UseMvc();                    // Enable the token endpoint (required to use the password flow).                  options.EnableTokenEndpoint("/connect/token");                    // Allow client applications to use the grant\_type=password flow.                  options.AllowPasswordFlow()                  .AllowRefreshTokenFlow();                    // Accept token requests that don't specify a client\_id.                  options.AcceptAnonymousClients();                    // During development, you can disable the HTTPS requirement.                  options.DisableHttpsRequirement();                    options.RegisterScopes(                      OpenIdConnectConstants.Scopes.OpenId,                      OpenIdConnectConstants.Scopes.Profile,                      OpenIdConnectConstants.Scopes.Email                  );                    // Note: to use JWT access tokens instead of the default                  // encrypted format, the following lines are required:                  //                  // options.UseJsonWebTokens();                  // options.AddEphemeralSigningKey();                });                services.AddAuthentication()              .AddOAuthValidation();          } |

Add app.UserAuthentication() to Configure method.

Note: you must add it before app.UseMvc()



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public void Configure(IApplicationBuilder app, IHostingEnvironment env)          {              if (env.IsDevelopment())              {                  app.UseDeveloperExceptionPage();                  app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions {                    HotModuleReplacement = true,                    ReactHotModuleReplacement = true                  });              }              else              {                  app.UseExceptionHandler("/Home/Error");              }                app.UseStaticFiles();               app.UseAuthentication();              app.UseMvc(routes =>              {                  routes.MapRoute(                      name: "default",                      template: "{controller=Home}/{action=Index}/{id?}");              });          } |

Don’t forget to add required namespaces using to startup.cs file

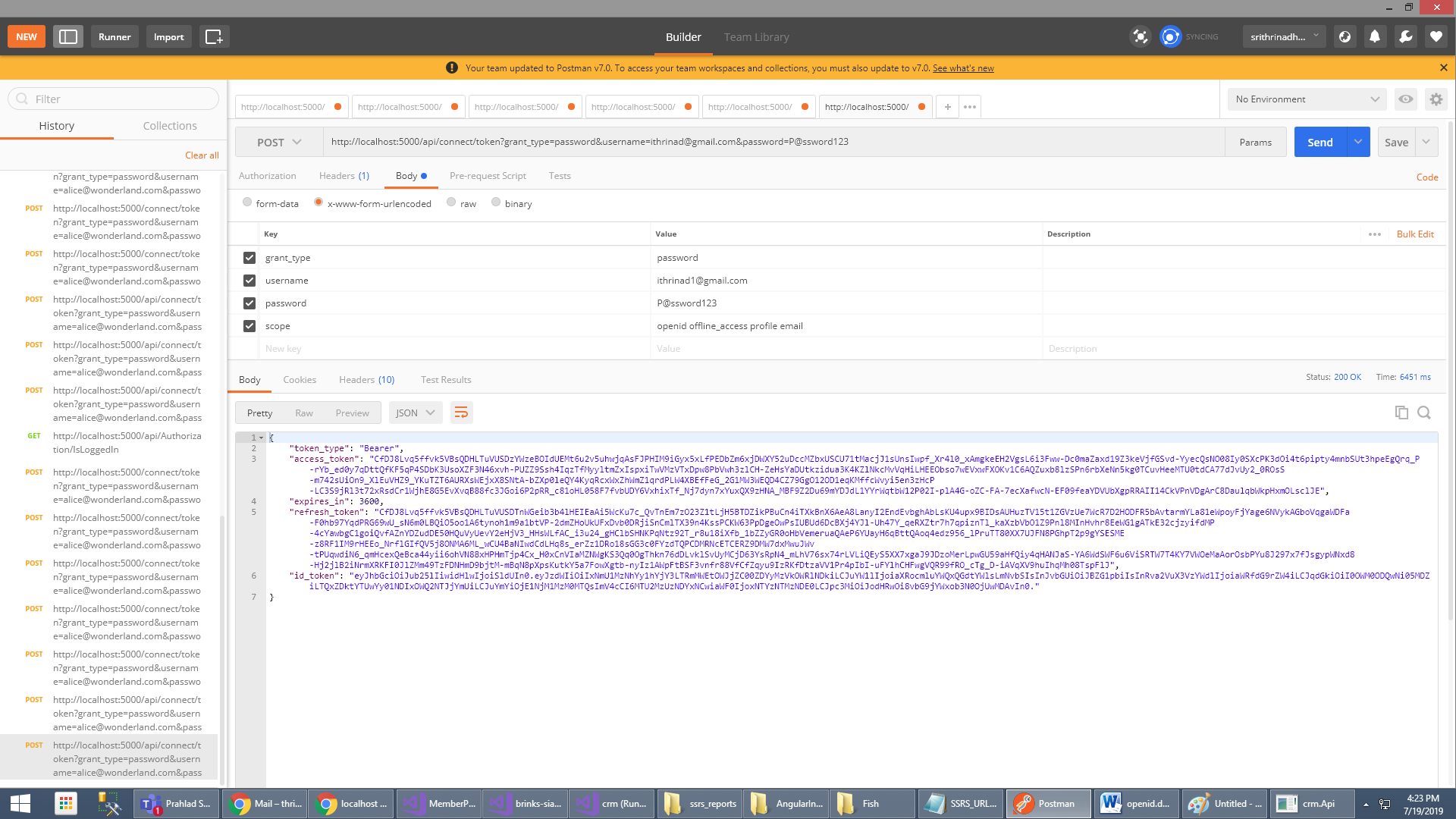
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Builder;  using Microsoft.AspNetCore.Hosting;  using Microsoft.AspNetCore.SpaServices.Webpack;  using Microsoft.Extensions.Configuration;  using Microsoft.Extensions.DependencyInjection;  using MyApp.Persistence;  using Microsoft.EntityFrameworkCore;  using MyApp.Core.Models;  using Microsoft.AspNetCore.Identity;  using AspNet.Security.OpenIdConnect.Primitives; |

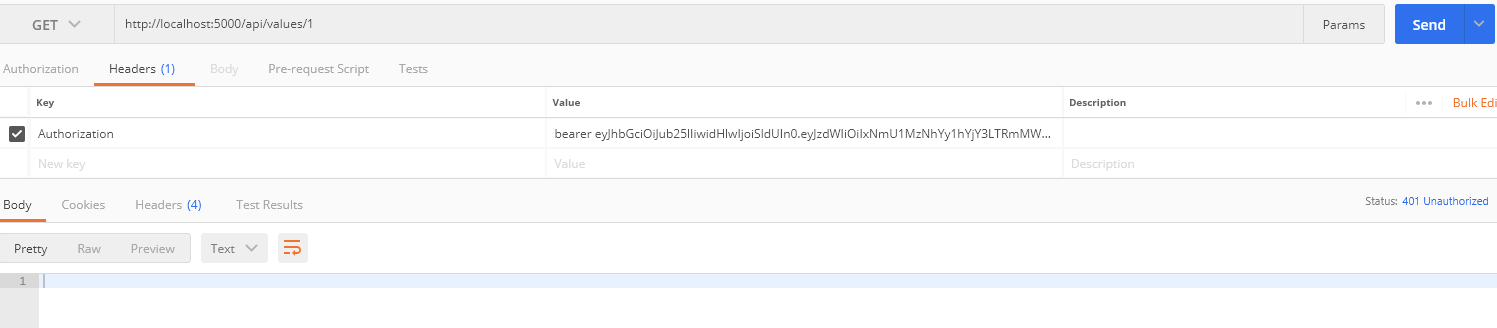
Run The Application

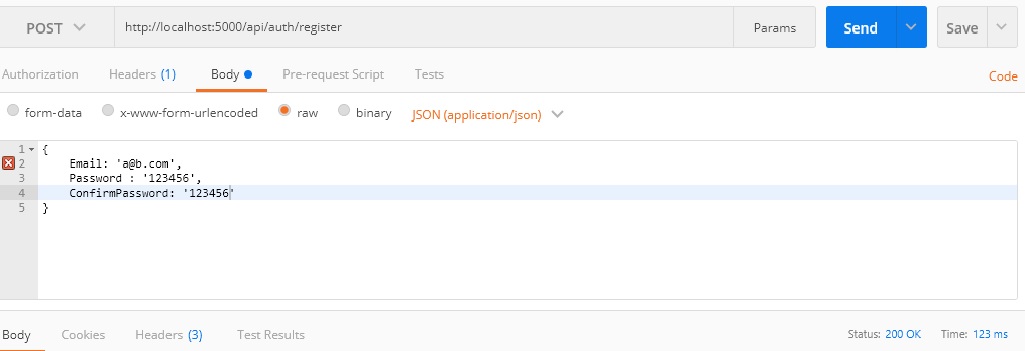
let’s test our app, first run the application.

|  |  |
| --- | --- |
| 1 | dotnet run |

then, open postman and create new **post** request to <http://localhost:5000/api/auth/register>and send Email, Password, and ConfirmPassword in the body of the form.







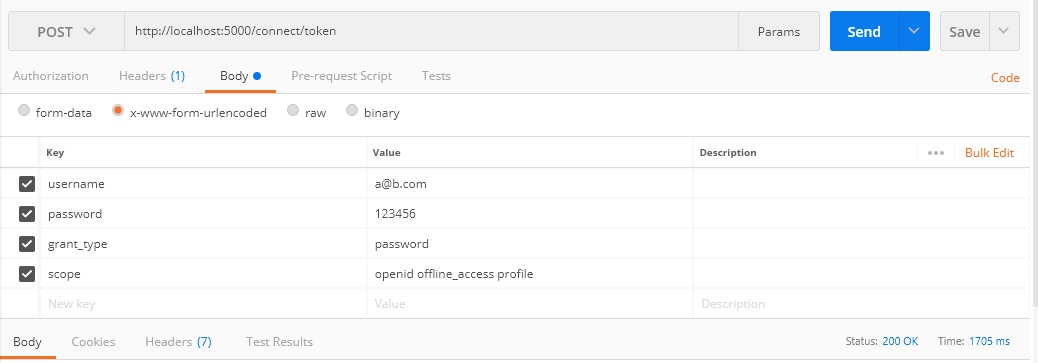
If everything goes fine you should see status 200 OK after sending the request.

If we try to login a user with our API, it will fail with an error that says: Invalid object name ‘OpenIddictAuthorizations’.

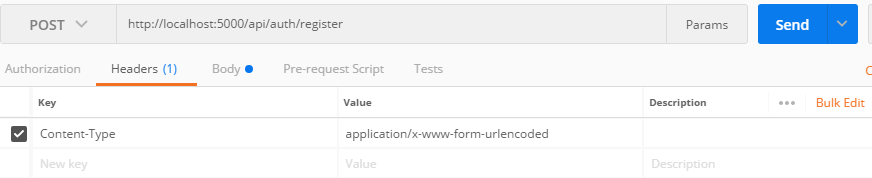
So Before we check login, we forgot to update our database to for openiddict tables. So, let create a new migration and update the database.

|  |  |
| --- | --- |
| 1  2 | dotnet ef migrations add addOpenIddictTables  dotnet ef database update |

Now we are good to go. So create a new request in postman with below parameters and send it to <http://localhost:5000/connect/token>



Don’t forget to set content-type of the request to application/x-www-form-urlencoded



After sending the request, you should see the response as below:



in the next chapter, we will create sigin-in / sign-up components.