Working With AuthAPI

We have coupon API and that is working as expected. When we think about a microservice architecture, we need some kind of authentication to make our API endpoint secure and only the allowed user with correct credentials can access our API endpoints. For such security requirement we will be using Dotnet identity and we will create a new API in our solution.

Go to Services folder and add a new empty web api project name it as Mango.Services.AuthAPI. Go to launch settings json file and change portid in https profile.

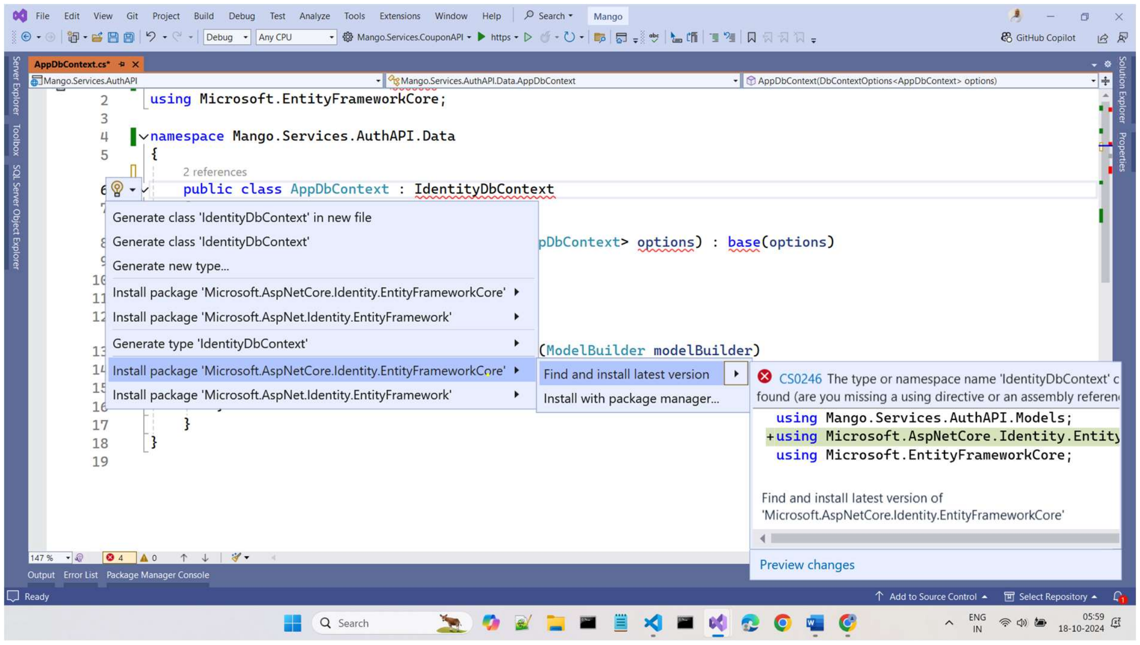
```
"https": {
  "commandName": "Project",
  "dotnetRunMessages": true,
  "launchBrowser": true,
  "launchUrl": "swagger",
  "applicationUrl": "https://localhost:7002;http://localhost:5241",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
},
```

Now we need to install NuGet packages because we need the Dotnet identity and authentication packages. What we can do, we have few packages that we have installed in the coupon API. Right click there, select the edit project file, copy all the packages from there, and we can paste it in the project file of auth API. That way, if we examine the dependencies, we have all the basic packages with entity framework core and automapper in the auth API.

```
<ItemGroup>
  <PackageReference Include="AutoMapper" Version="13.0.1" />
  <PackageReference Include="Microsoft.EntityFrameworkCore"
Version="8.0.10" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
Version="8.0.10">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
  </PackageReference>
  <PackageReference
Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.10"
/>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="8.0.10">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
```

```
  </PackageReference>
  <PackageReference Include="Swashbuckle.AspNetCore" Version="6.4.0"
/>
</ItemGroup>
```

We will be using net identity in order to use identity for authentication and authorization. You will have to implement from IdentityDbContext. Install the NuGet package using package manager or use IntelliSense, named Microsoft.AspNetCore.Identity.EntityFrameworkCore.



```
using Microsoft.EntityFrameworkCore;

namespace Mango.Services.AuthAPI.Data
{
    2 references
    public class AppDbContext : IdentityDbContext
```

Generate class 'IdentityDbContext' in new file
Generate class 'IdentityDbContext'
Generate new type...
Install package 'Microsoft.AspNetCore.Identity.EntityFrameworkCore'
Install package 'Microsoft.AspNet.Identity.EntityFramework'
Generate type 'IdentityDbContext'
Install package 'Microsoft.AspNetCore.Identity.EntityFrameworkCore'
Install package 'Microsoft.AspNet.Identity.EntityFramework'
Find and install latest version
Install with package manager...
Preview changes

CS0246 The type or namespace name 'IdentityDbContext' c... found (are you missing a using directive or an assembly referen...

```
 using Mango.Services.AuthAPI.Models;
+using Microsoft.AspNetCore.Identity.Entity
 using Microsoft.EntityFrameworkCore;
```

Find and install latest version of 'Microsoft.AspNetCore.Identity.EntityFrameworkCore'

Inside the context where we added the identity DB context, we can define the user right here. You can also see the default user is identity user.

```
public class AppDbContext : IdentityDbContext<IdentityUser>
{
    public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options)
    {
    }
    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
    }
}
```

Next thing that we have to do is configure Program.cs file. Copy that from coupon API to save some time there. We do not want automapper right now. We only want the DB context. paste that here. And scroll down where we have apply migration.

```
// Add services to the container.
```

```
builder.Services.AddDbContext<AppDbContext>(option =>
{
    option.UseSqlServer(builder.Configuration.GetConnectionString("D
efaultConnection"));
});
```

```
static void ApplyMigration(WebApplication? app)
{
    using (var scope = app.Services.CreateScope())
    {
        var _db =
scope.ServiceProvider.GetRequiredService<AppDbContext>();

        if (_db.Database.GetPendingMigrations().Count() > 0)
        {
            _db.Database.Migrate();
        }
    }
}
```

```
app.MapControllers();
ApplyMigration(app);
app.Run();
```

Now we know that auth API is responsible for authentication and authorization. So in the pipeline before authorization we have to add app.UseAuthentication();

When it comes to adding in the pipeline, Authentication must always come before Authorization.

Right now, we have only configured the DB context in our project. But in this project, we will have to tell entity framework code that, we will be using the Dotnet identity and we will be using entity framework core where we have the DB context with the Dotnet identity. That is how it will know that to use the DB context to create all the identity related tables.

Defining that is super simple on builder . services. We have something called as ADD identity and there you can define two things. What is the default user? That is identity user that comes by default. On top of that, you have to define the role. We will be using the default identity role for now.

We are setting up the default identity. We have to add the entity framework stores and we have to define the DB context that will act as a bridge between entity framework core and Dotnet identity. Finally, we will be using add default token providers there.

```
builder.Services.AddIdentity<IdentityUser,
IdentityRole>().AddEntityFrameworkStores<AppDbContext>()
.AddDefaultTokenProviders();
```

With that we have configured identity in this API project.

Now you might be thinking what exactly will happen because of this configuration that we added inside the Program.cs file? In order to see that, let me open package manager console, and the default project will be auth API. Now try to add the migration:

```
PM> add-migration AddIdentityTables
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
```

The migration will be added successfully. You can see here it is creating quite a few tables. We have ASP.NET Rules table, ASP.NET Users table with many columns. Here we have the role claims table, user claims table user logins, and much more. All of these tables are related to the net identity.

Now update database but before that go and edit to project files and, remove

<InvariantGlobalization>false</InvariantGlobalization>

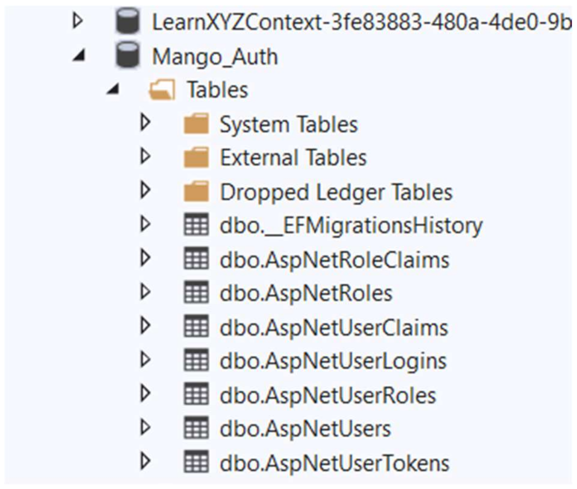
Also add connection string in appsettings.json file:

```
"ConnectionStrings": {
  "DefaultConnection":
"Server=(LocalDb)\\MSSQLLocalDB;Database=Mango_Auth;Trusted_Connecti
on=True;TrustServerCertificate=True"
},
```

Now update database using command in Package Manager.

```
PM> update-database
Build started...
Build succeeded..
PM>
```

Check your SQL Server after refreshing Database, you will find many tables will be created. These tables will be utilized in Authentication and Authorization.



```
▷ LearnXYZContext-3fe83883-480a-4de0-9b
◢ Mango_Auth
   ◢ Tables
      ▷ System Tables
      ▷ External Tables
      ▷ Dropped Ledger Tables
      ▷ dbo.__EFMigrationsHistory
      ▷ dbo.AspNetRoleClaims
      ▷ dbo.AspNetRoles
      ▷ dbo.AspNetUserClaims
      ▷ dbo.AspNetUserLogins
      ▷ dbo.AspNetUserRoles
      ▷ dbo.AspNetUsers
      ▷ dbo.AspNetUserTokens
```

The tables created here are extendable. You can add more properties into them.

Add one folder named Models to Auth project. Create a class named ApplicationUser in root of project.

```
public class ApplicationUser : IdentityUser
{
    public string Name { get; set; }
}
```

Now, change in the DB context. Open the DB context of auth API where we have identity user. This now will be application user. When we were working with the coupon API, we used to create a DB set for application user.

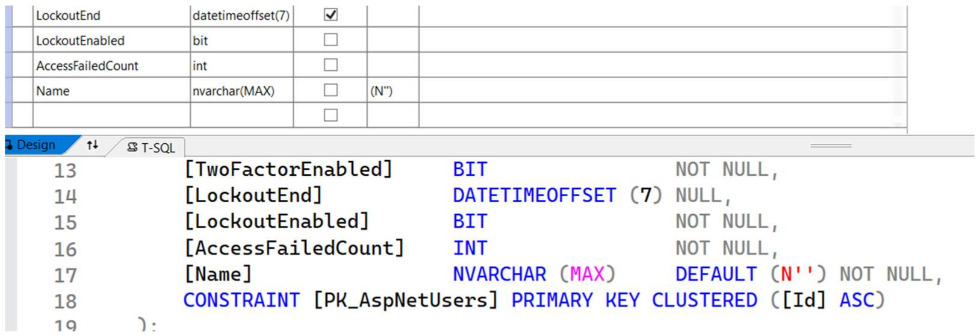
```
public class AppDbContext : IdentityDbContext<ApplicationUser>
{
    public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options)
    {
    }

    public DbSet<ApplicationUser> ApplicationUsers { get; set; }
    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
    }
}
```

Also modify the program.cs. When we are adding the identity, the default user now will be application user and not the identity user. With that change, we have added a DB set in the application DB context, so you might be thinking that it might create a new table with the name of application users, but it will not do that. Entity Framework core is smart. The .Net knows that application user is extending the identity user and it will add one more column to the ASP.NET Users table.

Add a new migration here named: AddNameToASPNetUsers.

```
PM> add-migration AddNameToASPNetUsers
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
```

You can see it is adding a column name to the ASP.NET Users table. If you run the project, we do not have that as the startup project and the multiple here. Let me select Auth API as well and run the project. It should automatically apply migration.


LockoutEnd	datetimeoffset(7)	✓	
LockoutEnabled	bit	☐	
AccessFailedCount	int	☐	
Name	nvarchar(MAX)	☐	(N'')
		☐	

```
        [TwoFactorEnabled]      BIT                 NOT NULL,
        [LockoutEnd]            DATETIMEOFFSET (7)  NULL,
        [LockoutEnabled]        BIT                 NOT NULL,
        [AccessFailedCount]     INT                 NOT NULL,
        [Name]                  NVARCHAR (MAX)      DEFAULT (N'') NOT NULL,
        CONSTRAINT [PK_AspNetUsers] PRIMARY KEY CLUSTERED ([Id] ASC)
    );
```

We have the name column right here. So, with that you can see adding new columns to the built in tables of identity is not that difficult.

Add endpoints

Now we want to work on the auth API and we need to create endpoints in the auth API. We have the default weather forecast controller and its model, delete both as we don't need them. In the Controllers folder, create a new controller using empty API controller named AuthAPIController.

Modify route name to [Route("api/auth")]

```
using Microsoft.AspNetCore.Mvc;

namespace Mango.Services.AuthAPI.Controllers
{
    [Route("api/auth")]
    [ApiController]
    public class AuthAPIController : ControllerBase
    {

    }
}
```

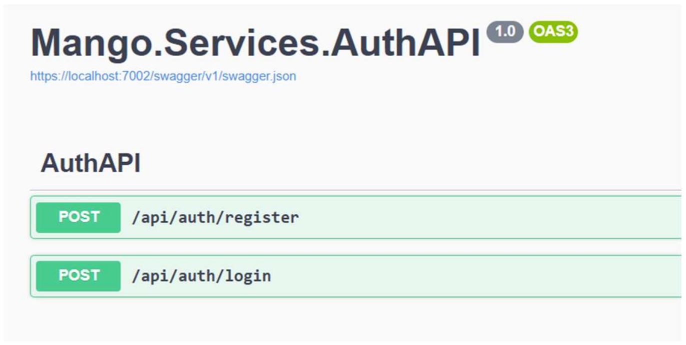
What will be the two end points in this controller?

First one will be to register an account and next one will be to login. Add to controller:

```
using Microsoft.AspNetCore.Mvc;

namespace Mango.Services.AuthAPI.Controllers
{
    [Route("api/auth")]
    [ApiController]
    public class AuthAPIController : ControllerBase
    {
        [HttpPost("register")]
        public async Task<IActionResult> Register()
        {
            return Ok();
        }
        [HttpPost("login")]
        public async Task<IActionResult> Login()
        {
            return Ok();
        }
    }
}
```

We have two API endpoints that we added in our controller.

Try executing but before that set the startup project here to be the auth API and run that. We see both the endpoints.



We have the end points for our auth API where we have register and login.

We need to think about what will be the DTOs or models that we will need for all of the endpoints. One thing that will be common when it comes to coupon API is the response. DTO let me copy the DTO folder here from coupon API. Paste that in models I will delete the coupon DTO we do not want that. Response DTO we need to make sure to change the namespace to be authAPI.

```
namespace Mango.Services.AuthAPI.Models.Dto
{
    public class ResponseDto
    {
        public object? Result { get; set; }
        public bool IsSuccess { get; set; } = true;
        public string Message { get; set; } = "";
    }
}
```

Now with auth API, we will be working with user application. User will have many properties that are not needed when we typically work with user. Like if we go to database, you can see application user has logout enabled access, failed count, concurrency stamp, security stamp. All of those are not needed in the details of auth API.

Let me add a class for userDto and in there we will add the critical properties that we care about user. The main four properties here will be userID, and userID will be a string and not an integer as that is the default type for user ID in Dotnet identity.

Next, we will have email of the user, name of the user that we just added, and there is also a default property for a phone number. Then first we want to think about all the models that are needed when we register a user.

```
namespace Mango.Services.AuthAPI.Models.Dto
{
    public class UserDto
    {
        public string ID { get; set; }
        public string Email { get; set; }
        public string Name { get; set; }
        public string PhoneNumber { get; set; }
    }
}
```

At that point, we will require name, email, phone number, and on top of these three we will also require password that a user wants to set for their account.

In the data here. Let me add a class here named RegistrationRequestDto and we will copy the three properties. On top of that I will add password right here that looks good for the registration request.

```
namespace Mango.Services.AuthAPI.Models.Dto
{
    public class RegistrationRequestDto
    {
        public string Email { get; set; }
        public string Name { get; set; }
        public string PhoneNumber { get; set; }
        public string Password { get; set; }
    }
}
```

Similarly, when we have to log in, we need a DTO for that as well. I will name that LoginRequestDto. When a user wants to log in, they will typically send username and password. So here we will have two properties.

```
namespace Mango.Services.AuthAPI.Models.Dto
{
    public class LoginRequestDto
    {
        public string UserName { get; set; }
        public string Password { get; set; }
    }
}
```

If a user is able to log in successfully, we will return a user DTO of the logged in user. On top of that, we will have to return something called as a token. That is the basic security where we return the JWT token to validate that okay, you are logged in and this is your special token that will prove that you are logged in successfully in the DTO.

Let me create one more class and I will call that LoginResponseDto. There we want two things, first is the user data and next one will be a string that will be token.

```
namespace Mango.Services.AuthAPI.Models.Dto
{
    public class LoginResponseDto
    {
        public UserDto User { get; set; }
        public string Token { get; set; }
    }
}
```

Configure JWT

When we work with authentication, we deal with creating JWT token that is used to validate a user that has logged in. You can see in the login response we added a string for that token.

Now that token is generated with the help of a secret key, and there are few more settings that you can configure. I will add all of the settings in one place. That is appsettings.json.

Let me create a new section named ApiSettings. In there we will create a section with the name of JwtOptions and there we want three key value names. First is the secret key that we will use, and you can make up your own secret key.

Next thing is an issuer and this will basically be like, who issued this certificate or this token. We can hardcode that to be mango-auth-api here.

And then we have something called as an audience. Audience will basically be that, okay, this token has been generated, but it has been generated for certain audience and we can give that a name of mango-client.

```
  "ApiSettings": {
    "JwtOptions": {
      "Secret": "THIS IS USED TO SIGN AND VERIFY JWT TOKENS, REPLACE IT WITH YOUR OWN SECRET",
      "Issuer": "mango-auth-api",
      "Audience": "mango-client"
    }
  },
```

What that will do is if some other audience tries to pass the token, it will treat that token as an invalid token.

Now we have added these settings inside the app settings. If you want to access them individually, you can get the configuration object, but there is other way of accessing that in your API. What we can do is in the models here, let me create a new class with the exact name that we have here, JwtOptions and in that class we will create three properties with the name of secret issuer and audience. We will set all of them by default to be string.empty.

```
namespace Mango.Services.AuthAPI.Models
```

```
{
    public class JwtOptions
    {
        public string Issuer { get; set; } = string.Empty;
        public string Audience { get; set; } = string.Empty;
        public string Secret { get; set; } = string.Empty;
    }
}
```

What we can do is in Program.cs, when we are configuring the pipeline, we can configure the JWT option. All of these properties should be populated with the values from app settings.

```
builder.Services.Configure<JwtOptions>(builder.Configuration.GetSect
ion("ApiSettings:JwtOptions"));
```

That will basically configure the class file and using dependency injection, we can access the settings that we have right here. Now, when the time is right, we will access that but right now we have configured that in our container.

Now we want to add services to our AuthApi, create a new folder for that, and to organize everything in a better way, we will create Service folder. Also, inside service folder add one folder named IService there and we will add our first interface in there named IAuthService.

Think about what will be the endpoints in this interface. We will have something for login and register. When a user is registering, what will be the return type that will be user DTO? Let me call the method as register here and what will we receive as input parameters, that will be the registration RequestDto.

Now when a user is logging in the return type, we have that as LoginResponseDto, method name will be login and the parameter that we will receive will be LoginRequestDto.

Then we need to implement the auth service.

```
using Mango.Services.AuthAPI.Models.Dto;

namespace Mango.Services.AuthAPI.Service.IService
{
    public interface IAuthService
    {
        Task<UserDto> Register(RegistrationRequestDto
registrationRequestDto);
        Task<LoginResponseDto> Login(LoginRequestDto
loginRequestDto);
    }
}
```

Next, In the service folder, we will add a class file AuthService and it will implement the IAuthService.

Now, you might be wondering that when we register a user, we have to create a record in this table. Maybe some other tables related to identity and the password hash that we have here. We will have to hash the password and do all the complex security. That will not be the case with Dotnet identity.

They have given us many helper methods to automatically create the user, secure the password, assign roles if we have to, and much more. To access all of those helper functions that we have with Dotnet identity, we will have to inject some things with dependency injection.

Now, first, when we register our log in a user, we will be updating our database. So, we will need the application DB context. So read only we have the app DB context call that _db and then we want to inject the helper methods.

First one here is a user manager. If you write user manager here, we have to define the user. We are not using the default user. We are using application user and I will call that _userManager.

When we have to work with the roles in the user, we will be using something called as role manager. That will be a read only role manager and you can define the default role. Now we will be using the default roles of the application.

So right here let me write identity, role and role manager. I have to inject all three of them with dependency injection. Let me do that here. And we will say _db, _userManager and _roleManager.

```
using Mango.Services.AuthAPI.Service.IService;
using Microsoft.AspNetCore.Identity;
namespace Mango.Services.AuthAPI.Service
{
    public class AuthService : IAuthService
    {
        private readonly AppDbContext _db;
        private readonly UserManager<ApplicationUser> _userManager;

        public AuthService(AppDbContext db,
UserManager<ApplicationUser> userManager)
        {
            _db = db;
            _userManager = userManager;

        }
```

These helper methods are injected automatically and we do not have to do any other configuration. With that injected now we can Implement the register and login functionalities.

Let me work on register right here. When we receive the registration request DTO, we get few properties. We need to create a new application user with these properties.

```
public async Task<UserDto> Register(RegistrationRequestDto
registrationRequestDto)
{
    ApplicationUser user = new()
    {
        UserName = registrationRequestDto.Email,
        Email = registrationRequestDto.Email,
        NormalizedEmail = registrationRequestDto.Email.ToUpper(),
        Name = registrationRequestDto.Name,
        PhoneNumber = registrationRequestDto.PhoneNumber
    };
    try
    {
        var result = await _userManager.CreateAsync(user,
registrationRequestDto.Password);
        if (result.Succeeded)
        {
            var userToReturn = _db.ApplicationUsers.First(u =>
u.UserName == registrationRequestDto.Email);

            UserDto userDto = new()
            {
                Email = userToReturn.Email,
                ID = userToReturn.Id,
                Name = userToReturn.Name,
                PhoneNumber = userToReturn.PhoneNumber
            };
            return userDto;
        }
    }
    catch (Exception ex)
    {

    }
    return new UserDto();
}
```

Right now, we are returning the user DTO, when the registration is successful. But I think a better approach will be to return a string and that will be empty If the registration was successful. If it failed, we will return the error message. You can see we have result.succeeded here.

So, if that was not successful, we can populate that error message. But if everything was successful, we will return empty. In the else part here we can say return result.Errors.FirstOrDefault(), because that is a list and this errors is basically the identity error that has code and description, we will take description that is more meaningful.

For some reason, if it catches an exception, we will return error encountered. That looks good in the Auth service. When we register, we will return a string right there. With that the auth service looks good. Finally, code looks alike.

```
public async Task<string> Register(RegistrationRequestDto
registrationRequestDto)
{
    ApplicationUser user = new()
    {
        UserName = registrationRequestDto.Email,
        Email = registrationRequestDto.Email,
        NormalizedEmail = registrationRequestDto.Email.ToUpper(),
        Name = registrationRequestDto.Name,
        PhoneNumber = registrationRequestDto.PhoneNumber
    };

    try
    {
        var result = await _userManager.CreateAsync(user, 
registrationRequestDto.Password);
        if (result.Succeeded)
        {
            var userToReturn = _db.ApplicationUsers.First(u => 
u.UserName == registrationRequestDto.Email);

            UserDto userDto = new()
            {
                Email = userToReturn.Email,
                ID = userToReturn.Id,
                Name = userToReturn.Name,
                PhoneNumber = userToReturn.PhoneNumber
            };

            return "";

        }
        else
        {
            return result.Errors.FirstOrDefault().Description;
        }

    }
    catch (Exception ex)
    {

    }
    return "Error Encountered";
}
```

In order to consume that, we will do that in the endpoint that we created. Right here we need two things, first, we need a response DTO, and then we need to inject the auth service. Let me inject the auth service and let me add the response DTO.

```
public class AuthAPIController : ControllerBase
{
    private readonly IAuthService _authService;
    protected ResponseDto _response;

    public AuthAPIController(IAuthService authService)
    {
        _authService = authService;
        _response = new();
    }
```

Now, in the register here

```
[HttpPost("register")]
public async Task<IActionResult> Register([FromBody]
RegistrationRequestDto model)
{
    var errorMessage = await _authService.Register(model);
    if (!string.IsNullOrEmpty(errorMessage))
    {
        _response.IsSuccess = false;
        _response.Message = errorMessage;
        return BadRequest(_response);
    }
    return Ok(_response);
}
```

In program.cs use

```
builder.Services.AddIdentity<ApplicationUser,
IdentityRole>().AddEntityFrameworkStores<AppDbContext>().AddDefaultT
okenProviders();
builder.Services.AddTransient<IAuthService, AuthService>();
```

Now try running API.

We have registered our first user. Next, what we can do is using the same credentials, we can implement the login functionality. First, we have to do that inside the auth service. Based on the login request. DTO we will get the username and password. We will first retrieve a user from database based on that user name.

If that username is valid, then the user will be populated. If a user is populated, then we can check their password. And in order to check password we will be using user manager. We have helper method there with the name of check password async. It is an asynchronous method, so make sure you await that method. We have to pass the user and login request DTO password. We will check here if user is null or if is valid is false. That means the combination is invalid. In that case we can return new login response DTO and inside there we can set user is null. And token will also be empty. But if the user was found, we need to generate the JWT token. Before we generate the token, let me populate the user DTO right here and I will also have the login response DTO. Inside there. We will pass the user object and token. For now, we will keep that empty, but we will generate the token here. And once we generate that, we will assign that token right here. Finally, I will return the login response DTO and I believe that looks good.

```
public async Task<LoginResponseDto> Login(LoginRequestDto
loginRequestDto)
{
    var user = _db.ApplicationUsers.FirstOrDefault(u =>
u.UserName.ToLower() == loginRequestDto.UserName.ToLower());
    bool isValid = await _userManager.CheckPasswordAsync(user,
loginRequestDto.Password);
    if (user == null || isValid == false)
    {
        return new LoginResponseDto() { User = null, Token = "" };
    }
    //Here if user was found, Generate JWT Token
    UserDto userDTO = new()
    {
        Email = user.Email,
        ID = user.Id,
        Name = user.Name,
        PhoneNumber = user.PhoneNumber
    };
    LoginResponseDto loginResponseDto = new LoginResponseDto()
    {
        User = userDTO,
        Token = ""
    };
    return loginResponseDto;
}
```

With that configured, let me consume that in the auth API controller where we have the login end point. We will say variable login response is equal to await underscore auth service dot login. We need to pass the login request. DTO we will get that in the from body here and we need to pass that model. Then we can check if login response dot user is null. That means the authentication was invalid, so, we will set the is success to be false, set the message and return back a bad request. But if that was successful, we will set the result to be log in response and return back in.

```
[HttpPost("login")]
public async Task<IActionResult> Login([FromBody] LoginRequestDto
model)
```

```
{
    var loginResponse = await _authService.Login(model);
    if (loginResponse.User == null)
    {
        _response.IsSuccess = false;
        _response.Message = "Username or password is incorrect";
        return BadRequest(_response);
    }
    _response.Result = loginResponse;
    return Ok(_response);
}
```

Run the application and see if that works.

Now, the next thing that we need to work on is when the login is successful. We need to generate a JWT token. We want to generate a token for the logged in user. Rather than clustering everything in the auth service. Let me create a new interface here. We will call that IJwtTokenGenerator.

The token will be a string, so that will be the return type. Let me call the method name as GenerateToken and the input to generate that token will be the ApplicationUser, that we will pass right here.

```
using Mango.Services.AuthAPI.Models;
namespace Mango.Services.AuthAPI.Service.IService
{
    public interface IJwtTokenGenerator
    {
        string GenerateToken(ApplicationUser applicationUser);
    }
}
```

Let me create the implementation for that inside service folder. I will create JWT token generator and that will implement the JWT token generator. Implement that interface.

When we generate token, we will be configuring our token with secret issuer and audience. For that we will require JWT option. And if we examine Program.cs, we configured that right here. And because of that, inside the service here, we can retrieve that in constructor using dependency injection.

We want to inject the JWT option here. Let me do that.

```
using Mango.Services.AuthAPI.Models;
using Mango.Services.AuthAPI.Service.IService;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
```

```
namespace Mango.Services.AuthAPI.Service
{
    public class JwtTokenGenerator : IJwtTokenGenerator
    {
        private readonly JwtOptions _jwtOptions;
        public JwtTokenGenerator(IOptions<JwtOptions> jwtOptions)
        {
            _jwtOptions = jwtOptions.Value;
        }
        public string GenerateToken(ApplicationUser applicationUser)
        {
            var tokenHandler = new JwtSecurityTokenHandler();

            var key = Encoding.ASCII.GetBytes(_jwtOptions.Secret);

            var claimList = new List<Claim>
            {
                new
Claim(JwtRegisteredClaimNames.Email,applicationUser.Email),
                new
Claim(JwtRegisteredClaimNames.Sub,applicationUser.Id),
                new
Claim(JwtRegisteredClaimNames.Name,applicationUser.UserName)
            };
            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Audience = _jwtOptions.Audience,
                Issuer = _jwtOptions.Issuer,
                Subject = new ClaimsIdentity(claimList),
                Expires = DateTime.UtcNow.AddDays(7),
                SigningCredentials = new SigningCredentials(new
SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
            };

            var token = tokenHandler.CreateToken(tokenDescriptor);
            return tokenHandler.WriteToken(token);
        }
    }
}
```

And before we implement the generate token, let me go to Program.cs and I will also register that service.

```
builder.Services.AddScoped<IJwtTokenGenerator, JwtTokenGenerator>();
builder.Services.AddScoped<IAuthService, AuthService>();
```

We have added the method to generate token in the JWT token generator and register it in Program.cs.

Let me consume that inside our auth service. We will have to inject the IJwtTokenGenerator. We already configured that in the PROGRAM.CS file. So let me inject here with dependency injection.

```
public class AuthService : IAuthService
{

    private readonly AppDbContext _db;
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly IJwtTokenGenerator _jwtTokenGenerator;

    public AuthService(AppDbContext db, IJwtTokenGenerator 
jwtTokenGenerator, UserManager<ApplicationUser> userManager)
    {
        _db = db;
        _jwtTokenGenerator = jwtTokenGenerator;
        _userManager = userManager;
    }
```

And in Login:

```
//if user was found, Generate JWT Token
var token = _jwtTokenGenerator.GenerateToken(user, roles);
```

and assign this token to LoginResponseDto as:

```
LoginResponseDto loginResponseDto = new LoginResponseDto()
{
    User = userDTO,
    Token = token
};
```

Try executing and you can see token there.

Applying Role Base Authentication

We have configured our login and register on top of that I want to add one more endpoint when we want to assign role to any user. Go to the SQL Server and examine the auth table, you can notice we have AspNetRoles table, but that is empty.

First, we need to create role in the AspNetRoles table, and then we have a mapping table AspNetUserRoles where we can pass the userID and roleID to assign some role to our user.

What I mean by role is let's say there is an admin role and a customer role. Customer can access few things as long as they are logged in. But admin is the only one who is able to, let's say, create a coupon or delete a coupon. And for that, when we register a user, we need to assign role to a user.

Let me go back to the application and inside the IAuthService interface, we will create one more method that will return a Boolean flag, and I will call that AssignRole.

```
Task<bool> AssignRole(string email, string roleName);
```

We will assign a role based on email and roleName strings, whatever role name we pass here, we will assign that to a user with the email address that we provide in the parameter.

To implement this interface method, we will go back to the AuthService. Also, In the dependency we will add role manager that will provide us with the helper method to create a role in our database.

```
private readonly RoleManager<IdentityRole> _roleManager;
```

Add to constructor a role manager

```
public AuthService(AppDbContext db, IJwtTokenGenerator
jwtTokenGenerator, UserManager<ApplicationUser> userManager,
RoleManager<IdentityRole> roleManager)
{
    _db = db;
    _jwtTokenGenerator = jwtTokenGenerator;
    _userManager = userManager;
    _roleManager = roleManager;
}
```

Now implement AssignRole method:

Here we need to retrieve user based on this email. We will check the email. That way we will retrieve any user based on the email. If the user is not null, then we want to proceed further before we assign role to a user. We need to make sure that that particular role is created in our database. To check if a role already exists or not. We will send _roleManager.

We have a helper method RoleExistsAsync. Now here we are checking if the role exists or not and we need to pass the roleName. But in the if condition, you can see it will not work because this is an asynchronous method. To make that synchronous. We can call, GetAwaiter.GetResult and that way we do not have to write the await keyword.

Inside the if condition, we need to create a role because that does not exist. To create that on role manager, we have the createAsync and that expects an IdentityRole, here we need to pass the role name and we have to add the GetAwaiter.GetResult(). That will create a role if it does not exist and then we have to assign user to a particular role. We can do that using _userManager.AddToRoleAsync.

```
public async Task<bool> AssignRole(string email, string roleName)
{
    var user = _db.ApplicationUsers.FirstOrDefault(u =>
u.Email.ToLower() == email.ToLower());
```

```
    if (user != null)
    {
        if
(!_roleManager.RoleExistsAsync(roleName).GetAwaiter().GetResult())
        {
            //create role if it does not exist
            _roleManager.CreateAsync(new
IdentityRole(roleName)).GetAwaiter().GetResult();
        }
        await _userManager.AddToRoleAsync(user, roleName);
        return true;
    }
    return false;
}
```

When you are working with a role, remember that you can work with one role or multiple roles. We want to add one role, so we will use AddToRoleAsync and that expects the application user and the role name. We will pass both of them and finally we will return True. If the user is null, we can return false here.

Add the endpoint in the auth API.

```
[HttpPost("AssignRole")]
public async Task<IActionResult> AssignRole([FromBody]
RegistrationRequestDto model)
{
    var assignRoleSuccessful = await
_authService.AssignRole(model.Email, model.Role.ToUpper());
    if (!assignRoleSuccessful)
    {
        _response.IsSuccess = false;
        _response.Message = "Error encountered";
        return BadRequest(_response);
    }
    return Ok(_response);

}
```

We will assign a role when a user will be registered from the front-end application. So let me expect the same DTO registration request DTO. Here we have assigned role we need to pass model.Email and the role name. We do not have role name inside registration request DTO. Add it there.

```
public class RegistrationRequestDto
{
    public string Email { get; set; }
    public string Name { get; set; }
    public string PhoneNumber { get; set; }
    public string Password { get; set; }
    public string? Role { get; set; }
}
```

Save it and now pass model.Role and convert that to an uppercase. That way we do not have any casing issues when we are creating Role. Let me change the variable name here to assignRoleSuccessful. And if that is not successful, we want to display error message here error encountered. If everything is successful, we return the response back.

Run the application and see if that works. Register a new user here and when we are registering give that a role of admin. Now, even though when we are registering, we are not using that, but I still added that right here. Perfect user is created.

Same schema use to assign role endpoint. Run it and afterwords check respective tables.