A intenção desse documento é gerar um registro para criar um projeto .Net core, hoje na versão 3.1, com Entity Framework, Migration e JWT.

Depois de criarmos o nosso projeto precisamos, no meu caso, uma API rest, vamos complementar nosso appsettings.json informando nossa connection string como no exemplo a seguir:

"ConnectionStrings": {

   "DefaultConnection": "Server=localhost;Database=AccessControl;User ID=developer; Password=123456"

 },

Através do Nuget podemos inserir referências para o Microsoft.EntityFrameworkCore, Microsoft.Extensions.DependencyInjection, e Microsoft.EntityFrameworkCore.Design.

Podemos dizer que a class DbContext é uma das, se não a mais importante classe do EF e para que possamos fazer uso do

Entity, é necessário criar uma classe que herde de DbContext.

No final da criação do nosso projeto teremos duas classes com essa herença, uma para o negócio propriamente dito e outra para que possamos fazer uso Identity.

A primeira de se parecer com:

using Microsoft.EntityFrameworkCore;

...

    public class AccessContext : DbContext

    {

        public AccessContext(DbContextOptions<AccessContext> options) : base(options)

        {

        }

        public DbSet<User> User { get; set; }

    }

Vamos alterar a oi método ConfigureService daclasse Startup para que fique assim:

public void ConfigureServices(IServiceCollection services)

   {

       services.AddDbContext<AccessContext>(options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"), b => b.MigrationsAssembly("AccessControl")));

     services.AddScoped<IUserRepository, UserRepository>();

     services.AddControllers();

   }

Por fim alteramos a classe Program para que ficasse da seguinte forma:

public static void Main(string[] args)

       {

           try

           {

               var host = CreateHostBuilder(args).Build();

               using (var scope = host.Services.CreateScope())

               {

                   var services = scope.ServiceProvider;

                   var context = services.GetRequiredService<AccessContext>();

                   context.Database.EnsureCreated();

               }

               host.Run();

           }

           catch (Exception exp)

           {

               throw new NotImplementedException();

           }

       }

Ao executar a aplicação, as tabelas do Banco de Dados devem ser ciadas automaticamente.

MIGRATION

Para que possamos fazer uso do Migration precisamos instalar o pacote Microsoft.EntityFrameworkCore.Tools. Feito isso basta executar o comando

Add-Migration Initial (o nome do pacote de ajustes criado é definido pelo usuário. Pode ser Initial o outro qualquer) e os arquivos de migração serão criados.

Para aplicar essa mudanças no nosso banco de dados, basta executarmos o comando Update-Database.

Repare que nesse ponto podemos alterar a classe Program removento a criação do banco de dados.

Criando uma api

Existem pelo menos duas formas de criarmos a nossa api, onde podemos definir em, com e sem repositor

Se não quisermos usar esse padrão podemos simplesmente ter um método Get, por exemplo e receber nosso contexto diretamente por DI, uma vez que definimos isso no nosso Startup.sc.

// GET: api/User

       [HttpGet]

       public async Task<ActionResult<List<User>>> Get([FromServices] AccessContext context)

       {

           var users = await context.User.ToListAsync();

           return users;

       }

Mas se quisermos usar a outra forma precisaremos definir isso no construtor assim:

private AccessContext \_context;

       public UserController(AccessContext context)

       {

           \_context = context;

           \_userRepository = new UserRepository(\_context);

       }

E nesse caso, o método Get ficaria assim:

[HttpGet]

       public async Task<ActionResult<List<User>>> Get()

       {

           var users = await \_userRepository.GetAll();

           return users;

       }

public class UserRepository : IUserRepository

   {

       private readonly AccessContext \_context;

       public UserRepository(AccessContext context)

       {

           \_context = context;

       }

       public async Task<List<User>> GetAll()

       {

           return await \_context.User.ToListAsync();

       }

   }

Identity

Agora vamos criar um segundo contexto para o Identity

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

namespace AccessControl.Infrastructure

{

    public class IdentityContext : IdentityDbContext

    {

        public IdentityContext(DbContextOptions<IdentityContext> options) : base(options)

        {

        }

    }

}

Vamos inserir a referência para esse contexto também no Startup.sc. Ele deve ficar assim:

public void ConfigureServices(IServiceCollection services)

       {

           services.AddDbContext<AccessContext>(options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"), b => b.MigrationsAssembly("AccessControl")));

           services.AddDbContext<IdentityContext>(options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"), b => b.MigrationsAssembly("AccessControl")));

           services.AddScoped<IUserRepository, UserRepository>();

           services.AddIdentity<IdentityUser, IdentityRole>()

                  .AddEntityFrameworkStores<IdentityContext>()

                  .AddDefaultTokenProviders();

           services.AddControllers();

       }

Depois dessas mudanças, para executarmos o Add-Migration e o Update-Databade vamos precisar definir nesses comandos o nome do context,

Add-Migration Identity-first -Context IdentityContext

Update-Database -Context IdentityContext

Alteramos um pouco o nosso controller para que se pareça com isso:

public class UserController : ControllerBase

   {

       private readonly IUserRepository \_userRepository;

       private readonly SignInManager<IdentityUser> \_singnManager;

       private readonly UserManager<IdentityUser> \_userManager;

       private AccessContext \_context;

       public UserController(AccessContext context, SignInManager<IdentityUser> singnManager, UserManager<IdentityUser> userManager)

       {

           \_singnManager = singnManager;

           \_userManager = userManager;

           \_context = context;

           \_userRepository = new UserRepository(\_context);

       }

...

public async Task<ActionResult<User>> Post([FromBody] User user, [FromServices] AccessContext context)

       {

User user = new User

          {

              Name = createUser.Name,

              LastName = createUser.LastName,

              BirthDate = createUser.BirthDate

          };

          var response = await context.User.AddAsync(user);

          var identityUser = new IdentityUser

          {

              Id = user.Id,

              UserName = createUser.Name,

              Email = createUser.Email,

              EmailConfirmed = true,

          };

          var result = await \_userManager.CreateAsync(identityUser, createUser.Password);

          if (!result.Succeeded) return BadRequest(result.Errors);

          await context.SaveChangesAsync();

          return Ok(response.Entity);

       }

...

Criando o método de login

[HttpPost("login")]

       public async Task<ActionResult<User>> Login([FromBody] LoginUser loginUser, [FromServices] AccessContext context)

       {

           ...

           var result = await \_singnManager.PasswordSignInAsync(loginUser.Login, loginUser.password, false, true);

           ...

       }

Fora o login e senha, o método PasswordSignAsync espera mais dois parâmetros do tipo bolean. O primeiro confirma se esse valor pode ser relembrado, com o uso de cokkies por exemplo e o segundo se o usuário vai ter um limite de tentativas