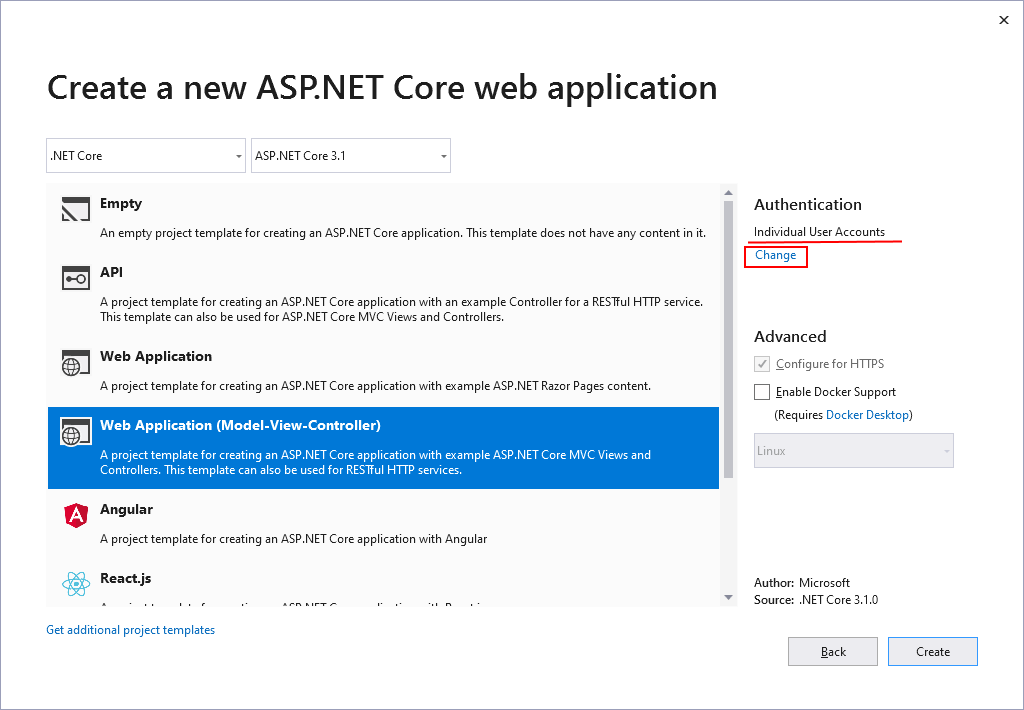
**ASP.NET Core Identity**

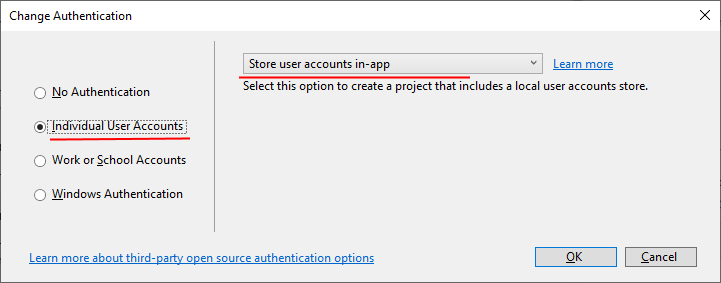
**Введение в ASP.NET Core Identity**

ASP.NET Identity представляет встроенную в ASP.NET систему аутентификации и авторизации. Данная система позволяет пользователям создавать учетные записи, аутентифицироваться, управлять учетными записями или использовать для входа на сайт учетные записи внешних провайдеров, таких как Facebook, Google, Microsoft, Twitter и других.

Так, создадим новый проект ASP.NET Core по типу ASP.NET Core Web App (Model-View-Controller) с типом аутентификации **Individual User Accounts**.:

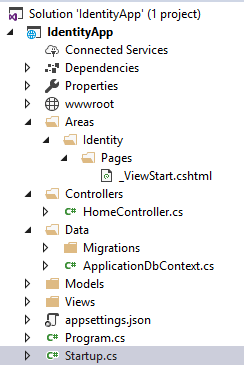


Для выбора типа аутентификации в окне выбора типа проекта в секции **Authentication** нажмем на ссылку **Change**. При нажати на эту ссылку откроется окно установки типа аутентифкации:



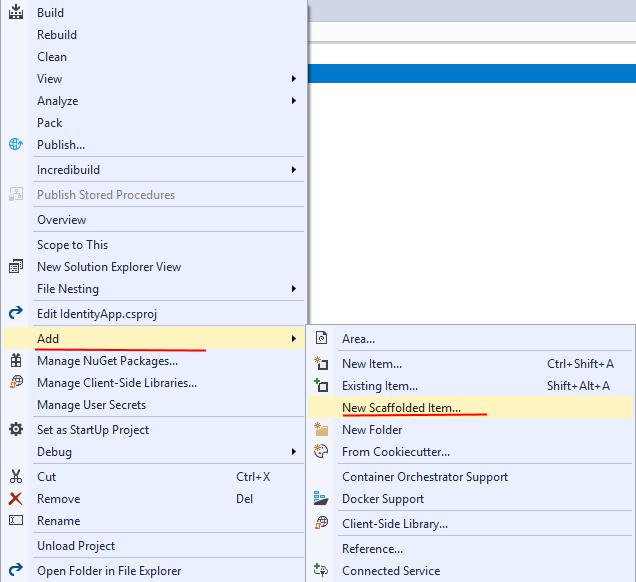
В этом окне соответственно надо указать **Individual User Accounts** и в центральной части **Store user accounts in-app** - сохранение учетных данных пользователей в локальной базе данных.

В итоге будет следующий проект:

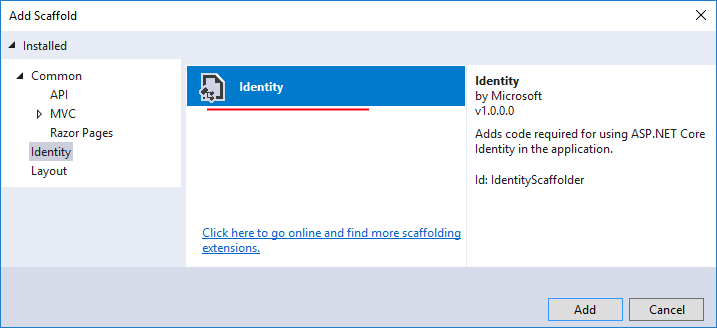


По большому счету сообственно от Identity здесь по умолчанию добавлено не так много функционала. В частности, добавлен класс ApplicationDbContext - контекст данных для работы с Identity. В папке **Areas->Identity->Pages** можно найти файл **\_ViewStart.cshtml**. И есть некоторые изменения в файле Startup.

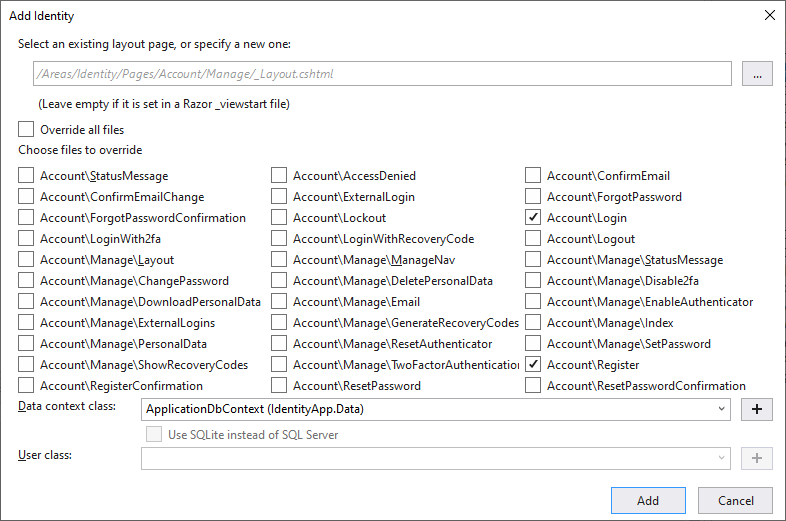
Однако в отличие от предыдущих версий ASP.NET Core в проект по умолчанию не добавляется никаких контроллеров, моделей, представлений, которые обеспечили бы нам полноценную работу с Identity сразу после создания проекта. И весь этот функционал необходимо добвлять самим. Но тем не менее Visual Studio позволяет добавить весь необходимый функционал. Для этого нажмем на название проекта правой кнопкой мыши и в контекстном меню выберем пункт **Add -> New Scaffolded Item**



Далее в открывшемся окне выберем **Idenity**:



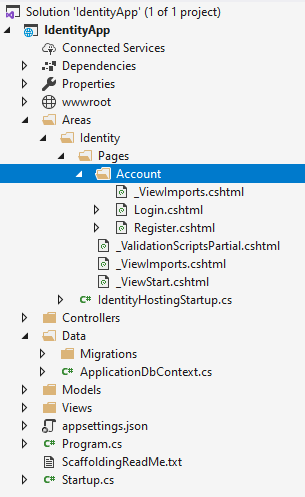
Затем отроется окно, где необходимо указать, какой именно функционал будет добавлен. Здесь очень много различных опций:



В частности, к примеру, я отметил пункты Account/Login и Account/Register, то есть функционал для регистрации и входа в приложение. Кроме того, в поле Data context class надо указать класс контекста данных. По умолчанию при создании в проект уже добавляется класс контекста, который можно указать в этом поле.

Опционально также можно указать путь внутри проекта к мастер-станице и класс пользователя. Но можно также оставить эти поля пустыми, как в моем случае.

После указания всех настроек нажмем на кноку Add, и в проект будет добавлен весь указанный функционал.



Поскольку я отметил пункты Account/Login и Account/Register, то в папке **Areas/Idenity/Pages/Account** можно будет найти Razor-страницы Login и Register, выполняющих соответствующие функции.

Вне зависимости от того, какие функции мы отметили при добавлении Identity, весь функционал по работе с учетными записями помещается в папку Areas/Identity. Начиная с версии ASP.NET Core 2.1 это страницы Razor Pages, которые выполняют какую-то определенную задачу - вход в систему, регистрацию и так далее.

Но чтобы все эти станицы могли выполнять свою роль, в классе Startup добавляется конфигурация Identity. В его методе ConfigureServices() в приложение добавляются сервисы Identity:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public void ConfigureServices(IServiceCollection services)  {      //.......................        // добавление ApplicationDbContext для взаимодействия с базой данных учетных записей      services.AddDbContext<ApplicationDbContext>(options =>          options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));        // добавление сервисов Idenity      services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)                  .AddEntityFrameworkStores<ApplicationDbContext>();        //........................  } |

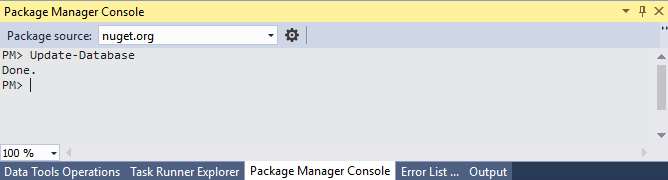
Во-первых, здесь добавляются сервисы Entity Framework, которые используются для работы с базой данных, хранящей учетные записи. Все настройки подключения заданы в файле *appsettings.json* в узле ConnectionStrings.

А в методе Configure() функционал аутентификации и авторизации на основе системы Identity становится доступным через вызовы методов **UseAuthentication** и **app.UseAuthorization()**, как и в любой системе аутентификации и авторизации в ASP.NET Core:

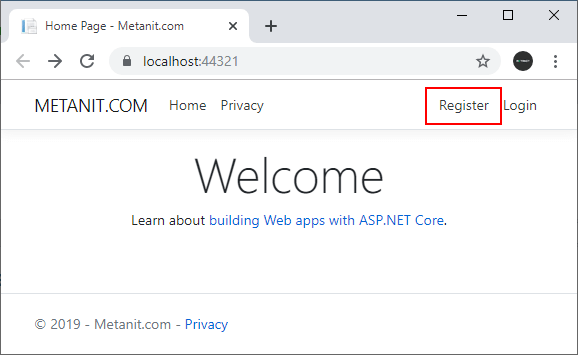
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public void Configure(IApplicationBuilder app, IHostingEnvironment env)  {      //..........................        app.UseAuthentication();      app.UseAuthorization();        //.................................  } |

При начале работы с проектом нам надо учитывать, что он использует Entity Framework, поэтому прежде всего нам надо выполнить миграции, чтобы создать базу данных. В проекте уже по умолчанию есть миграции, которые нам осталось применить. Для этого откроем окно **Package Manager Console** и введем в командную строку следующую команду:

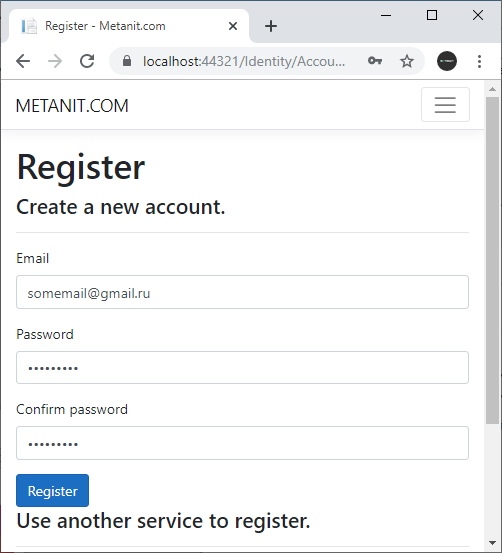
|  |  |
| --- | --- |
| 1 | Update-Database |



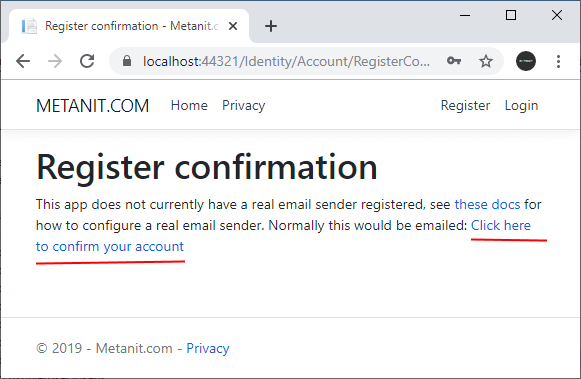
Запустим проект на выполнение. И в меню веб-страницы мы увидим дополнительные пункты для регистрации и входа на сайт:



Нажмем на кнопку регистрации и перед нами откроется форма для регистрации нового пользователя. Введем в нее какие-нибудь данные:

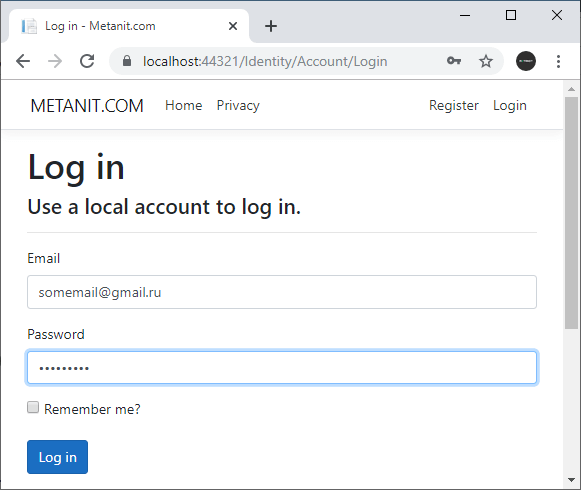
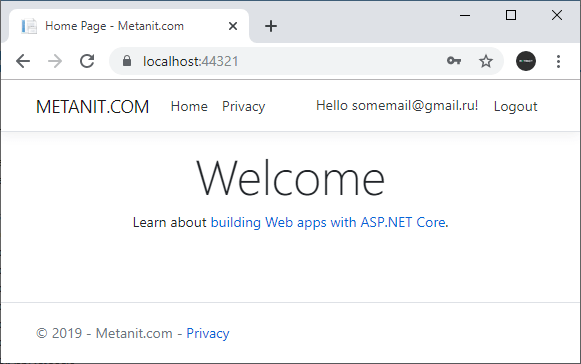


И после регистрации в базу данных будет добавлен новый пользователь, а нас перенаправят на страницу подтверждения регистрации:

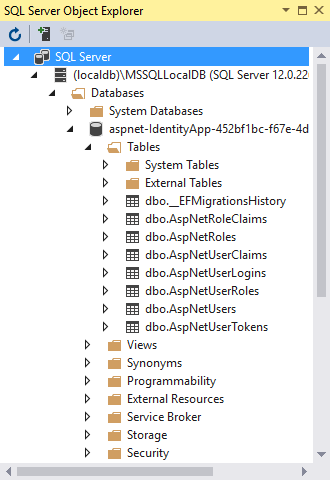


На этой странице нажмем на ссылку "Click here to confirm your account". По умолчанию при создании пользователя в базе данных в таблице пользователей в поле "EmailConfirmed" устанавливается значение False. От этого зависит возможности пользователя. В частности, пока он не подтвердит свой email, он даже не сможет войти в приложение. По умолчанию, в приложении нет конечно никаких возможностей по подтверждению через электонную почту. И нажатие на данную ссылку как раз симулирует подтверждение аккаунта.

Далее с логином и паролем созданного пользователя мы сможем заходить в приложение.

При желании мы можем найти базу данных в окне **SQL Server Object Explorer** по имени, заданному в файле appsettings.json. В частности, в таблице **dbo.AspNetUsers**, которая предназначена для хранения пользователей, мы как раз сможем обнаружить данные добавленного пользователя.



## Основные классы в ASP.NET Core Identity

ASP.NET Core Identity использует множество различных классов, предназначенных для различных задач. Рассмотрим основные из этих классов.

### Контекст данных IdentityDbContext

Для работы с базой данных ASP NET Identity использует контекст данных, который наследуется от класса **IdentityDbContext** из пространства имен Microsoft.AspNetCore.Identity.EntityFrameworkCore:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class ApplicationDbContext : IdentityDbContext<ApplicationUser>  {      public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)              : base(options)      {      }  } |

По умолчанию данный класс наследует весь функционал от IdentityDbContext. Так, мы можем получить содержимое таблиц из бд с помощью следующих свойств:

* **Users**: набор объектов IdentityUser, соответствует таблице пользователей
* **Roles**: набор объектов IdentityRole, соответствует таблице ролей
* **RoleClaims**: набор объектов IdentityRoleClaim, соответствует таблице связи ролей и объектов claims
* **UserLogins**: набор объектов IdentityUserLogin, соответствует таблице связи пользователей с их логинами их внешних сервисов
* **UserClaims**: набор объектов IdentityUserClaim, соответствует таблице связи пользователей и объектов claims
* **UserRoles**: набор объектов IdentityUserRole, соответствует таблице, которая сопоставляет пользователей и их роли
* **UserTokens**: набор объектов IdentityUserToken, соответствует таблице токенов пользователей

По умолчанию таблица, которая соответствует определенному набору, называется по имени набор плюс префикс *AspNet*. Например, таблица пользователей называется *AspNetUsers*.

Если нам надо хранить в базе данных объекты каких-то других классов, то в классе контекста можно определить для них свойство по типу DbSet<T>.

И мы можем напрямую работать с этим контекстом данных. Например, вывести всех пользователей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class AdminController : Controller  {      ApplicationDbContext \_context;      public AdminController(ApplicationDbContext context)      {          \_context = context;      }      public IActionResult Index()      {          var users = \_context.Users.ToList();            return View(users);      }  } |

С помощью стандартного механизма внедрения зависимостей контекст данных внедряется в контроллер, и, получив в конструкторе, мы сможем его использовать.

В представлении выведем список объектов IdentityUser:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @model IEnumerable<Microsoft.AspNetCore.Identity.IdentityUser>  @{      ViewData["Title"] = "Index";  }    <h1>Index</h1>  <ul>      @foreach (var user in Model)      {          <li>@user.Email</li>      }  </ul> |

### IdentityUser

В ASP.NET Core Identity пользователь представлен классом **IdentityUser** из пространства имен Microsoft.AspNetCore.Identity.EntityFrameworkCore. Этот класс предоставляет базовую информацию о пользователе с помощью следующих свойств:

* **Id**: уникальный идентификатор пользователя
* **UserName**: ник пользователя
* **Email**: электронный адрес пользователя
* **Logins**: коллекция логинов, которые использовались пользователем для входа через сторонние сервисы (Google, Facebook и т.д.)
* **Claims**: коллекция клеймов или дополнительных объектов, которые используются для авторизации пользователя
* **PasswordHash**: хеш пароля. В базе данных напрямую не хранится пароль, а только его хеш.
* **Roles**: набор ролей, к которым принадлежит пользователь
* **PhoneNumber**: номер телефона
* **SecurityStamp**: некоторое специальное значение, которое меняется при смене аутентификационных данных, например, пароля
* **AccessFailedCount**: количество неудачных входов пользователя в систему
* **EmailConfirmed**: подтвержден ли адрес электронной почты
* **PhoneNumberConfirmed**: подтвержден ли номер телефона

### Менеджер пользователей UserManager

Как правило, для управления пользователями используется не контекст данных, а специальный класс - **UserManager<T>** из пространства имен Microsoft.AspNetCore.Identity. Основные из его методов и свойств:

* **ChangePasswordAsync(user, old, new)**: изменяет пароль пользователя
* **CreateAsync(user)**: создает нового пользователя
* **DeleteAsync(user)**: удаляет пользователя
* **FindByIdAsync(id)**: ищет пользователя по id
* **FindByEmailAsync(email)**: ищет пользователя по email
* **FindByNameAsync(name)**: ищет пользователя по нику
* **UpdateAsync(user)**: обновляет пользователя
* **Users**: возвращает всех пользователей
* **AddToRoleAsync(user, role)**: добавляет для пользователя user роль role
* **GetRolesAsync (user)**: возвращает список ролей, к которым принадлежит пользователь user
* **IsInRoleAsync(user, name)**: возвращает true, если пользователь user принадлежит роли name
* **RemoveFromRoleAsync(user, name)**: удаляет роль name у пользователя user

После добавления сервисов Identity в классе Startup мы можем получить объект UserManager через механизм внедрения зависимостей, например, через конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | using System.Linq;  using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Identity;    namespace IdentityApp.Controllers  {      public class AdminController : Controller      {          UserManager<IdentityUser> \_userManager;          public AdminController(UserManager<IdentityUser> manager)          {              \_userManager = manager;          }          public IActionResult Index()          {              return View(\_userManager.Users.ToList());          }      }  } |

### IdentityRole и RoleManager

По умолчанию роль в ASP.NET Core Identity представлена классом **IdentityRole**, который определяет три свойства:

* Id: идентификатор роли
* Name: название роли
* Users: коллекция объектов IdentityUserRole, через которые пользователи ассоциированы с данной ролью

Также, как и с пользователями, с ролями мы можем работать напрямую через контекст данных. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | using System.Linq;  using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Identity;  using System.Threading.Tasks;  using Microsoft.EntityFrameworkCore;    namespace IdentityApp.Controllers  {      public class AdminController : Controller      {          ApplicationDbContext \_context;          public AdminController(ApplicationDbContext db)          {              \_context = db;          }          public async Task<IActionResult> GetRoles()          {              \_context.Add(new IdentityRole("simpleUser"));              await \_context.SaveChangesAsync();              return View(await \_context.Roles.ToListAsync());          }      }  } |

И в представлении мы можем вывести все роли:

@model IEnumerable<Microsoft.AspNetCore.Identity.IdentityRole>

@{

ViewData["Title"] = "Index";

}

<h1>Index</h1>

<ul>

@foreach (var role in Model)

{

<li>@role.Name</li>

}

</ul>

Но, как правило, за работу с ролями в ASP.NET Core Identity отвечает класс **RoleManager<T>**, где параметр T представляет класс, описывающий роль. Этот класс представляет ряд методов ля управления ролями:

* **CreateAsync(role)**: создает новую роль
* **DeleteAsync(role)**: удаляет роль
* **FindByIdAsync(id)**: возвращает роль по id
* **FindByNameAsync(name)**: возвращает роль по названию
* **RoleExistsAsync(name)**: возвращает true, если роль с данным именем существует
* **UpdateAsync(role)**: обновляет роль
* **Roles**: возвращает все роли

И также мы можем получить объект RoleManager в контроллерах через механизм внедрения зависимостей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | using System.Linq;  using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Identity;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Identity.EntityFrameworkCore;    namespace IdentityApp.Controllers  {      public class AdminController : Controller      {          RoleManager<IdentityRole> \_roleManager;          public AdminController(RoleManager<IdentityRole> manager)          {              \_roleManager = manager;          }            public async Task<IActionResult> GetRoles()          {              await \_roleManager.CreateAsync(new IdentityRole { Name = "Admin", NormalizedName = "ADMIN" });              return View(await \_roleManager.Roles.ToListAsync());          }      }  } |

Чтобы воспользоваться классом RoleManager, нам надо изменить подключение сервисов Identity в классе Startup в методе ConfigureServices. По умолчанию оно выглядит так:

|  |  |
| --- | --- |
| 1  2 | services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)          .AddEntityFrameworkStores<ApplicationDbContext>(); |

Изменим следующим образом:

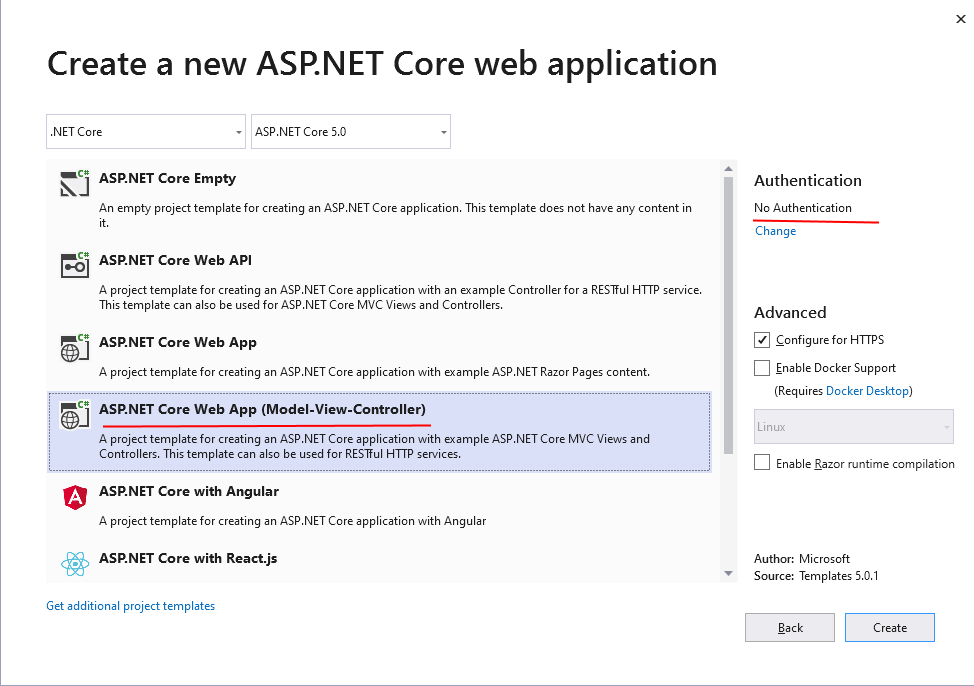
|  |  |
| --- | --- |
| 1  2 | services.AddIdentity<IdentityUser, IdentityRole>(options => options.SignIn.RequireConfirmedAccount = true)          .AddEntityFrameworkStores<ApplicationDbContext>(); |

Метод AddIdentity типизируется классом роли - IdentityRole, тем самым также будет внедряться сервис RoleManager, который мы сможем получить в контроллере.

## Добавление Identity в проект с нуля

Проект с типом аутентификации Individual User Accounts позволяет автоматически добавить все необходимые файлы для работы с ASP.NET Core Identity. Однако, как правило, редко востребован весь стандартный функционал, кроме того, нередко вознкает необходимость более детально настроить различные компоненты, из которых состоит Identity. И в реальности мы можем выбрать любой другой тип проекта и в него уже добавить вручную и подкорректировать функционал ASP.NET Core Identity. Это даст нам больший контроль над тем кодом, который размещается в проекте. Также это может быть актуально если у нас уже есть рабочий проект, но в него необходимо добавить поддержку ASP.NET Core Identity.

Итак, вначале создадим проект ASP.NET Core по шаблону ASP.NET Core Web App (Model-View-Controller) но без аутентификации:

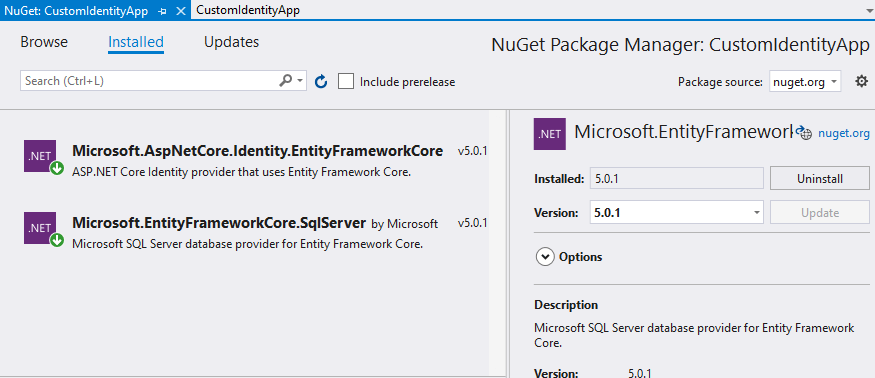


Вначале добавим в папку **Models** новый класс **User**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | using Microsoft.AspNetCore.Identity;    namespace CustomIdentityApp.Models  {      public class User : IdentityUser      {          public int Year { get; set; }      }  } |

Класс User представляет пользователя и наследуется от класса IdentityUser, перенимая все его свойства. Кроме того, для примера здесь добавлено свойство Year, которое будет представлять год рождения пользователя. При желании можно определить любые другие свойства.

Для взаимодействия с MS SQL Server через ASP.NET Core Identity добавим в проект через Nuget пакеты **Microsoft.AspNetCore.Identity.EntityFrameworkCore** и **Microsoft.EntityFrameworkCore.SqlServer**.



Далее добавим в папку Models класс контекста данных **ApplicationContext**. Так как мы используем Identity, то класс контекста данных будет наследоваться не от DbContext, а от IdentityDbContext:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.AspNetCore.Identity.EntityFrameworkCore;  using Microsoft.EntityFrameworkCore;    namespace CustomIdentityApp.Models  {      public class ApplicationContext : IdentityDbContext<User>      {          public ApplicationContext(DbContextOptions<ApplicationContext> options)              : base(options)          {              Database.EnsureCreated();          }      }  } |

У нас есть контекст и модели, и теперь нам необходима база данных, которая будет хранить все данные. Вначале определим в файле *appsettings.json* строку подключения:

|  |  |
| --- | --- |
| 1  2  3 | "ConnectionStrings": {      "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=usersstoredb;Trusted\_Connection=True;"    }, |

Далее нам надо изменить класс Startup, чтобы применить все необходимые сервисы для работы с Identity и базой данных:

|  |  |
| --- | --- |
| 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 | using CustomIdentityApp.Models;  using Microsoft.AspNetCore.Builder;  using Microsoft.AspNetCore.Identity;  using Microsoft.Extensions.Configuration;  using Microsoft.Extensions.DependencyInjection;  using Microsoft.EntityFrameworkCore;    namespace CustomIdentityApp  {      public class Startup      {          public Startup(IConfiguration configuration)          {              Configuration = configuration;          }            public IConfiguration Configuration { get; }            public void ConfigureServices(IServiceCollection services)          {              services.AddDbContext<ApplicationContext>(options =>                  options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));                services.AddIdentity<User, IdentityRole>()                  .AddEntityFrameworkStores<ApplicationContext>();                services.AddControllersWithViews();          }            public void Configure(IApplicationBuilder app)          {              app.UseDeveloperExceptionPage();                app.UseHttpsRedirection();              app.UseStaticFiles();                app.UseRouting();                app.UseAuthentication();    // подключение аутентификации              app.UseAuthorization();                app.UseEndpoints(endpoints =>              {                  endpoints.MapControllerRoute(                      name: "default",                      pattern: "{controller=Home}/{action=Index}/{id?}");              });          }      }  } |

В методе ConfigureServices() добавляются сервисы для Entity Framework Core:

|  |  |
| --- | --- |
| 1  2 | services.AddDbContext<ApplicationContext>(options =>      options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))); |

Далее добавляются специфичные для Identity сервисы:

|  |  |
| --- | --- |
| 1  2 | services.AddIdentity<User, IdentityRole>()      .AddEntityFrameworkStores<ApplicationContext>(); |

Метод AddIdentity() позволяет установить некоторую начальную конфигурацию. Здесь мы указываем тип пользователя и тип роли, которые будут использоваться системой Identity. В качестве типа пользователя выступает созданный нами выше класс User, а в качестве типа роли взят стандартный класс IdentityRole, который находится в пространстве имен Microsoft.AspNetCore.Identity.EntityFrameworkCore.

Метод AddEntityFrameworkStores() устанавливает тип хранилища, которое будет применяться в Identity для хранения данных. В качестве типа хранилища здесь указывается класс контекста данных.

Затем, чтобы использовать Identity, в методе Configure() устанавливается компонент middeware - UseAuthentication. Причем это middleware вызывается перед app.UseEndpoints(), тем самым гарантируя, что ко времени обращения к системе маршрутизации, контроллерам и их методам, куки должным образом обработаны и установлены.

Теперь система Identity подключена в проект, и мы можем с ней работать.

## Регистрация и создание пользователей в Identity

Продолжим работу с проектом из прошлой темы и добавим в него функционал регистрации пользователей.

Пользователи в приложении представлены классом User, который наследует множество свойств. Однако нам необязательно всех их устанавливать. Достаточно установить ключевые свойства вроде логина и пароля. Для этого лучше воспользоваться вспомогательной моделью, которая установит все необходимые свойства.

Итак, добавим в проект новую папку, которую назовем **ViewModels**. Затем в этой папке определим новый класс **RegisterViewModel**, который будет представлять регистрирующегося пользователя:

|  |  |
| --- | --- |
| 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 | using System.ComponentModel.DataAnnotations;    namespace CustomIdentityApp.ViewModels  {      public class RegisterViewModel      {          [Required]          [Display(Name = "Email")]          public string Email { get; set; }            [Required]          [Display(Name = "Год рождения")]          public int Year { get; set; }            [Required]          [DataType(DataType.Password)]          [Display(Name = "Пароль")]          public string Password { get; set; }            [Required]          [Compare("Password", ErrorMessage = "Пароли не совпадают")]          [DataType(DataType.Password)]          [Display(Name = "Подтвердить пароль")]          public string PasswordConfirm { get; set; }      }  } |

Для работы с учетными записями пользователей добавим в папку **Controllers** новый контроллер **AccountController** и определим в нем метод для регистрации пользователей:

|  |  |
| --- | --- |
| 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 | using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using CustomIdentityApp.ViewModels;  using CustomIdentityApp.Models;  using Microsoft.AspNetCore.Identity;    namespace CustomIdentityApp.Controllers  {      public class AccountController : Controller      {          private readonly UserManager<User> \_userManager;          private readonly SignInManager<User> \_signInManager;            public AccountController(UserManager<User> userManager, SignInManager<User> signInManager)          {              \_userManager = userManager;              \_signInManager = signInManager;          }          [HttpGet]          public IActionResult Register()          {              return View();          }          [HttpPost]          public async Task<IActionResult> Register(RegisterViewModel model)          {              if(ModelState.IsValid)              {                  User user = new User { Email = model.Email, UserName = model.Email, Year=model.Year};                  // добавляем пользователя                  var result = await \_userManager.CreateAsync(user, model.Password);                  if (result.Succeeded)                  {                      // установка куки                      await \_signInManager.SignInAsync(user, false);                      return RedirectToAction("Index", "Home");                  }                  else                  {                      foreach (var error in result.Errors)                      {                          ModelState.AddModelError(string.Empty, error.Description);                      }                  }              }              return View(model);          }      }  } |

Поскольку в классе Startup были добавлены сервисы Identity, то здесь в контроллере через конструктор мы можем их получить. В данном случае мы получаем сервис по управлению пользователями - UserManager и сервис SignInManager, который позволяет аутентифицировать пользователя и устанавливать или удалять его куки.

С помощью метода \_userManager.CreateAsync пользователь добавляется в базу данных. В качестве параметра передается сам пользователь и его пароль.

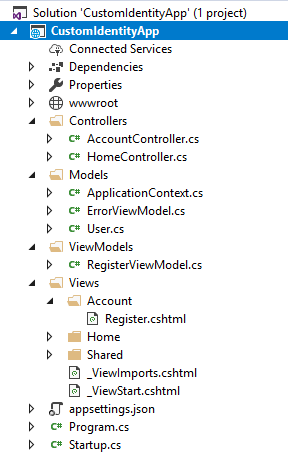
Данный метод возвращает объект IdentityResult, с помощью которого можно узнать успешность выполненной операции. Вполне возможно, что переданные значения не удовлетворяют требованиям, и тогда пользователь не будет добавлен в базу данных. В случае удачного добавления с помощью метода \_signInManager.SignInAsync() устанавливаем аутентификационные куки для добавленного пользователя. В этот метод передается объект пользователя, который аутентифицируется, и логическое значение, указывающее, надо ли сохранять куки в течение продолжительного времени. И далее выполняем переадресацию на главную страницу приложения.

Если добавление прошло неудачно, то добавляем к состоянию модели с помощью метода ModelState все возникшие при добавлении ошибки, и отправленная модель возвращается в представление.

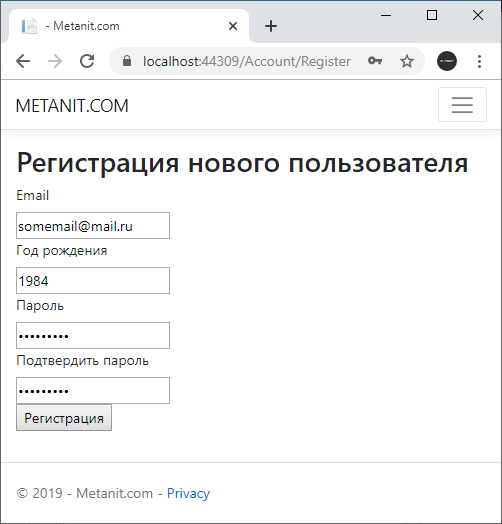
Для представлений этого контроллера в каталоге Views определим подкаталог **Account**, в который добавим новое представление **Register.cshtml**. Это представление будет служить для регистрации пользователя:

|  |  |
| --- | --- |
| 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 | @model CustomIdentityApp.ViewModels.RegisterViewModel  <h2>Регистрация нового пользователя</h2>  <form method="post" asp-controller="Account" asp-action="Register">      <div asp-validation-summary="ModelOnly"></div>      <div>          <label asp-for="Email"></label><br />          <input asp-for="Email" />          <span asp-validation-for="Email"></span>      </div>      <div>          <label asp-for="Year"></label><br />          <input asp-for="Year" />          <span asp-validation-for="Year"></span>      </div>      <div>          <label asp-for="Password"></label><br />          <input asp-for="Password" />          <span asp-validation-for="Password"></span>      </div>      <div>          <label asp-for="PasswordConfirm"></label><br />          <input asp-for="PasswordConfirm" />          <span asp-validation-for="PasswordConfirm"></span>      </div>      <div>          <input type="submit" value="Регистрация" />      </div>  </form> |

В итоге вся структура проекта на данный момент будет выглядеть следующим образом:

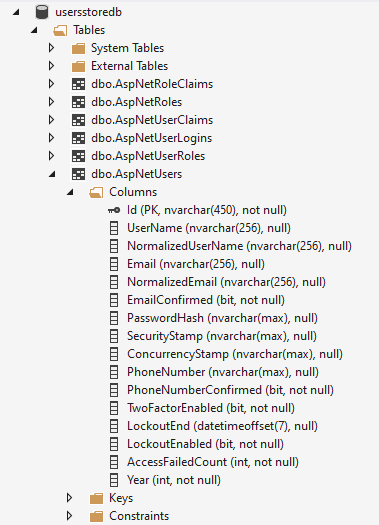


Запустим приложение и обратимся к действию регистрации. Введем какие-нибудь данные:

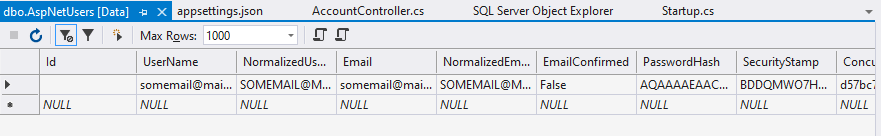


После удачной регистрации нас переадресует на главную страницу.

После добавленяи первого пользователя через окно **SQL Server Objects Explorer** мы можем увидеть созданную базу данных:



А в самой базе данных в таблице **dbo.AspNetUsers** мы сможем увидеть добавленного пользователя.



## Авторизация пользователей в Identity

Для создания механизма авторизации пользователей в приложении вначале добавим в проект из прошлой темы в папку ViewModels специальную модель **LoginViewModel**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | using System.ComponentModel.DataAnnotations;    namespace CustomIdentityApp.ViewModels  {      public class LoginViewModel      {          [Required]          [Display(Name = "Email")]          public string Email { get; set; }            [Required]          [DataType(DataType.Password)]          [Display(Name = "Пароль")]          public string Password { get; set; }            [Display(Name = "Запомнить?")]          public bool RememberMe { get; set; }            public string ReturnUrl { get; set; }      }  } |

В прошлой теме у нас уже был добавлен контроллер AccountController. Теперь же добавим в него следующие три метода:

|  |  |
| --- | --- |
| 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 | [HttpGet]  public IActionResult Login(string returnUrl = null)  {      return View(new LoginViewModel { ReturnUrl = returnUrl });  }    [HttpPost]  [ValidateAntiForgeryToken]  public async Task<IActionResult> Login(LoginViewModel model)  {      if (ModelState.IsValid)      {          var result =              await \_signInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, false);          if (result.Succeeded)          {              // проверяем, принадлежит ли URL приложению              if (!string.IsNullOrEmpty(model.ReturnUrl) && Url.IsLocalUrl(model.ReturnUrl))              {                  return Redirect(model.ReturnUrl);              }              else              {                  return RedirectToAction("Index", "Home");              }          }          else          {              ModelState.AddModelError("", "Неправильный логин и (или) пароль");          }      }      return View(model);  }    [HttpPost]  [ValidateAntiForgeryToken]  public async Task<IActionResult> Logout()  {      // удаляем аутентификационные куки      await \_signInManager.SignOutAsync();      return RedirectToAction("Index", "Home");  } |

В Get-версии метода Login мы получаем адрес для возврата в виде параметра returnUrl и передаем его в модель LoginViewModel.

В Post-версии метода Login получаем данные из представления в виде модели LoginViewModel. Всю работу по аутентификации пользователя выполняет метод signInManager.PasswordSignInAsync(). Этот метод принимает логин и пароль пользователя. Третий параметр метода указывает, надо ли сохранять устанавливаемые куки на долгое время.

Данный метод также возвращает IdentityResult, с помощью которого можно узнать, завершилась ли аутентификация успешно. Если она завершилось успешно, то используем свойство ReturnUrl модели LoginViewModel для возврата пользователя на предыдущее место. Для этого нужно еще удостовериться, что адрес возврата принадлежит приложению с помощью метода Url.IsLocalUrl(). Это позволит избежать перенаправлений на нежелательные сайты. Если же адрес возврата не установлен или не принадлежит приложению, выполняем переадресацию на главную страницу.

Третий метод - метод Logout выполняет выход пользователя из приложения. За выход отвечает метод \_signInManager.SignOutAsync(), который очищает аутентификационные куки.

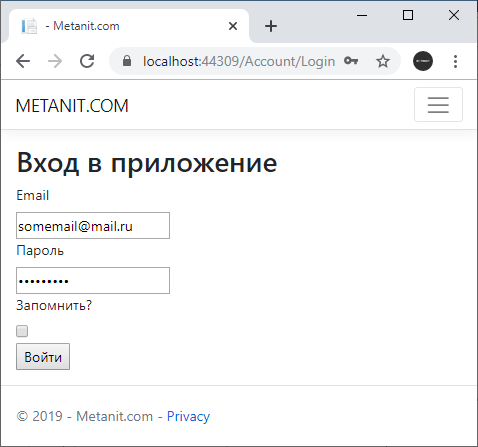
И теперь определим в проекте в папке *Views/Account* представление **Login.cshtml**, через которое будет осуществляться вход в приложение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | @model CustomIdentityApp.ViewModels.LoginViewModel    <h2>Вход в приложение</h2>  <form method="post" asp-controller="Account" asp-action="Login"        asp-route-returnUrl="@Model.ReturnUrl">      <div asp-validation-summary="ModelOnly"></div>      <div>          <label asp-for="Email"></label><br />          <input asp-for="Email" />          <span asp-validation-for="Email"></span>      </div>      <div>          <label asp-for="Password"></label><br />          <input asp-for="Password" />          <span asp-validation-for="Password"></span>      </div>      <div>          <label asp-for="RememberMe"></label><br />          <input asp-for="RememberMe" />      </div>      <div>          <input type="submit" value="Войти" />      </div>  </form> |

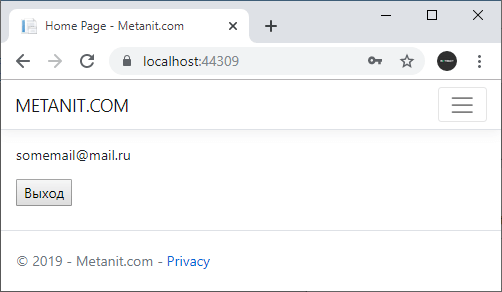
Для проверки логина для простоты определим в представлении Index.cshtml контроллера HomeController вывод имени пользователя и ссылки на вход и регистрацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @{      ViewData["Title"] = "Home Page";  }      @if(User.Identity.IsAuthenticated)  {      <p>@User.Identity.Name</p>        <form method="post" asp-controller="Account" asp-action="Logout">          <input type="submit" value="Выход" />      </form>  }  else  {      <a asp-controller="Account" asp-action="Login">Вход</a>      <a asp-controller="Account" asp-action="Register">Регистрация</a>  } |

Запустим проект и выполним вход в приложение:



И после удачной авторизации на главной странице увидим свое имя и кнопку для выхода:



## Управление пользователями

Для администрирования пользователей продолжим работу с проектом из прошлой темы и добавим в папку Controllers новый контроллер **UsersController**:

|  |  |
| --- | --- |
| 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 | using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Identity;  using CustomIdentityApp.Models;  using CustomIdentityApp.ViewModels;    namespace CustomIdentityApp.Controllers  {      public class UsersController : Controller      {          UserManager<User> \_userManager;            public UsersController(UserManager<User> userManager)          {              \_userManager = userManager;          }            public IActionResult Index() => View(\_userManager.Users.ToList());            public IActionResult Create() => View();            [HttpPost]          public async Task<IActionResult> Create(CreateUserViewModel model)          {              if (ModelState.IsValid)              {                  User user = new User { Email = model.Email, UserName = model.Email, Year = model.Year };                  var result = await \_userManager.CreateAsync(user, model.Password);                  if (result.Succeeded)                  {                      return RedirectToAction("Index");                  }                  else                  {                      foreach (var error in result.Errors)                      {                          ModelState.AddModelError(string.Empty, error.Description);                      }                  }              }              return View(model);          }            public async Task<IActionResult> Edit(string id)          {              User user = await \_userManager.FindByIdAsync(id);              if (user == null)              {                  return NotFound();              }              EditUserViewModel model = new EditUserViewModel {Id = user.Id, Email = user.Email, Year = user.Year };              return View(model);          }            [HttpPost]          public async Task<IActionResult> Edit(EditUserViewModel model)          {              if (ModelState.IsValid)              {                  User user = await \_userManager.FindByIdAsync(model.Id);                  if(user!=null)                  {                      user.Email = model.Email;                      user.UserName = model.Email;                      user.Year = model.Year;                        var result = await \_userManager.UpdateAsync(user);                      if (result.Succeeded)                      {                          return RedirectToAction("Index");                      }                      else                      {                          foreach (var error in result.Errors)                          {                              ModelState.AddModelError(string.Empty, error.Description);                          }                      }                  }              }              return View(model);          }            [HttpPost]          public async Task<ActionResult> Delete(string id)          {              User user = await \_userManager.FindByIdAsync(id);              if (user != null)              {                  IdentityResult result = await \_userManager.DeleteAsync(user);              }              return RedirectToAction("Index");          }      }  } |

Вся работа с пользователями производится с помощью ранее рассмотренных методов класса UserManager.

Для методов Create и Edit в папке ViewModels создадим дополнительные модели представлений:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class CreateUserViewModel  {      public string Email { get; set; }      public string Password { get; set; }      public int Year { get; set; }  }  public class EditUserViewModel  {      public string Id { get; set; }      public string Email { get; set; }      public int Year { get; set; }  } |

А в папку Views добавим подкаталог *Users* для представлений контроллера. Далее определим в ней представление *Index.cshtml* для вывода пользователей:

|  |  |
| --- | --- |
| 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 | @model IEnumerable<CustomIdentityApp.Models.User>  @{      ViewBag.Title = "Список пользователей";  }    <a asp-action="Create">Добавить пользователя</a>    <table class="table">      <tr><th>Email</th><th>Год</th><th></th></tr>      @foreach (var user in Model)      {          <tr>              <td>@user.Email</td>              <td>@user.Year</td>              <td>                  <form asp-action="Delete" asp-route-id="@user.Id" method="post">                      <a class="btn btn-sm btn-primary" asp-action="Edit" asp-route-id="@user.Id">Изменить</a>                      <button type="submit" class="btn btn-sm btn-danger">                          Удалить                      </button>                  </form>              </td>          </tr>      }  </table> |

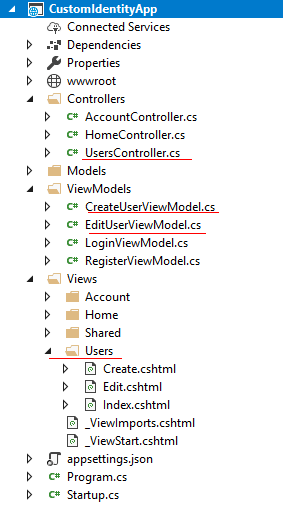
Также добавим представление *Create.cshtml* для создания новых пользователей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | @model CustomIdentityApp.ViewModels.CreateUserViewModel  @{      ViewBag.Title = "Добавление пользователя";  }  <form asp-action="Create" asp-controller="Users">      <div asp-validation-summary="All" class="text-danger"></div>      <div class="form-group">          <label asp-for="Email" class="control-label">Email</label>          <input type="text" asp-for="Email" class="form-control" />      </div>      <div class="form-group">          <label asp-for="Password" class="control-label">Пароль</label>          <input type="password" asp-for="Password" class="form-control" />      </div>      <div class="form-group">          <label asp-for="Year" class="control-label">Год рождения</label>          <input type="number" asp-for="Year" class="form-control" />      </div>      <div class="form-group">          <input type="submit" value="Добавить" class="btn btn-outline-secondary" />      </div>  </form> |

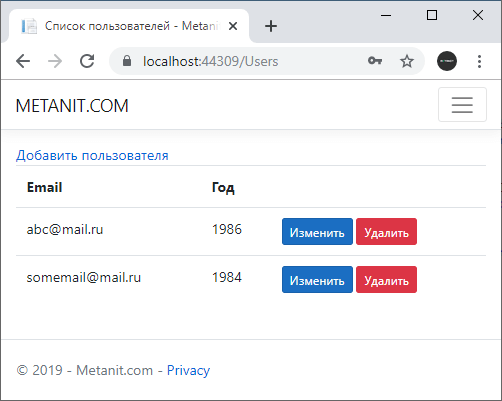
И также добавим представление *Edit.cshtml* для редактирования пользователей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | @model CustomIdentityApp.ViewModels.EditUserViewModel  @{      ViewBag.Title = "Редактирование пользователя";  }  <form asp-action="Edit" asp-controller="Users">      <div asp-validation-summary="All" class="text-danger"></div>      <div class="form-group">          <input type="hidden" asp-for="Id" />      </div>      <div class="form-group">          <label asp-for="Email" class="control-label">Email</label>          <input type="text" asp-for="Email" class="form-control" />      </div>      <div class="form-group">          <label asp-for="Year" class="control-label">Год рождения</label>          <input type="number" asp-for="Year" class="form-control" />      </div>      <div class="form-group">          <input type="submit" value="Сохранить" class="btn btn-outline-secondary" />      </div>  </form> |

В итоге проект изменится следующим образом:



В теперь при обращении к методу Index контроллера Users мы сможем увидеть список пользователей, отредактировать, удалить или добавить новых пользователей:



## Изменение пароля

В прошлой теме было рассмотрено редактирование пользователя. Однако процесс редактирования не включал изменение пароля. В этой же теме рассмотрим, как отдельно сменить пароль пользователя.

Вначале создадим модель представления ChangePasswordViewModel:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class ChangePasswordViewModel  {      public string Id { get; set; }      public string Email { get; set; }      public string NewPassword { get; set; }  } |

Здесь определены свойства для Id и электронной почты пользователя, а также для нового пароля.

Далее добавим в контроллер UsersController, который был создан в прошлой теме, новое действие ChangePassword:

|  |  |
| --- | --- |
| 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 | public async Task<IActionResult> ChangePassword(string id)  {      User user = await \_userManager.FindByIdAsync(id);      if (user == null)      {          return NotFound();      }      ChangePasswordViewModel model = new ChangePasswordViewModel { Id = user.Id, Email = user.Email};      return View(model);  }    [HttpPost]  public async Task<IActionResult> ChangePassword(ChangePasswordViewModel model)  {      if (ModelState.IsValid)      {          User user = await \_userManager.FindByIdAsync(model.Id);          if (user != null)          {              var \_passwordValidator =                  HttpContext.RequestServices.GetService(typeof(IPasswordValidator<User>)) as IPasswordValidator<User>;              var \_passwordHasher =                  HttpContext.RequestServices.GetService(typeof(IPasswordHasher<User>)) as IPasswordHasher<User>;                IdentityResult result =                  await \_passwordValidator.ValidateAsync(\_userManager, user, model.NewPassword);              if(result.Succeeded)              {                  user.PasswordHash = \_passwordHasher.HashPassword(user, model.NewPassword);                  await \_userManager.UpdateAsync(user);                  return RedirectToAction("Index");              }              else              {                  foreach (var error in result.Errors)                  {                      ModelState.AddModelError(string.Empty, error.Description);                  }              }          }          else          {              ModelState.AddModelError(string.Empty, "Пользователь не найден");          }      }      return View(model);  } |

Таблица пользователей не хранит оригинальный пароль, а вместо него хранит хеш пароля. Для его получения мы можем воспользоваться методом HashPassword() сервиса **IPasswordHasher**, который добавляется вместе со всеми сервисами ASP.NET Core Identity через механизм внедрения зависимостей.

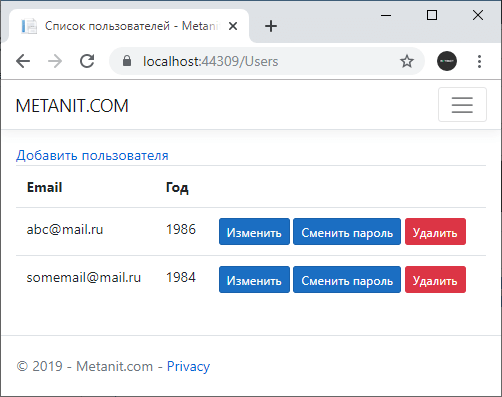
Однако перед созданием хеша нам надо валидировать пароль с помощью валидатора. Валидатор пароля также можно получить из коллекции сервисов приложения в виде объекта **IPasswordValidator**.

Затем добавим представление *ChangePassword.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | @model CustomIdentityApp.ViewModels.ChangePasswordViewModel  @{      ViewBag.Title = "Изменение пароля";  }  <h2>Изменение пароля для пользователя @Model.Email</h2>  <form asp-action="ChangePassword" asp-controller="Users">      <div asp-validation-summary="All" class="text-danger"></div>        <input type="hidden" asp-for="Id" />      <input type="hidden" asp-for="Email" />        <div class="form-group">          <label asp-for="NewPassword" class="control-label">Новый пароль</label>          <input type="text" asp-for="NewPassword" class="form-control" />      </div>      <div class="form-group">          <input type="submit" value="Сохранить" class="btn btn-outline-dark" />      </div>  </form> |

И в конце добавим на представление *Index.cshtml* ссылку для смены пароля:

|  |  |
| --- | --- |
| 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 | @model IEnumerable<CustomIdentityApp.Models.User>  @{      ViewBag.Title = "Список пользователей";  }    <a asp-action="Create">Добавить пользователя</a>    <table class="table">      <tr><th>Email</th><th>Год</th><th></th></tr>      @foreach (var user in Model)      {          <tr>              <td>@user.Email</td>              <td>@user.Year</td>              <td>                  <form asp-action="Delete" asp-route-id="@user.Id" method="post">                      <a class="btn btn-sm btn-primary" asp-action="Edit" asp-route-id="@user.Id">Изменить</a>                      <a class="btn btn-sm btn-primary" asp-action="ChangePassword" asp-route-id="@user.Id">Сменить пароль</a>                      <button type="submit" class="btn btn-sm btn-danger">                          Удалить                      </button>                  </form>              </td>          </tr>      }  </table> |



### Второй вариант

В предыдущем случае мы явным образом создавали хеш пароля и обновляли пользователя. Однако мы можем воспользоваться и другим методом - методом ChangePasswordAsync() класса UserManager. Он в качестве параметра принимает объект пользователя, старый и новый пароль. То есть, чтобы сменить пароль на новый, нам надо указать также и старый, что может быть целесообразно в различных ситуациях.

Для этого изменим модель ChangePasswordViewModel:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class ChangePasswordViewModel  {      public string Id { get; set; }      public string Email { get; set; }      public string NewPassword { get; set; }      public string OldPassword { get; set; }  } |

Изменим post-версию метода ChangePassword:

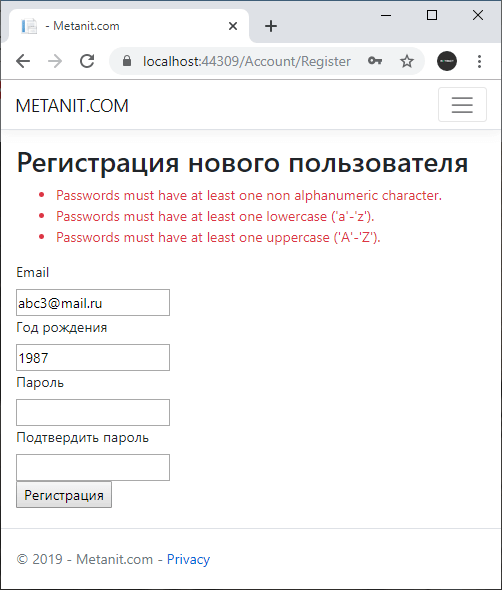
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | [HttpPost]  public async Task<IActionResult> ChangePassword(ChangePasswordViewModel model)  {      if (ModelState.IsValid)      {          User user = await \_userManager.FindByIdAsync(model.Id);          if (user != null)          {              IdentityResult result =                  await \_userManager.ChangePasswordAsync(user, model.OldPassword, model.NewPassword);              if(result.Succeeded)              {                  return RedirectToAction("Index");              }              else              {                  foreach (var error in result.Errors)                  {                      ModelState.AddModelError(string.Empty, error.Description);                  }              }          }          else          {              ModelState.AddModelError(string.Empty, "Пользователь не найден");          }      }      return View(model);  } |

И соответственно изменим представление *ChangePassword.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | @model CustomIdentityApp.ViewModels.ChangePasswordViewModel  @{      ViewBag.Title = "Изменение пароля";  }  <h2>Изменение пароля для пользователя @Model.Email</h2>  <form asp-action="ChangePassword" asp-controller="Users">      <div asp-validation-summary="All" class="text-danger"></div>        <input type="hidden" asp-for="Id" />      <input type="hidden" asp-for="Email" />      <div class="form-group">          <label asp-for="OldPassword" class="control-label">Старый пароль</label>          <input type="text" asp-for="OldPassword" class="form-control" />      </div>      <div class="form-group">          <label asp-for="NewPassword" class="control-label">Новый пароль</label>          <input type="text" asp-for="NewPassword" class="form-control" />      </div>      <div class="form-group">          <input type="submit" value="Сохранить" class="btn btn-default" />      </div>  </form> |

## Валидация пароля

Система ASP.NET Identity уже имеет ряд ограничений на вводимые данные, которые нам надо использовать. Например, если мы возьмем любой проект с ASP.NET Core Identity, например, созданный в прошлых темах, то мы заметим, что для валидации пароля в Identity уже заложен ряд ограничений:



При вводе пароля срабатывает встроенная логика, согласно которой длина пароля не может быть меньше 6 символов, пароль должен содержать как минимум один цифровой и один не алфавитно-цифровой символ, а также как минимум один алфавитный символ должен быть в верхнем регистре. Но что, если нам надо установить другую минимальную длину пароля? Или что если мы хотим, чтобы в пароле могли бы использоваться только цифровые или только алфавитные символы?

И несмотря на то, что встроенная логика валидации скрыта от наших глаз, мы ее можем переопределить с помощью объекта **Microsoft.AspNetCore.Identity.PasswordOptions**.

Чтобы его применить, необходимо изменить код добавления контекста в классе Startup в методе ConfigureServices(), который по умолчанию имеет следующее содержимое:

|  |  |
| --- | --- |
| 1  2 | services.AddIdentity<User, IdentityRole>()          .AddEntityFrameworkStores<ApplicationContext>(); |

Итак, изменим данный код следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | services.AddIdentity<User, IdentityRole>(opts=> {          opts.Password.RequiredLength = 5;   // минимальная длина          opts.Password.RequireNonAlphanumeric = false;   // требуются ли не алфавитно-цифровые символы          opts.Password.RequireLowercase = false; // требуются ли символы в нижнем регистре          opts.Password.RequireUppercase = false; // требуются ли символы в верхнем регистре          opts.Password.RequireDigit = false; // требуются ли цифры      })      .AddEntityFrameworkStores<ApplicationDbContext>(); |

Класс **PasswordOptions** предоставляет следующие свойства:

* RequiredLength: минимальная длина пароля
* RequireNonLetterOrDigit: если равно true, то пароль должен будет иметь как минимум один символ, который не является алфавитно-цифровым
* RequireDigit: если равно true, то пароль должен будет иметь как минимум одну цифру
* RequireLowercase: если равно true, то пароль должен будет иметь как минимум один символ в нижнем регистре
* RequireUppercase: если равно true, то пароль должен будет иметь как минимум один символ в верхнем регистре

Если мы меняем длину пароля, то соответственно нам надо изменить определение модели регистрации, если она устанавливает ограничение на длину:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class RegisterViewModel  {      [Required]      [Display(Name = "Email")]      public string Email { get; set; }        [Required]      [Display(Name = "Год рождения")]      public int Year { get; set; }        [Required]      [DataType(DataType.Password)]      [StringLength(100, ErrorMessage = "Поле {0} должно иметь минимум {2} и максимум {1} символов.", MinimumLength = 5)]      [Display(Name = "Пароль")]      public string Password { get; set; }        [Required]      [Compare("Password", ErrorMessage = "Пароли не совпадают")]      [DataType(DataType.Password)]      [Display(Name = "Подтвердить пароль")]      public string PasswordConfirm { get; set; }  } |

Для большинства случаев настроек класса PasswordOptions вполне хватит, мы можем настроить минимальную длину пароля, ввод алфавитных, цифровых или не алфавитно-цифровых символов. Однако если нам потребуется более сложная логика валидации, то придется определять свой класс валидатора.

Объект валидатора представляет объект интерфейса IPasswordValidator<T>:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IPasswordValidator<T> where T : class {        Task<IdentityResult> ValidateAsync(UserManager<T> manager, T user, string password);  } |

Метод ValidateAsync() вызывается при валидации пароля. В качестве параметров он получает объект UserManager, объект пользователя и сам пароль, который надо валидировать.

Результат метода представляет объект IdentityResult, через статус которого можно узнать об успешности валидации: если свойство Succeeded равно true, то валидация прошла успешно.

Итак, создадим новый класс, который назовем и который будет иметь следующий код:

|  |  |
| --- | --- |
| 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 | using System;  using System.Collections.Generic;  using System.Text.RegularExpressions;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Identity;    namespace CustomIdentityApp.Models  {      public class CustomPasswordValidator : IPasswordValidator<User>      {          public int RequiredLength { get; set; } // минимальная длина            public CustomPasswordValidator(int length)          {              RequiredLength = length;          }            public Task<IdentityResult> ValidateAsync(UserManager<User> manager, User user, string password)          {              List<IdentityError> errors = new List<IdentityError>();                if (String.IsNullOrEmpty(password) || password.Length < RequiredLength)              {                  errors.Add(new IdentityError                  {                      Description = $"Минимальная длина пароля равна {RequiredLength}"                  });              }              string pattern = "^[0-9]+$";                if (!Regex.IsMatch(password, pattern))              {                  errors.Add(new IdentityError                  {                      Description = "Пароль должен состоять только из цифр"                  });              }              return Task.FromResult(errors.Count == 0 ?                  IdentityResult.Success : IdentityResult.Failed(errors.ToArray()));          }      }  } |

Для проверки на длину в классе определено свойство RequiredLength. Кроме проверки количества символов пароль здесь также проверяется на соответствие регулярному выражению, которое говорит о том, что все символы должны представлять цифры (string pattern = "^[0-9]+$"). Таким образом, мы можем определить в методе различные способы проверок.

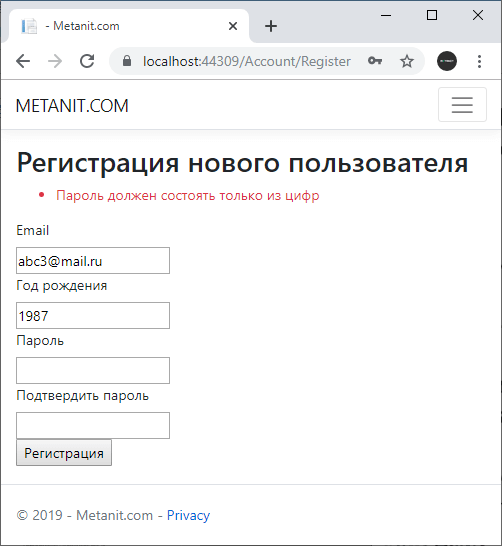
Если какая-то из проверок не будет пройдена, то в список ошибок добавляется очередная ошибка в виде объекта IdentityError. При отстутствии ошибок возвращается результат IdentityResult.Success.

Для использования класса перейдем в класс Startup и изменим его метод ConfigureServices():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public void ConfigureServices(IServiceCollection services)  {      // Добавляем сервис валидатора пароля      services.AddTransient<IPasswordValidator<User>,              CustomPasswordValidator>(serv => new CustomPasswordValidator(6));        services.AddDbContext<ApplicationContext>(options =>          options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));        services.AddIdentity<User, IdentityRole>()              .AddEntityFrameworkStores<ApplicationContext>();        services.AddControllersWithViews();  } |

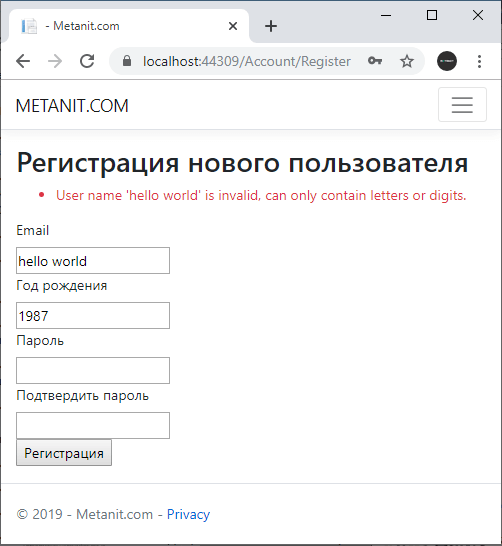
Первый вызов добавляет зависимость для интерфейса IPasswordValidator в виде объекта CustomPasswordValidator.

И теперь, если мы введем некорректные с точки зрения нашего валидатора значения, то получим соответствующие сообщения:



## Валидация пользователя

Кроме проверки пароля в ASP.NET Core Identity также валидируется и пользователь. В частности, если мы введем при регистрации некорректные данные для поля Email, то получим ошибки валидации пользователя:



В данном случае ошибка говорит о том, что email некорректен и должен содержать только алфавитно-цифровые символы.

За валидацию пользователя по умолчанию отвечает класс **UserOptions**, который определяет следующие свойства:

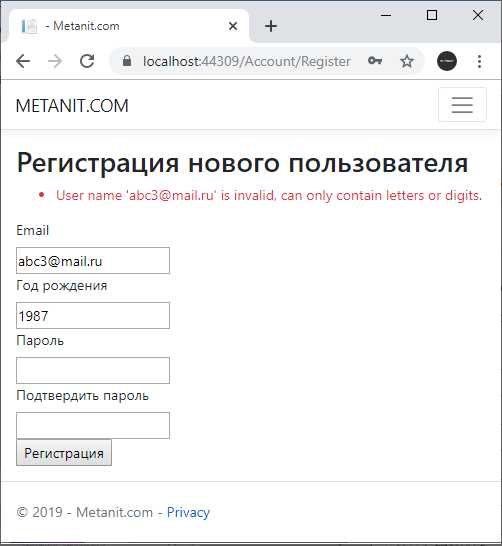
* AllowedUserNameCharacters: если равно true, то юзернейм должен содержать только алфавитно-цифровые символы
* RequireUniqueEmail: если равно true, то email пользователя должен быть уникальным

Для применения этих свойств изменим установку контекста данных в методе ConfigureServices() в классе Startup:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public void ConfigureServices(IServiceCollection services)  {      services.AddDbContext<ApplicationDbContext>(options =>          options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));        services.AddIdentity<User, IdentityRole>(opts => {              opts.User.RequireUniqueEmail = true;    // уникальный email              opts.User.AllowedUserNameCharacters = ".@abcdefghijklmnopqrstuvwxyz"; // допустимые символы          })          .AddEntityFrameworkStores<ApplicationDbContext>();        services.AddControllersWithViews();  } |

Свойство opts.User как раз и представляет класс UserOptions.

Этот способ работает, но все же он имеет недостатки:



Несмотря на то, что разрешены только алфавитные символы, в сообщении все равно говорится, что email должен содержать алфавитные или цифровые символы. И во-вторых, сообщения отображаются на английском языке. И в этом случае мы можем полностью переопределить логику валидации, создав свой класс валидатора.

Класс валидатора должен реализовать интерфейс IUserValidator<T>:

|  |  |
| --- | --- |
| 1  2  3 | public interface IUserValidator<TUser> where TUser : class {      Task<IdentityResult> ValidateAsync(UserManager<TUser> manager, TUser user);  } |

Метод ValidateAsync() вызывается при валидации адреса электронной почты. В качестве параметров в метод передаются объект UserManager и валидируемый пользователь.

Теперь создадим класс-валидатор CustomUserValidator:

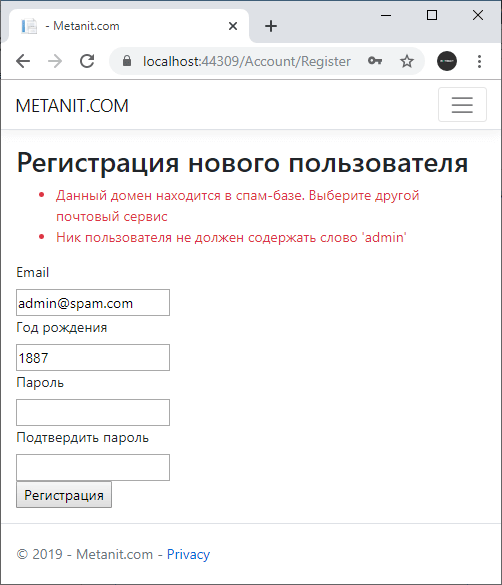
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | using System.Collections.Generic;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Identity;    namespace CustomIdentityApp.Models  {      public class CustomUserValidator : IUserValidator<User>      {          public Task<IdentityResult> ValidateAsync(UserManager<User> manager, User user)          {              List<IdentityError> errors = new List<IdentityError>();                if (user.Email.ToLower().EndsWith("@spam.com"))              {                  errors.Add(new IdentityError                  {                      Description = "Данный домен находится в спам-базе. Выберите другой почтовый сервис"                  });              }              if (user.UserName.Contains("admin"))              {                  errors.Add(new IdentityError                  {                      Description = "Ник пользователя не должен содержать слово 'admin'"                  });              }              return Task.FromResult(errors.Count == 0 ?                  IdentityResult.Success : IdentityResult.Failed(errors.ToArray()));          }      }  } |

Здесь проверяются пара условий: наличие слова "admin" и домен электронного адреса. В зависимости от результатов проверки добавляются соответствующие ошибки в коллекцию errors. И если ошибок не будет найдено, что значит валидация прошла успешно, и возвращается значение IdentityResult.Success.

Теперь применим наш валидатор. Для этого добавим установку валидатора в метод ConfigureServices() класса Startup:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public void ConfigureServices(IServiceCollection services)  {      services.AddTransient<IUserValidator<User>, CustomUserValidator>();        services.AddDbContext<ApplicationDbContext>(options =>          options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));        services.AddIdentity<User, IdentityRole>()          .AddEntityFrameworkStores<ApplicationDbContext>();        services.AddControllersWithViews();  } |

И в этом случае, если мы введем некорректные значения, то увидим сообщения из нашего валидатора:



Также для создания валидатора мы можем унаследовать свой класс от класса UserValidator:

|  |  |
| --- | --- |
| 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 class CustomUserValidator : UserValidator<User>  {      public override Task<IdentityResult> ValidateAsync(UserManager<User> manager, User user)      {          List<IdentityError> errors = new List<IdentityError>();            if (user.Email.ToLower().EndsWith("@spam.com"))          {              errors.Add(new IdentityError              {                  Description = "Данный домен находится в спам-базе. Выберите другой почтовый сервис"              });          }          if (user.UserName.Contains("admin"))          {              errors.Add(new IdentityError              {                  Description = "Ник пользователя не должен содержать слово 'admin'"              });          }          return Task.FromResult(errors.Count == 0 ?              IdentityResult.Success : IdentityResult.Failed(errors.ToArray()));      }  } |

Этот класс также реализует интерфейс IUserValidator, поэтому для его переопределения мы можем изменить метод ValidateAsync(). Результат будет тем же, что и в предыдущем случае.

## Управление ролями

Для работы с ролями ASP.NET Core Identity предоставляет класс **IdentityRole**. В этом классе определяется несколько свойств:

|  |  |
| --- | --- |
| 1  2  3  4 | public virtual TKey Id { get; set; }  public virtual string Name { get; set; }  public virtual string NormalizedName { get; set; }  public virtual string ConcurrencyStamp { get; set; } |

Основым является свойство Name, которое и хранит название роли.

Возьмем какой-нибудь имеющийся контроллер и применим к нему доступ по ролям. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | using Microsoft.AspNetCore.Authorization;    [Authorize(Roles="admin")]  public class HomeController : Controller  {      // содержимое контроллера  } |

В данном случае мы указали, что доступ к контроллеру HomeController будет открыт только для пользователей, которые принадлежат к роли "admin". Но на данный момент у нас нет никакой роли "admin".

Рассмотрим как мы можем управлять ролями и для это продолжим работу с проектом из прошлой темы (либо создадим новый проект с ASP.NET Core Identity). Для администрирования ролями воспользуемся методами классов UserManager и RoleManager.

Вначале добавим в папку *ViewModels* модель **ChangeRoleViewModel**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using Microsoft.AspNetCore.Identity;  using System.Collections.Generic;    public class ChangeRoleViewModel  {      public string UserId { get; set; }      public string UserEmail { get; set; }      public List<IdentityRole> AllRoles { get; set; }      public IList<string> UserRoles { get; set; }      public ChangeRoleViewModel()      {          AllRoles = new List<IdentityRole>();          UserRoles = new List<string>();      }  } |

Эта модель позволит управлять всеми ролями для одного пользователя (в ASP.NET Core Identity один пользователь може иметь множество ролей).

Вначале добавим в папку Controllers новый контроллер **RolesController**:

|  |  |
| --- | --- |
| 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 | using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Identity;  using Microsoft.AspNetCore.Identity.EntityFrameworkCore;  using CustomIdentityApp.Models;  using CustomIdentityApp.ViewModels;    namespace CustomIdentityApp.Controllers  {      public class RolesController : Controller      {          RoleManager<IdentityRole> \_roleManager;          UserManager<User> \_userManager;          public RolesController(RoleManager<IdentityRole> roleManager, UserManager<User> userManager)          {              \_roleManager = roleManager;              \_userManager = userManager;          }          public IActionResult Index() => View(\_roleManager.Roles.ToList());            public IActionResult Create() => View();          [HttpPost]          public async Task<IActionResult> Create(string name)          {              if (!string.IsNullOrEmpty(name))              {                  IdentityResult result = await \_roleManager.CreateAsync(new IdentityRole(name));                  if (result.Succeeded)                  {                      return RedirectToAction("Index");                  }                  else                  {                      foreach (var error in result.Errors)                      {                          ModelState.AddModelError(string.Empty, error.Description);                      }                  }              }              return View(name);          }            [HttpPost]          public async Task<IActionResult> Delete(string id)          {              IdentityRole role = await \_roleManager.FindByIdAsync(id);              if (role != null)              {                  IdentityResult result = await \_roleManager.DeleteAsync(role);              }              return RedirectToAction("Index");          }            public IActionResult UserList() => View(\_userManager.Users.ToList());            public async Task<IActionResult> Edit(string userId)          {              // получаем пользователя              User user = await \_userManager.FindByIdAsync(userId);              if(user!=null)              {                  // получем список ролей пользователя                  var userRoles = await \_userManager.GetRolesAsync(user);                  var allRoles = \_roleManager.Roles.ToList();                  ChangeRoleViewModel model = new ChangeRoleViewModel                  {                      UserId = user.Id,                      UserEmail = user.Email,                      UserRoles = userRoles,                      AllRoles = allRoles                  };                  return View(model);              }                return NotFound();          }          [HttpPost]          public async Task<IActionResult> Edit(string userId, List<string> roles)          {              // получаем пользователя              User user = await \_userManager.FindByIdAsync(userId);              if(user!=null)              {                  // получем список ролей пользователя                  var userRoles = await \_userManager.GetRolesAsync(user);                  // получаем все роли                  var allRoles = \_roleManager.Roles.ToList();                  // получаем список ролей, которые были добавлены                  var addedRoles = roles.Except(userRoles);                  // получаем роли, которые были удалены                  var removedRoles = userRoles.Except(roles);                    await \_userManager.AddToRolesAsync(user, addedRoles);                    await \_userManager.RemoveFromRolesAsync(user, removedRoles);                    return RedirectToAction("UserList");              }                return NotFound();          }      }  } |

В контроллере получаем объекты RoleManager и UserManager, которые передаются через механизм внедрения зависимостей.

Для представлений для данного контроллера сразу определим в папке Views каталог *Roles*.

Метод Index выводит список ролей. Для их вывода добавим в каталог *Views/Roles* новое представление *Index.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | @model IEnumerable<Microsoft.AspNetCore.Identity.IdentityRole>    <h2>Список ролей</h2>  <table class="table">      @foreach (var role in Model)      {          <tr>              <td>@role.Name</td>              <td>                  <form asp-action="Delete" asp-route-id="@role.Id" method="post">                      <button type="submit" class="btn btn-sm btn-danger">                          Удалить                      </button>                  </form>              </td>          </tr>      }  </table>  <a asp-action="Create">Добавить роль</a>  <a asp-action="UserList">Список пользователей</a> |

В методе Create создаем роль и добавляем ее через вызов \_roleManager.CreateAsync(). Также добавим для этого метода представление *Create.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @model string    <div asp-validation-summary="All" class="text-danger"></div>    <form asp-action="Create" method="post">      <div class="form-group">          <label for="name">Новая роль</label>          <input name="name" class="form-control" />      </div>      <button type="submit" class="btn btn-primary">Добавить</button>  </form> |

В методе Delete по id получаем роль и удаляем ее с помощью вызова метода \_roleManager.DeleteAsync().

Отдельно стоит сказать про редактирование. Поскольку класс IdentityRole содержит только два свойства: Id и Name, то особо редактировать здесь нечего. Можно конечно изменять значение свойства Name, но это нежелательно, поскольку в приложении могут использоваться атрибуты Authorize, которые по имени роли устанавливают ограничение доступа. Но, конечно, если бы роль представляла какой-то производный класс от IdentityRole и имела бы какие-то дополнительные свойства, то там было бы больше возможностей по редактированию.

В данном же случае для примера используется редактирование списка ролей, к которым принадлежит определенный пользователь. Для вывода списка пользователей здесь определен метод UserList. Для вывода пользователей добавим следующее представление *UserList.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @model IEnumerable<CustomIdentityApp.Models.User>    <h2>Список пользователей</h2>  <table class="table">      @foreach (var user in Model)      {          <tr>              <td>@user.Email</td>              <td>                  <a class="btn btn-sm btn-primary" asp-action="Edit" asp-route-userid="@user.Id">Права доступа</a>              </td>          </tr>      }  </table> |

По нажатию на ссылку в этом представлении в метод Edit передается id пользователя. Затем из этого метод в представление через специальную модель ChangeRoleViewModel передаются его id, email, роли, а также список вообще всех ролей из базы данных.

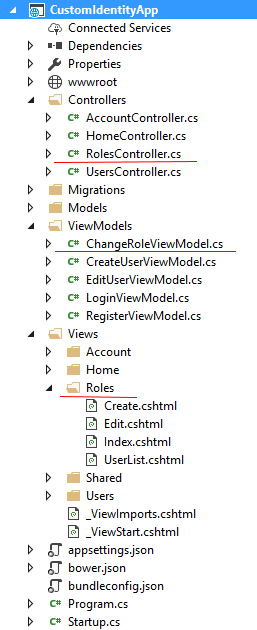
И также добавим для этого метода в папку *Views/Roles* представление *Edit.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @using Microsoft.AspNetCore.Identity  @model CustomIdentityApp.ViewModels.ChangeRoleViewModel    <h2>Изменение ролей для пользователя @Model.UserEmail</h2>    <form asp-action="Edit" method="post">      <input type="hidden" name="userId" value="@Model.UserId" />      <div class="form-group">          @foreach (IdentityRole role in Model.AllRoles)          {              <input type="checkbox" name="roles" value="@role.Name"                     @(Model.UserRoles.Contains(role.Name) ? "checked=\"checked\"" : "") />@role.Name <br />          }      </div>      <button type="submit" class="btn btn-primary">Сохранить</button>  </form> |

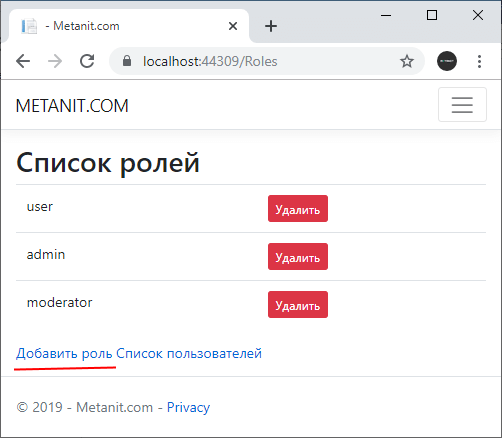
С помощью флажков (<input type="checkbox" >) здесь выводятся все роли. А те роли, для которых добавлен данный пользователь, имеют отмеченные флажки.

В post-версии действия Edit мы ожидаем получить id редактируемого пользователя и список выбранных ролей. С помощью операции разности множеств получаем добавленные и удаленные роли и далее применяем к ним методы \_userManager.AddToRolesAsync() и \_userManager.RemoveFromRolesAsync().

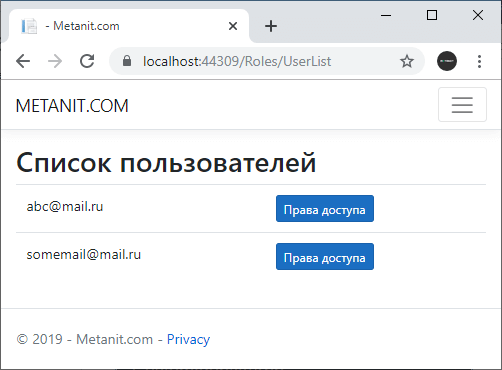
Основные моменты, добавленные в этой теме в проект:



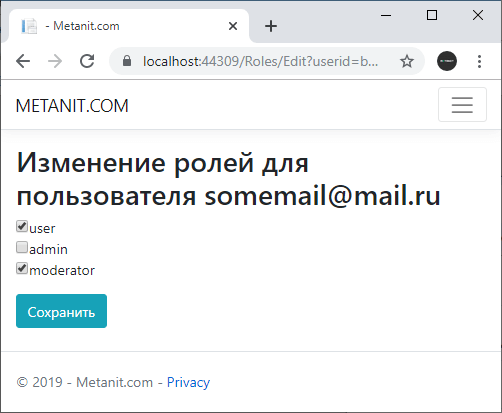
Добавим несколько ролей и перейдем к методу Index:



Перейдем к списку пользователей:



И изменим для кого-нибудь из них набор ролей:



## Инициализация БД ролями и пользователями

Нередко возникает необходимость, чтобы база данных к моменту первого обращения приложения уже содержала начальные данные, например, базовые роли и учетную запись администратора.

Для инициализации базы данных начальными ролями и пользователями определим новый класс RoleInitializer:

|  |  |
| --- | --- |
| 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 | using CustomIdentityApp.Models;  // пространство имен модели User  using Microsoft.AspNetCore.Identity;  using System.Threading.Tasks;    namespace CustomIdentityApp  {      public class RoleInitializer      {          public static async Task InitializeAsync(UserManager<User> userManager, RoleManager<IdentityRole> roleManager)          {              string adminEmail = "admin@gmail.com";              string password = "\_Aa123456";              if (await roleManager.FindByNameAsync("admin") == null)              {                  await roleManager.CreateAsync(new IdentityRole("admin"));              }              if (await roleManager.FindByNameAsync("employee") == null)              {                  await roleManager.CreateAsync(new IdentityRole("employee"));              }              if (await userManager.FindByNameAsync(adminEmail) == null)              {                  User admin = new User { Email = adminEmail, UserName = adminEmail };                  IdentityResult result = await userManager.CreateAsync(admin, password);                  if (result.Succeeded)                  {                      await userManager.AddToRoleAsync(admin, "admin");                  }              }          }      }  } |

В классе определен метод InitializeAsync(), который добавляет в базу данных две роли - "admin" и "user", а также одного пользователя - администратора. Для добавления здесь применяются методы классов UserManager и RoleManager, которые нам доступны через коллекцию сервисов приложения.

Для краткости все данные (логин и пароль администратора) здесь определены в коде, но естественно можно указать значения и в файле конфигурации и затем передавать в метод.

Используем данных класс в классе Program при старте приложения:

|  |  |
| --- | --- |
| 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 | using System;  using System.Threading.Tasks;  using CustomIdentityApp.Models;  using Microsoft.AspNetCore.Hosting;  using Microsoft.AspNetCore.Identity;  using Microsoft.Extensions.DependencyInjection;  using Microsoft.Extensions.Hosting;  using Microsoft.Extensions.Logging;    namespace CustomIdentityApp  {      public class Program      {          public static async Task Main(string[] args)          {                var host = CreateHostBuilder(args).Build();                using (var scope = host.Services.CreateScope())              {                  var services = scope.ServiceProvider;                  try                  {                      var userManager = services.GetRequiredService<UserManager<User>>();                      var rolesManager = services.GetRequiredService<RoleManager<IdentityRole>>();                      await RoleInitializer.InitializeAsync(userManager, rolesManager);                  }                  catch (Exception ex)                  {                      var logger = services.GetRequiredService<ILogger<Program>>();                      logger.LogError(ex, "An error occurred while seeding the database.");                  }              }                host.Run();          }            public static IHostBuilder CreateHostBuilder(string[] args) =>              Host.CreateDefaultBuilder(args)                  .ConfigureWebHostDefaults(webBuilder =>                  {                      webBuilder.UseStartup<Startup>();                  });      }  } |

Здесь с помощью метода services.GetRequiredService() получаем сервисы UserManager и RoleManager и передаем их в метод RoleInitializer.InitializeAsync. Причем поскольку здесь используется асинхронная версия метода Main, то нам необходимо установить в параметрах проекта в качестве версии языка как минимум версию 7.1, начиная с которой и была добавлена поддержка асинхронных методов Main.

В итоге при первом запуске в базу данных будут добавлены все необходимые данные, и мы сможем зайти из под администратора, используя указанные в коде логин и пароль.