# Авторизация и аутентификация

## ASP.NET Identity

В ASP.NET Core используется система авторизации и аутентификации под названием **ASP.NET Identity**.

***Основные классы:***

- **IdentityDbContext** (Microsoft.AspNetCore.Identity.EntityFrameworkCore) - контекст данных, производный от DbContext, который уже содержит свойства, необходимые для управления пользователями и ролями: свойства **Users** и **Roles**. В реальном приложении лучше создавать класс, производный от IdentityDbContext.

```
public class AccountDbContext : IdentityDbContext<User>
    {
        public AccountDbContext(DbContextOptions<AccountDbContext>
                                                              options)
            : base(options)
        {

            Database.EnsureCreated();
        }
    }
```

**Кроме Users и Roles в контексте есть свойства:**

- **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

- **IdentityUser** реализует интерфейс **IUser** и определяет следующие свойства:

**Claims**: возвращает коллекцию специальных атрибутов, которыми обладает пользователь и которые хранят о пользователе определенную информацию

**Email**: email пользователя

**Id**: уникальный идентификатор пользователя

**Logins**: возвращает коллекцию логинов пользователя

**PasswordHash**: возвращает хэш пароля

**Roles**: возвращает коллекцию ролей пользователя

**PhoneNumber**: возвращает номер телефона

**SecurityStamp**: возвращает некоторое значение, которое меняется при каждой смене настроек аутентификации для данного пользователя

**UserName**: возвращает ник пользователя
**AccessFailedCount**: число попыток неудачного входа в систему

**EmailConfirmed**: возвращает true, если email был подтвержден
**PhoneNumberConfirmed**: возвращает true, если телефонный номер был подтвержден
**TwoFactorEnabled**: если равен true, то для данного пользователя включена двухфакторная авторизация

Обычно для управления пользователями определяют класс, производный от IdentityUser:

```
public class ApplicationUser : IdentityUser
    {
    }
```

- **UserManager** осуществляет непосредственное управление пользователями

*Основные методы*:

- **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 имени
- **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 с именем name у пользователя user

- **UserStore<T>** - хранилище пользователей. Класс UserStore представляет реализацию интерфейса IUserStore<T>.Чтобы создать объект UserStore, необходимо использовать контекст данных ApplicationContext.

- **IdentityRole**

Свойства:

Id: уникальный идентификатор роли

Name: название роли

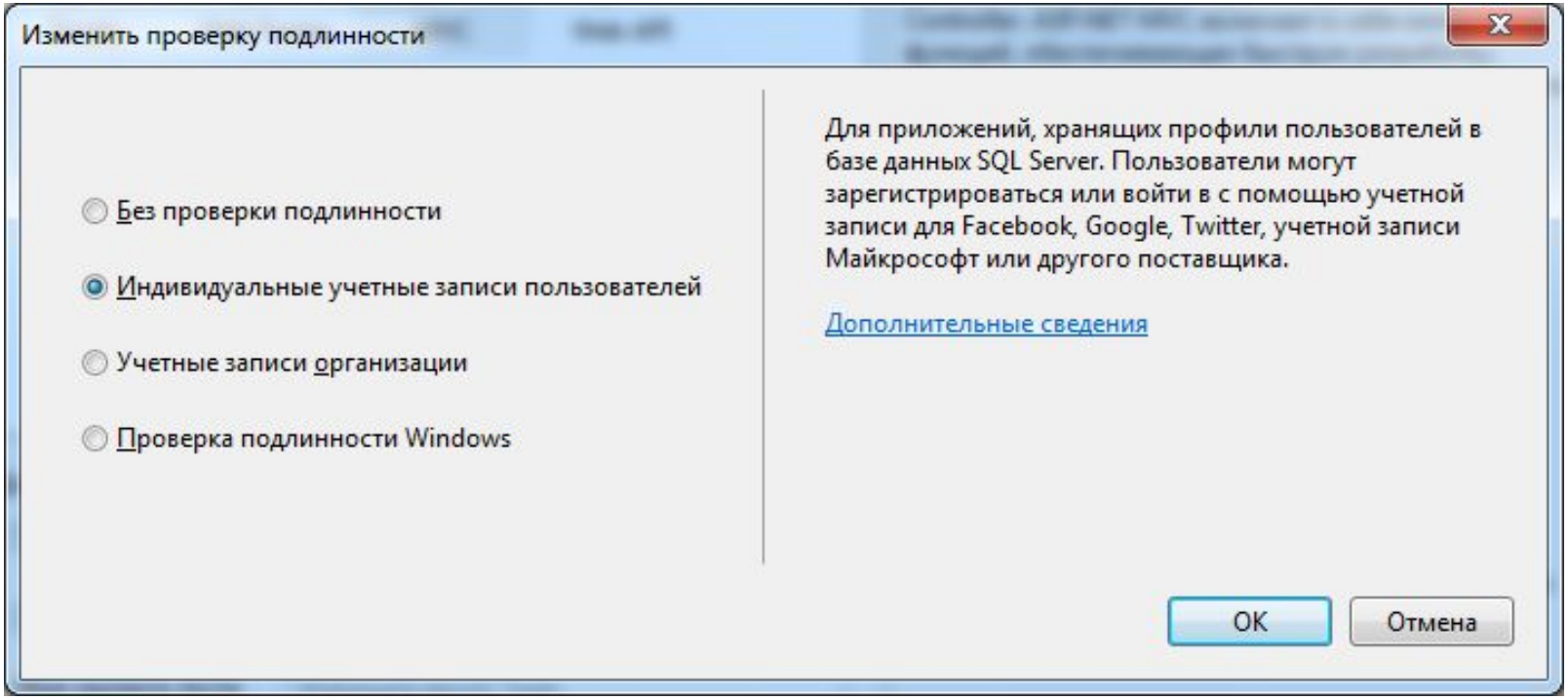
Users: коллекция объектов IdentityUserRole, который связывают пользователя и роль

- **RoleManager<T>**, где T - реализация интерфейса **IRole.**

Управляет ролями с помощью следующих методов:

- **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

# *Типы аутентификации*



**Пример**. Добавить аутентификацию и авторизацию в приложение – **Магазин телефонов**. Зарегистрированные пользователи могут делать заказ. Администратор может управлять пользователями. Гости могут только просматривать.

Для взаимодействия с MS SQL Server через ASP.NET Core Identity нужно добавить в проект через Nuget пакеты

- **Microsoft.AspNetCore.Identity.EntityFrameworkCore**
- **Microsoft.EntityFrameworkCore.SqlServer**

В проекте уже был создан контекст данных:

```
public class ShopContext : DbContext
    {
        public DbSet<Phone> Phones { get; set; }
        public DbSet<Order> Orders { get; set; }
        public ShopContext(DbContextOptions<ShopContext> options)
            : base(options)
        {
            //Database.EnsureCreated();
        }
    }
```

- Добавим классы пользователей и контекста данных в папку Models.

Класс пользователей:

```
public class User:IdentityUser
    {
        public string Lastname { get; set; }
        public string Address { get; set; }
    }
```

Класс контекста данных:

```
public class AccountDbContext : IdentityDbContext<User>
{
  public AccountDbContext(DbContextOptions<AccountDbContext> options)
            : base(options)
   {
            //Database.EnsureCreated();
   }
}
```

Для обеспечения возможности использования миграций добавим классы для конфигурирования подключения при создании контекста

```
class ContextFactory : IDesignTimeDbContextFactory<AccountDbContext>
    {
        public AccountDbContext CreateDbContext(string[] args)
        {
            var builder = new ConfigurationBuilder();
            // установка пути к текущему каталогу
            builder.SetBasePath(Directory.GetCurrentDirectory());
            // получаем конфигурацию из файла appsettings.json
            builder.AddJsonFile("appsettings.json");
            // создаем конфигурацию
            var config = builder.Build();
            // получаем строку подключения
            string connectionString =
config.GetConnectionString("DefaultConnection");
            var optionsBuilder = new DbContextOptionsBuilder<AccountDbContext>();

            DbContextOptions<AccountDbContext> options = optionsBuilder
                    .UseSqlServer(connectionString)
                    .Options;
            return new AccountDbContext(optionsBuilder.Options);

        }
    }
```

```
public class ShopContextFactory : IDesignTimeDbContextFactory<ShopContext>
    {
        public ShopContext CreateDbContext(string[] args)
        {
            var builder = new ConfigurationBuilder();
            // установка пути к текущему каталогу
            builder.SetBasePath(Directory.GetCurrentDirectory());
            // получаем конфигурацию из файла appsettings.json
            builder.AddJsonFile("appsettings.json");
            // создаем конфигурацию
            var config = builder.Build();
            // получаем строку подключения
            string connectionString =
config.GetConnectionString("DefaultConnection");
            var optionsBuilder = new DbContextOptionsBuilder<ShopContext>();
            DbContextOptions<ShopContext> options = optionsBuilder
                    .UseSqlServer(connectionString)
                    .Options;
            return new ShopContext(optionsBuilder.Options);

        }
    }
```

В классе Startup нужно применить все необходимые сервисы для работы с Identity и базой данных

```
string connection =
Configuration.GetConnectionString("DefaultConnection");
            services.AddDbContext<ShopContext>(options =>
options.UseSqlServer(connection));
            services.AddDbContext<AccountDbContext>(options =>
options.UseSqlServer(connection));
            services.AddIdentity<User, IdentityRole>()
                .AddEntityFrameworkStores<AccountDbContext>();
```

В методе Configure() нужно установить компонент middeware -вызвать **UseAuthentication**. Этот метод middleware вызывается перед app.UseEndpoints()

```
app.UseAuthentication();
```

Для инициализации базы данных начальными ролями и пользователями, а заодно и информацией о товарах определим класс

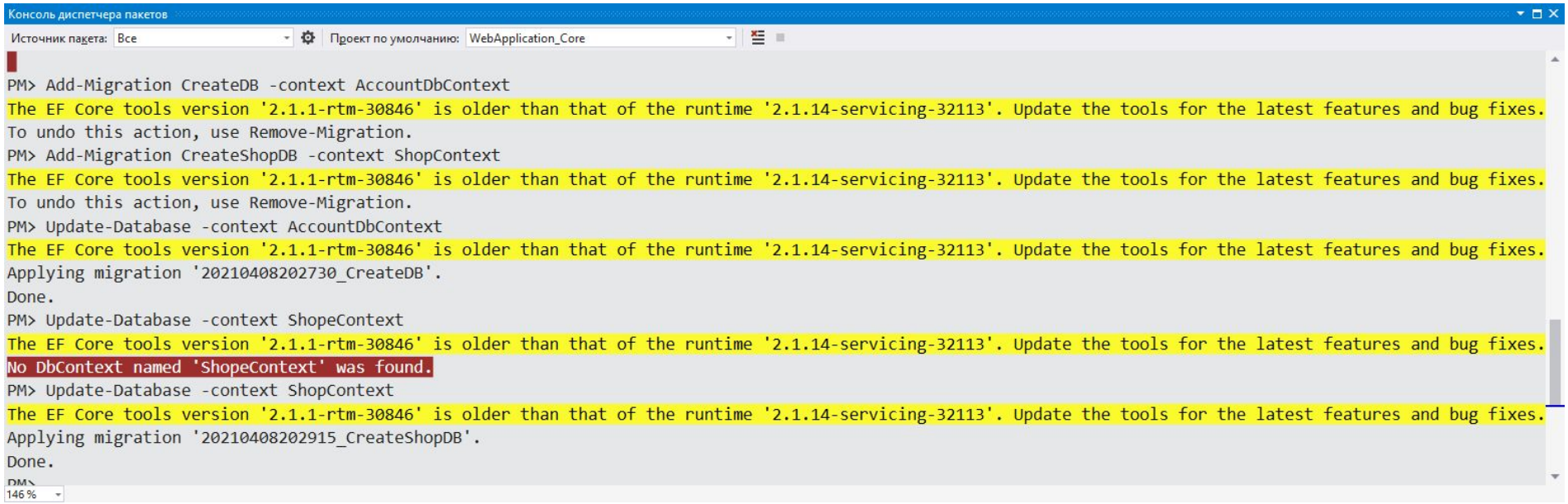
```
public class DbInitializer
   {
       public static async Task InitializeAsync(AccountDbContext accountContext,
                                ShopContext context, UserManager<User> userManager,
                                 RoleManager<IdentityRole> roleManager)
       {
           if (context.Phones.Count() == 0)
           {
               context.Phones.AddRange(
                   new Phone
                   {
                       Name = "Gnusmus galaxy note",
                       Company = "Gnusmus",
                       Price = 600
                   },
                   new Phone
                   {
                       Name = "Samsung Galaxy Note",
                       Company = "Samsung",
                       Price = 550
                   },
                   … и т.д.
               );
               context.SaveChanges();
           }
```

```
    if (await roleManager.FindByNameAsync("admin") == null)
     {
         await roleManager.CreateAsync(new IdentityRole("admin"));
       accountContext.SaveChanges();
     }
     if (await roleManager.FindByNameAsync("user") == null)
     {
         await roleManager.CreateAsync(new IdentityRole("user"));
         accountContext.SaveChanges();
     }
     string adminNik = "Boss";
     string password = "&Aa1234";
     if (await userManager.FindByNameAsync(adminNik) == null)
     {
         User admin = new User
         {
             UserName = adminNik,
             Address = "Гомель"
         };
         IdentityResult result = await userManager.CreateAsync(admin, password);
         if (result.Succeeded)
         {
             await userManager.AddToRoleAsync(admin, "admin");
             accountContext.SaveChanges();
         }

     }
}
```

# В методе Main можно вызвать метод инициализации:

```
var host = CreateWebHostBuilder(args).Build();
       using (var scope = host.Services.CreateScope())
       {
          var services = scope.ServiceProvider;
          try
          {

             var context = services.GetRequiredService<ShopContext>();
             var accountContext = services.GetRequiredService<AccountDbContext>();
             var userManager = services.GetRequiredService<UserManager<User>>();
            var rolesManager = services.GetRequiredService<RoleManager<IdentityRole>>();
   DbInitializer.InitializeAsync(accountContext, context, userManager
                                                          rolesManager).Wait();
 //await  DbInitializer.InitializeAsync(accountContext,context, userManager,
                                                          rolesManager);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "Ошибка при инициализации базы данных");
                }
            }
            host.Run();
```

Для создания (или изменения БД) нужно создать и применить миграции:



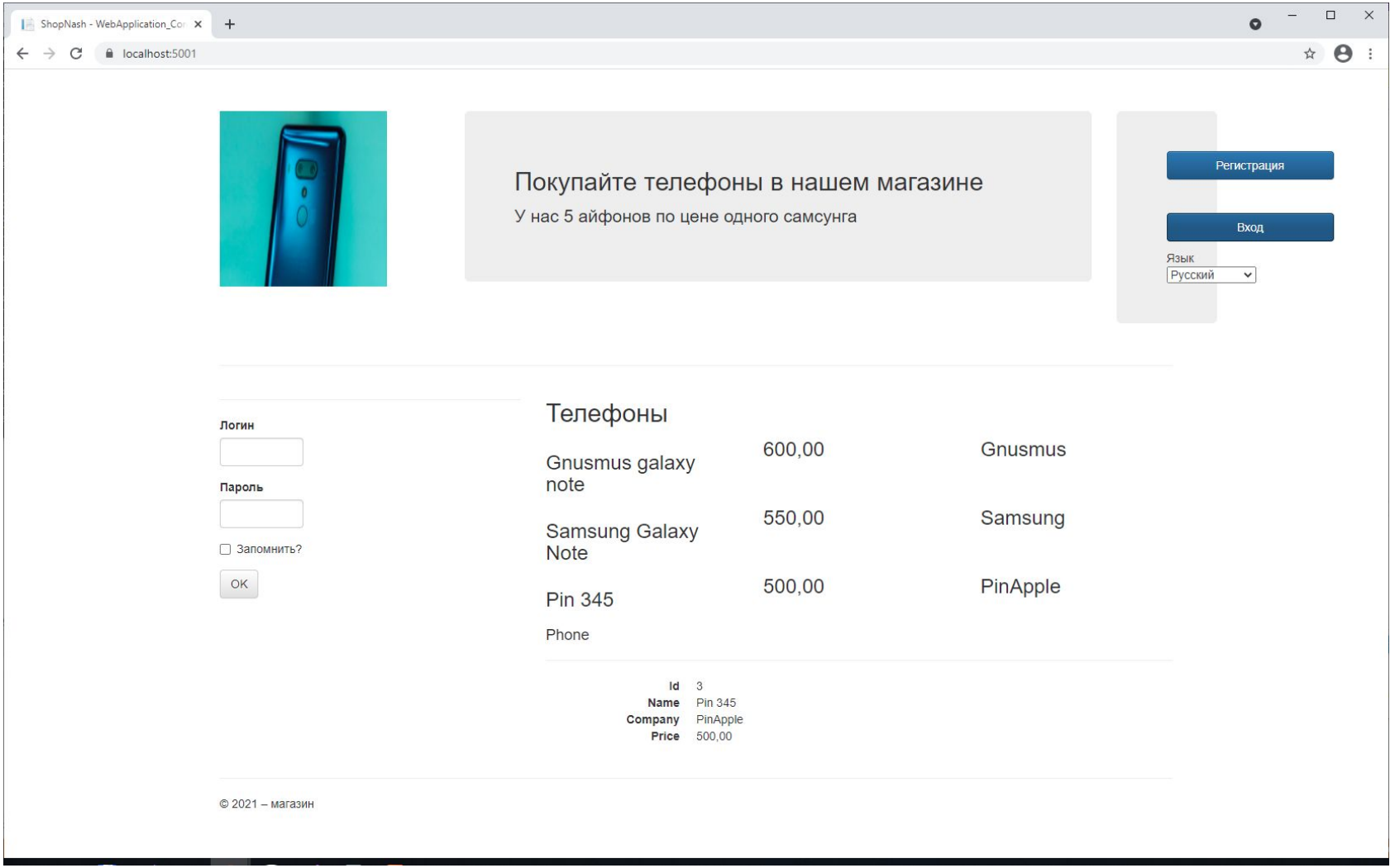
- Для создания функционала входа пользователей вначале добавим в проект в папку ViewModels специальную модель LoginModel:

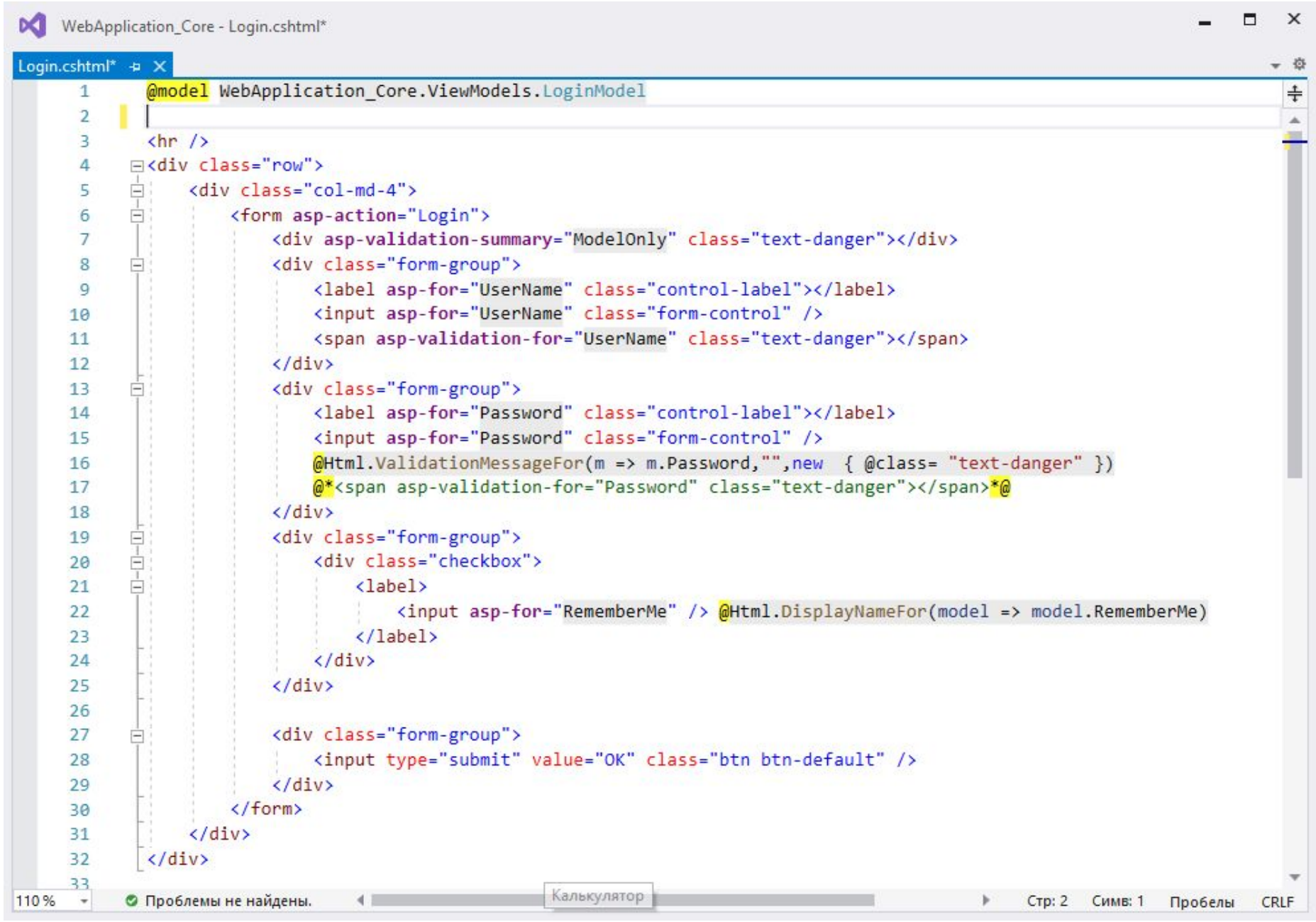
```
public class LoginModel
    {
        [Required(ErrorMessage ="Не задано имя пользователя")]
        [Display(Name = "Логин")]
        public string UserName { get; set; }

        [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; }
    }
```

- Добавим в папку *Controllers* новый контроллер AccountController и добавим в него:

```
public class AccountController : Controller
    {
        private readonly UserManager<User> userManager;
        private readonly SignInManager<User> signInManager;

        public AccountController(UserManager<User> userManager,
                                 SignInManager<User> signInManager)
        {
            this.userManager = userManager;
            this.signInManager = signInManager;
        }

 [HttpGet]
         public IActionResult Login(string returnUrl = null)
         {
             //returnUrl = HttpContext.Request.Path;
             return PartialView(new LoginModel
             {
                 ReturnUrl = returnUrl
             });
         }
```

# Создадим представление для входа:




WebApplication_Core - Login.cshtml*
Login.cshtml*
@model WebApplication_Core.ViewModels.LoginModel
<hr />

<form asp-action="Login">

<label asp-for="UserName" class="control-label"></label>
<input asp-for="UserName" class="form-control" />

<label asp-for="Password" class="control-label"></label>
<input asp-for="Password" class="form-control" />
@Html.ValidationMessageFor(m => m.Password,"",new { @class= "text-danger" })
@**@

<label>
<input asp-for="RememberMe" /> @Html.DisplayNameFor(model => model.RememberMe)
</label>

<input type="submit" value="OK" class="btn btn-default" />

</form>

110 %
Проблемы не найдены.
Калькулятор
Стр: 2
Симв: 1
Пробелы
CRLF

Добавим в контроллер действие, которое будет выполняться в ответ на POST запрос по кнопке **Вход**.

```
[HttpPost]
 [ValidateAntiForgeryToken]
 public IActionResult Login(LoginModel model)
 {
     if (ModelState.IsValid)
     {
      var result =
           signInManager.PasswordSignInAsync(model.UserName,
              model.Password, model.RememberMe, false).Result;

        if (result.Succeeded)
         {

               return RedirectToAction("Index", "Home");
          }
          else
          {
     ModelState.AddModelError("", "Неправильный логин и (или) пароль");
           }

     }
     return View(model);
 }
```

# Метод для выхода

```
public async Task<ActionResult> Logout()
 {
     await signInManager.SignOutAsync();
     return RedirectToAction("Index", "Home");
 }
```

- Добавим функционал регистрации пользователей.

```
public class RegisterModel
    {
        [Required]
        public string Email { get; set; }

        [Required]
        public int Age { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }

        [Required]
        [Compare("Password", ErrorMessage = "Пароли не совпадают")]
        [DataType(DataType.Password)]
        public string PasswordConfirm { get; set; }
    }
```

Перечисление DataType может принимать несколько различных значений:

| Значение | Описание |
| --- | --- |
| Currency | Отображает текст в виде валюты |
| DateTime | Отображает дату и время |
| Date | Отображает только дату, без времени |
| Time | Отображает только время |
| Text | Отображает однострочный текст |
| MultilineText | Отображает многострочный текст (элемент textarea) |
| Password | Отображает символы с использованием маски |
| Url | Отображает строку URL |
| EmailAddress | Отображает электронный адрес |

- Добавим в контроллер метод регистрации пользователей:

```
//регистрация пользователей
        public ActionResult Register()
        {
            return View();
        }
```

- Создадим представление для регистрации пользователя

```
@model Identity_Post.Models.RegisterModel

@{
    ViewBag.Title = "Регистрация";
}
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div>
        <h4>Регистрация пользователя</h4>
        <hr />
        @Html.ValidationSummary()

        <div>
            Имя
            <div>
                @Html.EditorFor(model => model.Name)
            </div>
        </div>

        <div>
            Электронный адрес
            <div>
                @Html.EditorFor(model => model.Email)
            </div>
        </div>
```

```
<div>
            Возраст
            <div>
                @Html.EditorFor(model => model.Age)
            </div>
        </div>
        <div>
            Пароль
            <div>
                @Html.EditorFor(model => model.Password)
            </div>
        </div>

        <div>
            Подтвердить пароль
            <div>
                @Html.EditorFor(model => model.PasswordConfirm)
            </div>
        </div>
        <div>
            <div>
                <input type="submit" value="Зарегистрировать" />
            </div>
        </div>
    </div>
}
```

- Добавим в контроллер Post-версию метода регистрации

```
[HttpPost]
         [ValidateAntiForgeryToken]
        public async Task<ActionResult> Register(RegisterModel model)
        {
            if (ModelState.IsValid)
            {
            ApplicationUser user = new ApplicationUser { UserName = model.Name,
Email = model.Email, Age = model.Age };
    IdentityResult result = await UserManager.CreateAsync(user, model.Password);
                if (result.Succeeded)
                {
                    return RedirectToAction("Index", "Home");
                }
                else
                {
                    foreach (string error in result.Errors)
                    {
                        ModelState.AddModelError("", error);
                    }
                }
            }
            SetViewBag();
            return View(model);
        }
```

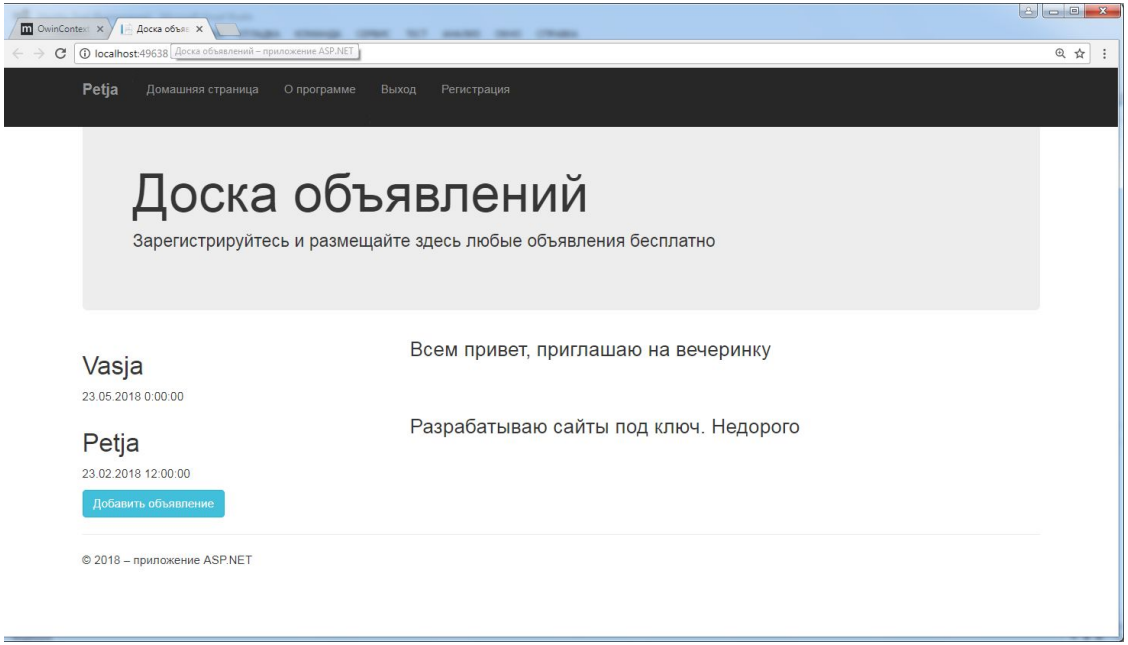
```
private void SetViewBag()
        {
            bool isAuthenticated =
HttpContext.GetOwinContext().Authentication.User.Identity.IsAuthenticated;

            if (isAuthenticated)
            {
                ViewBag.Login = "Выход";
                ViewBag.LoginAction = "Logout";
                ViewBag.UserName =
HttpContext.GetOwinContext().Authentication.User.Identity.Name;
            }
            else
            {
                ViewBag.Login = "Вход";
                ViewBag.LoginAction = "Login";
                ViewBag.UserName = "Гость";
            }
        }
```

***Задание***. Заменить этот метод фильтром.

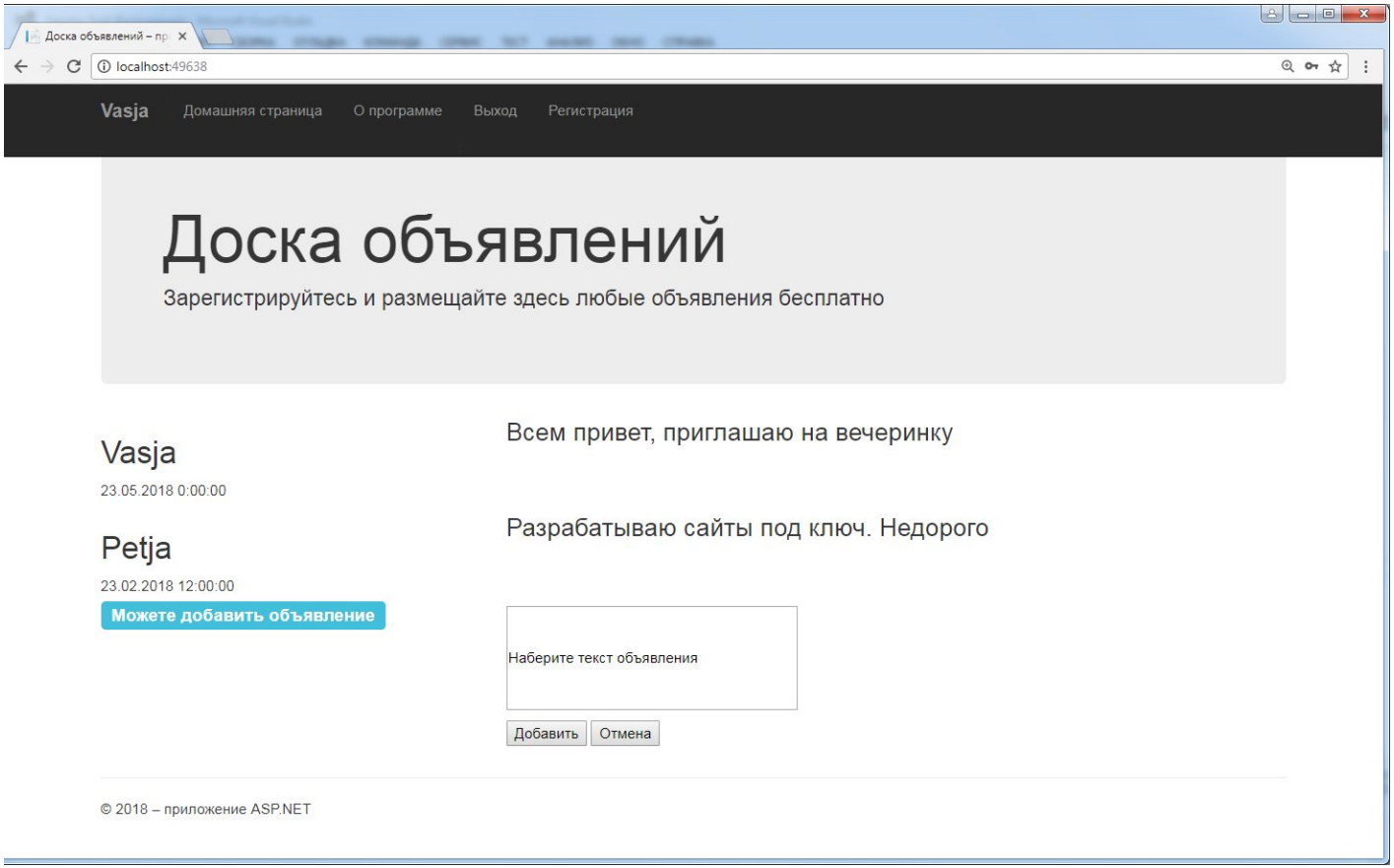
- Для получения сохраненных в базе данных объявлений в действии Index обратимся к методу-расширению Get для IOwinContext. Для этого нужно импортировать пространство имен Microsoft.AspNet.Identity.Owin.

```
IEnumerable<Post> posts =
HttpContext.GetOwinContext().Get<ApplicationContext>().Post.ToList();
```

- Изменим представление Index, чтобы страница выглядела следующим образом:



# Вид страницы после нажатия кнопки:



## *Фильтры аутентификации*

Фильтры аутентификации срабатывают до любого другого фильтра и выполнения метода, а также тогда, когда метод уже завершил выполнение, но его результат - объект ActionResult - не обработан.

Фильтры аутентификации реализуют интерфейс **IAuthenticationFilter**:

```
public interface IAuthenticationFilter
    {
        void OnAuthentication(AuthenticationContext filterContext);
        void OnAuthenticationChallenge(AuthenticationChallengeContext
filterContext);
    }
```

Реализация метода OnAuthentication используется для проверки пользователя: аутентифицирован ли он в системе.

Метод OnAuthenticationChallenge используется для ограничения доступа для аутентифицированного пользователя.

*Метод* *OnAuthenticationChallenge()* вызывается инфраструктурой MVC Framework каждый раз, когда запрос не прошел аутентификацию или не удовлетворил политикам авторизации для метода действия.

Методу OnAuthenticationChallenge() передается экземпляр класса **AuthenticationChallengeContext**, производного от класса ControllerContext, который был описан ранее и определяет свойства:

- **ActionDescriptor** Возвращает объект ActionDescriptor, описывающий метод действия, к которому был применен фильтр
- **Result** Устанавливает объект ActionResult, который выражает результат запроса аутентификации

Методу OnAuthentication() передается экземпляр класса **AuthenticationContext**, который подобно классу AuthenticationChallengeContext унаследован от ControllerContext. В классе AuthenticationContext также определены свойства ActionDescriptor и Result, а также *свойство* ***Principal*****,** которое возвращает объект ActionDescriptor, описывающий метод действия, к которому был применен фильтр.

Метод OnAuthentication() используется для создания результата, который сообщает пользователю об ошибке аутентификации и может затем быть переопределен методом OnAuthenticationChallenge()

Фильтры аутентификации могут также комбинироваться с фильтрами авторизации, чтобы проводить аутентификацию запросов, которые не соблюдают политику авторизации.

- Создадим свой фильтр аутентификации

```
public class AuthenticateAttribute : FilterAttribute,
IAuthenticationFilter
    {
     public void OnAuthentication(AuthenticationContext filterContext)
      {
         var user = filterContext.HttpContext.User;
         if (user == null || !user.Identity.IsAuthenticated)
            {
                filterContext.Result = new HttpUnauthorizedResult();
            }
      }

public void OnAuthenticationChallenge(AuthenticationChallengeContext
context)
     {
       if (context.Result == null || context.Result is
HttpUnauthorizedResult)
       {
         context.Result = new RedirectToRouteResult(new
System.Web.Routing.RouteValueDictionary {
            { "controller", "Account" }, { "action", "Login" } });
            }
        }
    }
```

Внутри реализации метода OnAuthentication() выполняется проверка, прошел ли запрос аутентификацию. Если запрос не был аутентифицирован, свойству Result объекта AuthenticationContext присваивается новый экземпляр **HttpUnauthorizedResult**.

Экземпляр класса HttpUnauthorizedResult устанавливается в качестве значения свойства Result для объекта AuthenticationChallengeContext, который передается методу OnAuthenticationChallenge().

- Применим фильтр к методу Contact

## *Работа с ролями*

- Добавим в модели класс ApplicationRoleManager

```
public class ApplicationRoleManager : RoleManager<IdentityRole>
    {
        public ApplicationRoleManager(RoleStore<IdentityRole> store)
            : base(store)
        { }
        public static ApplicationRoleManager
Create(IdentityFactoryOptions<ApplicationRoleManager> options,
                                                IOwinContext context)
        {
            return new ApplicationRoleManager(new

RoleStore<IdentityRole>(context.Get<ApplicationContext>()));
        }
    }
```

- Создадим класс для инициализации БД, в котором будут создаваться роли и пользователи с назначенными ролями

```
public class AppDbInitializer :
DropCreateDatabaseAlways<ApplicationContext>
    {
        protected override void Seed(ApplicationContext context)
        {
            var userManager = new ApplicationUserManager(
           new UserStore<ApplicationUser>(context));

            var roleManager = new ApplicationRoleManager(
           new RoleStore<IdentityRole>(context));

            // создаем две роли
            var role1 = new IdentityRole { Name = "admin" };
            var role2 = new IdentityRole { Name = "user" };

            // добавляем роли в бд
            roleManager.Create(role1);
            roleManager.Create(role2);
```

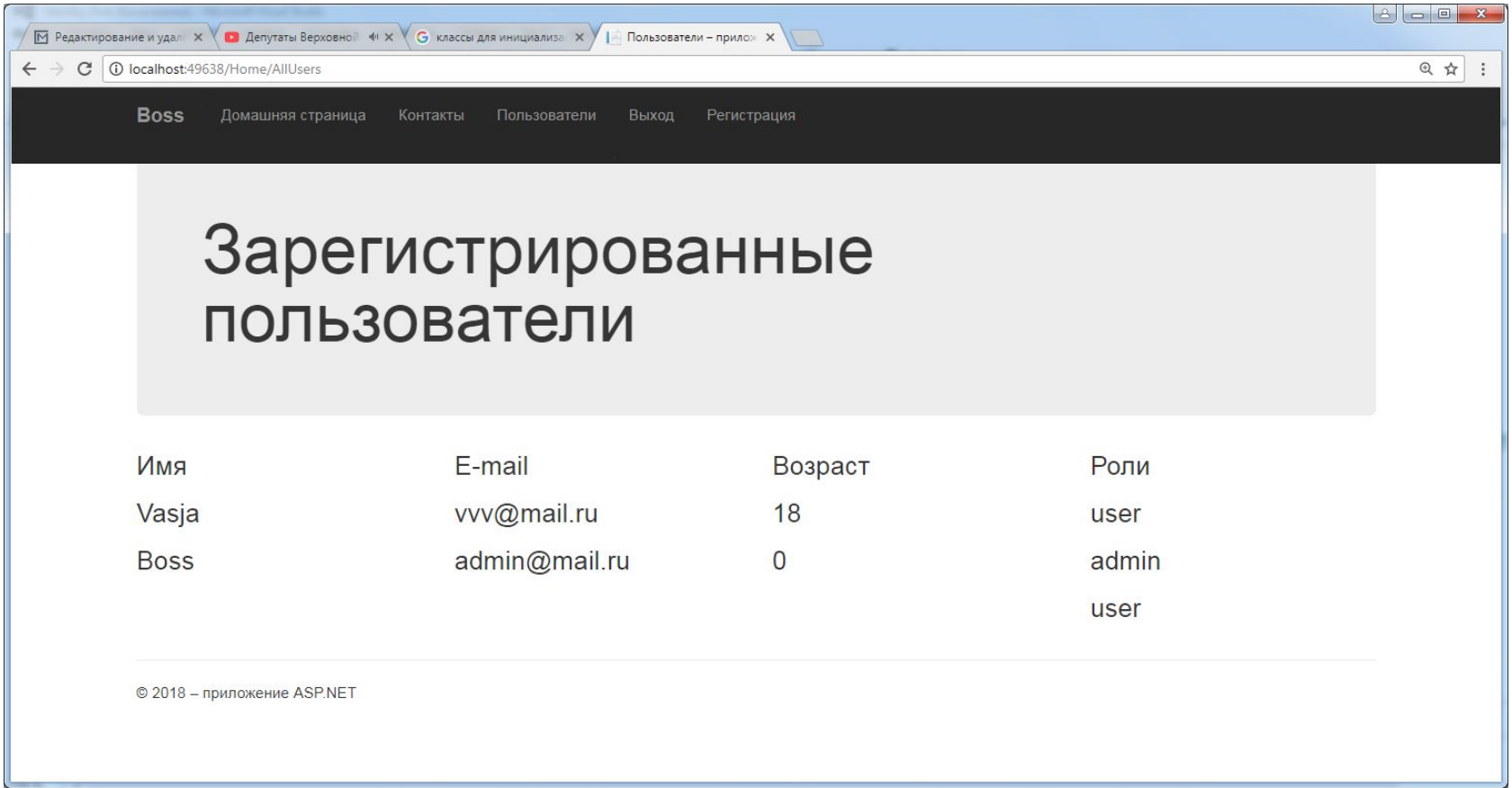
```
// создаем администратора
            var admin = new ApplicationUser {
           Email = "admin@mail.ru",
           UserName = "Boss" };
            string password = "123456";
            var result = userManager.Create(admin, password);

            // если создание пользователя прошло успешно
            if(result.Succeeded)
            {
                // добавляем для пользователя роль
                userManager.AddToRole(admin.Id, role1.Name);
                userManager.AddToRole(admin.Id, role2.Name);
            }

            base.Seed(context);
        }
    }
```

- добавим инициализацию БД в файл Global.asax

- В класс Startup добавляем

```
app.CreatePerOwinContext<ApplicationRoleManager>(
                                    ApplicationRoleManager.Create);
```

- Добавим действие контроллера и представление к нему для отображения информации обо всех зарегистрированных пользователях



- Сделаем так, чтобы просмотр информации о пользователях был доступен только администратору, используя фильтр авторизации [Authorize(Roles = "admin")]

***Фильтры авторизации*** срабатывают после фильтров аутентификации и до запуска остальных фильтров и вызова методов действий. https://metanit.com/sharp/mvc5/8.3.php

Цель фильтров авторизации - разграничить доступ пользователей, чтобы к определенным ресурсам приложения имели доступ только определенные пользователи.

Фильтры авторизации реализуют интерфейс **IAuthorizationFilter:**

```
public interface IAuthorizationFilter
    {
        void OnAuthorization(AuthorizationContext filterContext);
    }
```

Если при получении запроса окажется, что к запрашиваемому действию контроллера применяется данный фильтр, то сначала срабатывает метод OnAuthorization данного интерфейса. И если фильтр одобрит запрос, то далее вызывается действие. Иначе действие не будет работать.

В ASP.NET MVC 5 есть встроенная реализация данного фильтра - **AuthorizeAttribute**.

Применяется посредством установки атрибута [Authorize] *над контроллером* или *методом* контроллера.

Атрибут **AllowAnonymous** позволяет разрешить доступ к ресурсам для анонимных, не авторизованных пользователей.

Чтобы настроить доступ к ресурсам для отдельных пользователей или групп пользователей, можно использовать два свойства атрибута AuthorizeAtribute:

**Users** - содержит перечисление имен пользователей, которым разрешен вход

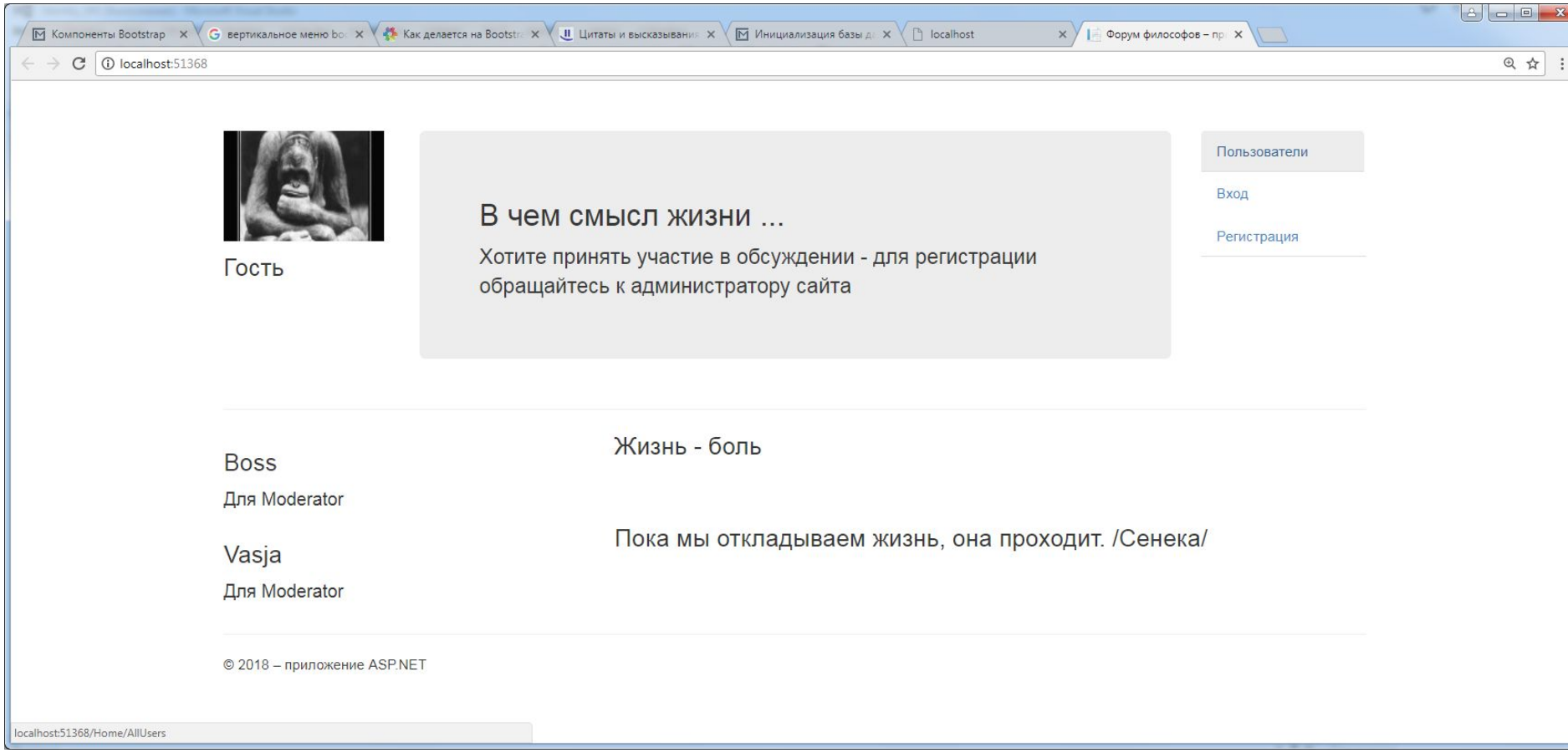
**Roles** - содержит перечисление имен ролей, которым разрешен вход

```
[Authorize (Roles="admin, moderator", Users="eugene, sergey")]
public ActionResult Edit()
{
    ..............
}
```

**Задание.** Разработать Web-приложение с использованием ASP.Net MVC *Форум на тему* «*О смысле жизни*»

- Неавторизованные пользователи могут просматривать сообщения других пользователей
- Обычные зарегистрированные участники могут просматривать и добавлять сообщения.
- Модераторы и пользователь Vasja могут редактировать сообщения
- Администратор может регистрировать и удалять пользователей

Использовать **ASP.NET Identity**


Компоненты Bootstrap
вертикальное меню bo
Как делается на Bootstra
Цитаты и высказывани
Инициализация базы да
localhost
Форум философов – пр
localhost:51368
Гость
В чем смысл жизни ...
Хотите принять участие в обсуждении - для регистрации обращайтесь к администратору сайта
Пользователи
Вход
Регистрация
Boss
Для Moderator
Жизнь - боль
Vasja
Для Moderator
Пока мы откладываем жизнь, она проходит. /Сенека/
© 2018 – приложение ASP.NET
localhost:51368/Home/AllUsers