Авторизация и аутентификация в MVC 5

ASP.NET Identity

Релиз ASP.NET MVC 5 ознаменовался выходом новой системой авторизации и аутентификации в .NET приложениях под названием ASP.NET Identity. Эта система пришла на смену провайдерам Simple Membership, которые были введены в ASP.NET MVC 4.

Рассмотрим систему авторизации и аутентификации ASP.NET Identity на примере приложения MVC 5. Итак, при создании приложения MVC 5 Visual Studio предлагает нам выбрать один из типов аутентификации:

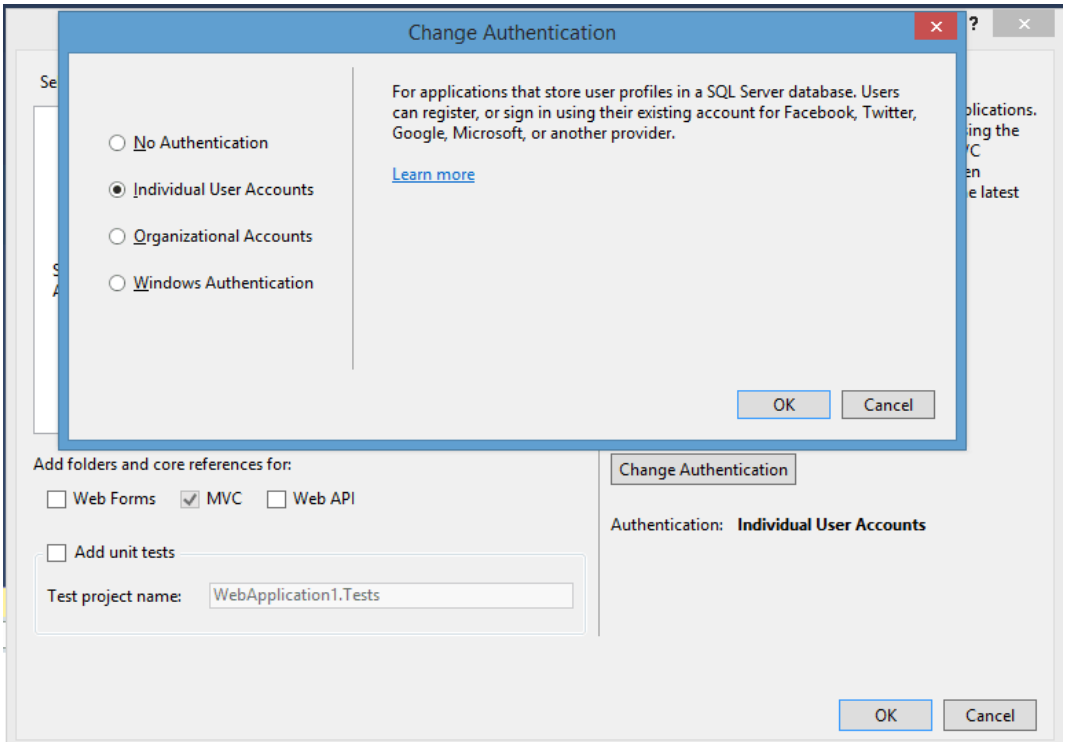

Рисунок 1 - Окно Change Authentication

Нажав на кнопку Change Authentication, мы можем изменить тип аутентификации, выбрав одно из следующих:

- No Authentication: ASP.NET Identity и встроенная система аутентификации отсутствует;

- Individual User Accounts: проект по умолчанию включает систему ASP.NET Identity, которая позволяет авторизовать как пользователей внутри приложения, так и с помощью внешних сервисов, как google, твиттер и т.д.;
- Organizational Accounts: подходит для сайтов и веб-приложений отдельных компаний и организаций;
- Windows Authentication: система аутентификации для сетей intranet с помощью учетных записей Windows.

Оставим значение по умолчанию, то есть Individual User Accounts и создадим проект.

Созданный проект уже по умолчанию имеет всю необходимую для авторизации инфраструктуру: модели, контроллеры, представления. Если мы заглянем в узел References (Библиотеки), то увидим там ряд ключевых библиотек, которые и содержит необходимые для авторизации и аутентификации классы:

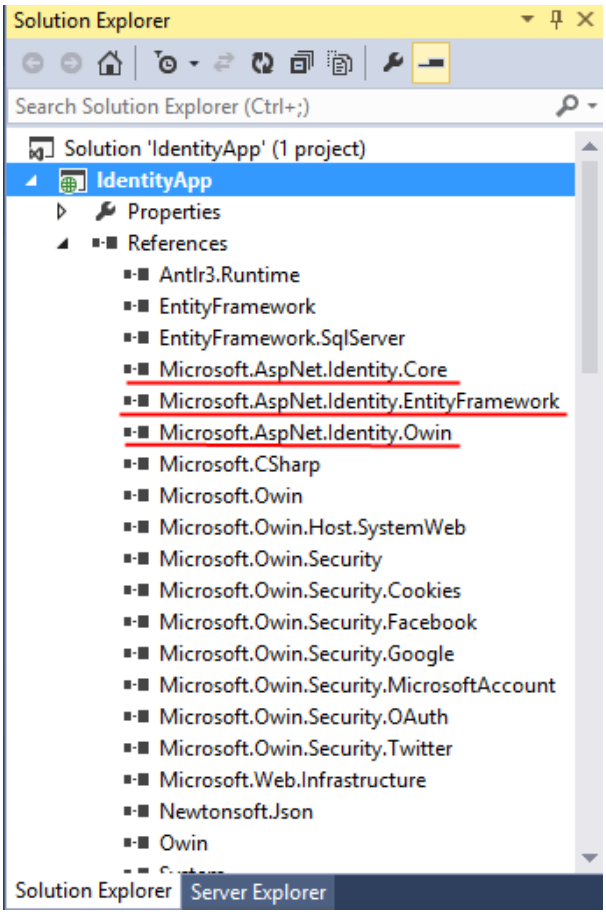

Рисунок 2 - Библиотеки для аутентификации через ASP.NET Identity

Это ряд библиотек OWIN, которые добавляют функциональность OWIN в проект, а также три библиотеки собственно ASP.NET Identity:

- Microsoft.AspNet.Identity.EntityFramework: содержит классы Entity Framework, применяющие ASP.NET Identity и осуществляющие связь с SQL Serverом;
- Microsoft.AspNet.Identity.Core: содержит ряд ключевых интерфейсов ASP.NET Identity. Реализация этих интерфейсов позволит выйти за рамки MS SQL Server и использовать в качестве хранилища учетных записей другие СУБД, в том числе системы NoSQL;
- Microsoft.AspNet.Identity.OWIN: привносит в приложение ASP.NET MVC аутентификацию OWIN с помощью ASP.NET Identity.

Поскольку вся инфраструктура уже имеется в проекте, запустим проект, на отобразившейся в браузере странице найдем ссылку Register и нажмем на нее. На открывшейся странице регистрации введем какие-нибудь данные:

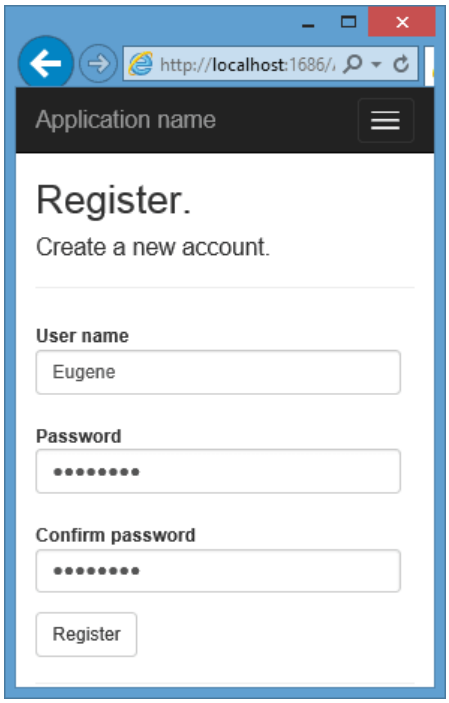

Рисунок 3 - Регистрация через ASP.NET Identity

После регистрации логин будет отображаться в правом верхнем углу веб-страницы веб-приложения. Осуществив регистрацию, мы можем разлогиниться, нажав на LogOff, и снова войти в систему. Таким образом, мы можем уже начать пользоваться встроенной системой аутентификации в приложении ASP.NET MVC 5. Теперь же рассомотрим ее основные моменты.

Во-первых, где это все хранится? Куда попадают данные зарегистрированных пользователей?

В данном случае используется подход Code First. В файле web.config уже имеется строка подключения по умолчанию, которая задает каталог базы данных. Если мы раскроем папку App_Data, то сможем увидеть созданную базу данных:

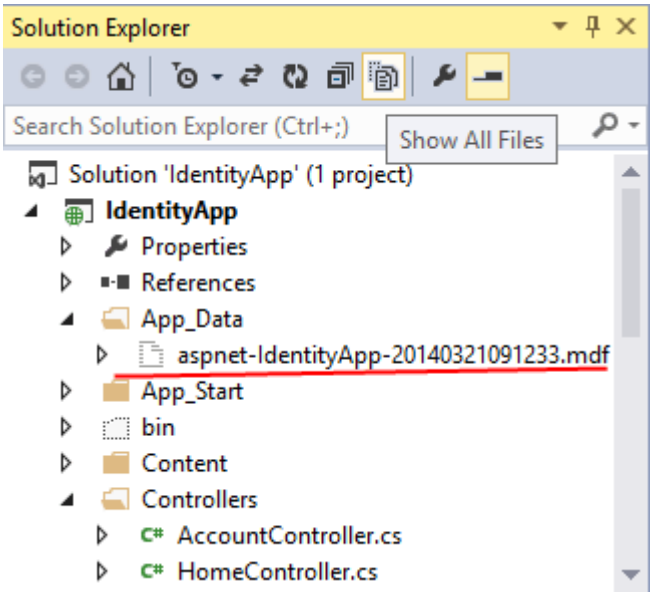

Рисунок 4 - База данных по умолчанию

Если вдруг в папке база данных не видна, нажмем вверху окна Solution Explorer на кнопку Show All Files (Показать все файлы).

Мы можем открыть эту базу данных в окне Server Explorer и увидеть ее содержимое:

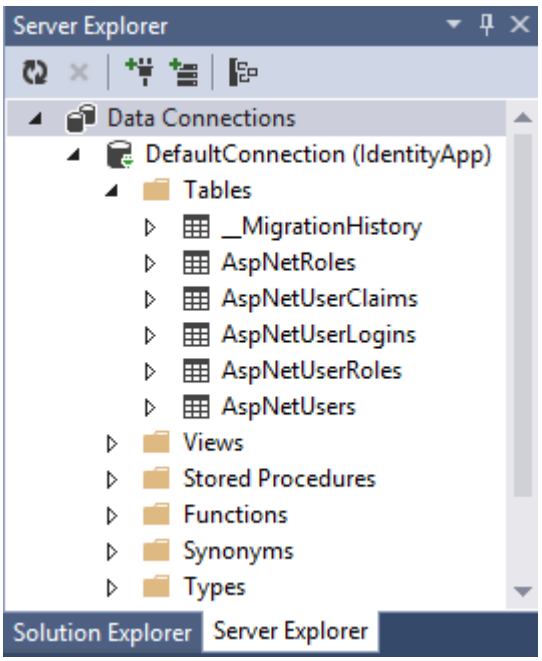

Рисунок 5 - База данных в ASP.NET Identity

По умолчанию при регистрации первого пользователя создается следующий набор таблиц:

- _MigrationHistory: используется EntityFramework для миграций БД;
- AspNetRoles: содержит определения ролей;
- AspNetUserClaims: таблица, хранящая набор клеймов (claim). Claim представляет иную модель авторизации по сравнению с ролями. Грубо говоря, claim содержит некоторую информацию о пользователе, например, адрес электронной почты, логин, возраст и т.д. И эта информация позволяет идентифицировать пользователя и наделить его соответствующими правами доступа;
- AspNetUserLogins: таблица логинов пользователя;
- AspNetUserRoles: таблица, устанавливающая для пользователей определенные роли;
- AspNetUsers: собственно таблица пользователей. Если мы ее откроем, то увидим данные зарегистрированного пользователя.

Ключевыми объектами в AspNet Identity являются пользователи и роли. Вся функциональность по созданию, удалению пользователей, взаимодействию с хранилищем пользователей хранится в классе UserManager. Для работы с ролями и их управлением в AspNet Identity определен класс RoleManager. Классы UserManager и RoleManager находятся в библиотеке Microsoft.AspNet.Identity.Core.

Рисунок 6 - библиотеке Microsoft.AspNet.Identity.Core

Каждый пользователь для UserManager представляет объект интерфейса IUser. А все операции по управлению пользователями производятся через хранилище, представленное объектом IUserStore.

Каждая роль представляет реализацию интерфейса IRole, а управление ролями классом RoleManager происходит через хранилище IRoleStore.

Непосредственную реализацию интерфейсов IUser, IRole, IUserStore и IRoleStore предоставляет пространство имен Microsoft.AspNet.Identity.EntityFramework:

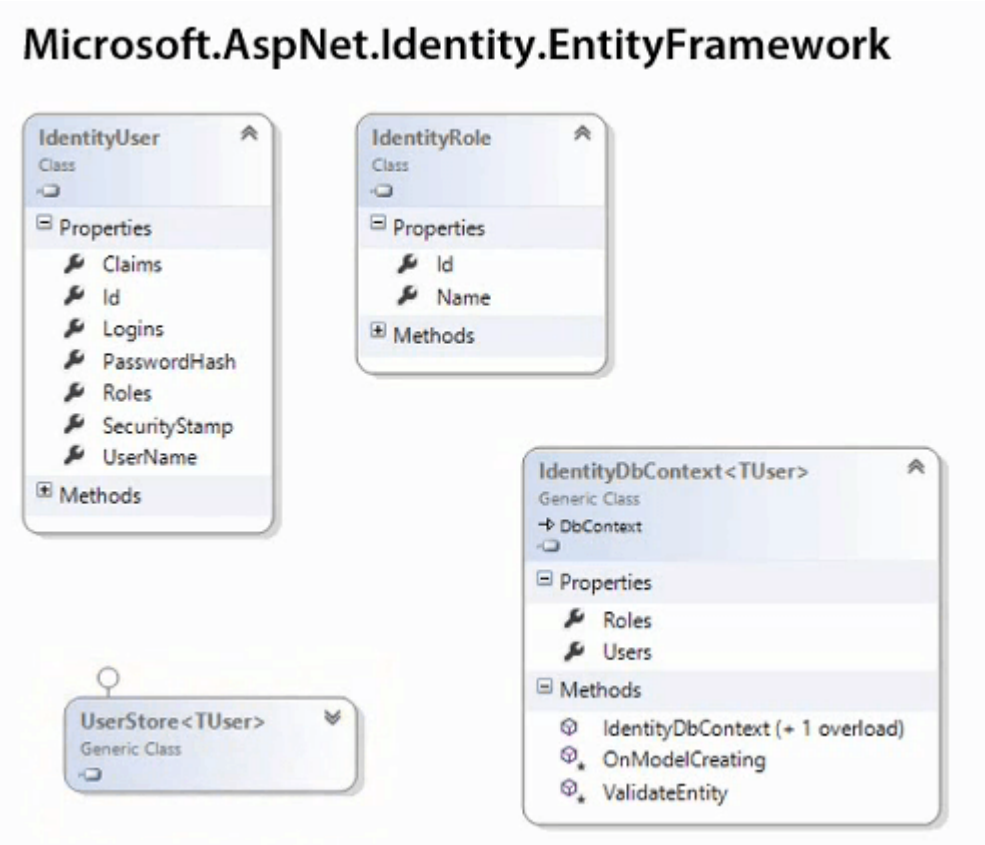

Рисунок 7 - пространство имен Microsoft.AspNet.Identity.EntityFramework

Класс IdentityUser является реализацией интерфейса IUser. А класс хранилища пользователей - UserStore реализует интерфейс IUserStore.

Подобным образом класс IdentityRole реализует интерфейс IRole, а класс хранилища ролей - RoleStore реализует интерфейс IRoleStore.

А для взаимодействия с базой данных в пространстве имен Microsoft.AspNet.Identity.EntityFramework определен класс контекста IdentityDbContext

В приложении ASP.NET MVC мы не будем работать напрямую с классами IdentityUser и IdentityDbContext. По умолчанию в проект в папку Models добавляется файл IdentityModels.cs, который содержит определения классов пользователей и контекста данных:

```
public class ApplicationUser : IdentityUser
{
   public async Task<ClaimsIdentity> GenerateUserIdentityAsync
                        (UserManager<ApplicationUser> manager)
   {
      var userIdentity = await manager.CreateIdentityAsync(this,
                        DefaultAuthenticationTypes.ApplicationCookie);
      return userIdentity;
   }
}

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
   public ApplicationDbContext()
      : base("DefaultConnection", throwIfV1Schema: false)
   {
   }

   public static ApplicationDbContext Create()
   {
      return new ApplicationDbContext();
   }
}
```

В приложении мы не работаем напрямую с классами IdentityUser и IdentityDbContext, а имеем дело с классами-наследниками.

Класс ApplicationUser наследует от IdentityUser все свойства. И кроме того добавляет метод GenerateUserIdentityAsync(), в котором с помощью вызова UserManager.CreateIdentityAsync создается объект ClaimsIdentity. Данный объект содержит информацию о данном пользователе.

Подобная архитектура позволяет взять уже готовый функционал и при необходимости добавить новый, например, добавить для пользователя новое свойство или добавить новую таблицу в бд.

Я не буду подробно расписывать весь функционал AspNet Identity, который по умолчанию добавляется в проект, обозначу вкратце лишь основные возможности.

Во-первых, чтобы задействовать AspNet Identity, в проект в папку App_Start добавляются два файла. Файл Startup.Auth.cs содержит класс запуска приложения OWIN. Поскольку AspNet Identity использует инфраструктуру OWIN, то данный класс является одним из ключевых и необходимых для работы.

Файл IdentityConfig.cs содержит ряд дополнительных вспомогательных классов: сервисы для двухфакторной валидации с помощью email и телефона EmailService и SmsService, класс менеджера пользователей ApplicationUserManager, добавляющий к UserManager ряд дополнительных функций, и класс ApplicationSignInManager, используемый для входа и выхода с сайта.

Базовая функциональность системы аутентификации и управления учетными записями расположена в двух контроллерах: AccountController и ManageController

В AccountController определены методы для логина, регистрации, верификации кода, отправленного по email или по смс, сброс пароля, напоминание пароля, вход на сайт с помощью внешних сервисов. Контроллер ManageController используется для управления учетной записью и предполагает возможности по смене пароля и управлению телефонными номерами в системе. Для обоих контроллеров уже по умолчанию генерируются все необходимые представления и специальные модели представлений.

Несмотря на то, что по умолчанию нам уже предоставляется готовый функционал, однако в ряде случаев, например, для отправки смс или электронной почты необходима дополнительная настройка. Теперь рассмотрим основные моменты системы AspNet Identity.

Базовые классы AspNet Identity

Систему AspNet Identity образует множество различных классов, предназначенных для различных задач. Рассмотрим основные классы.

Контекст данных IdentityDbContext

Для взаимодействия с базой данных в пространстве имен Microsoft.AspNet.Identity.EntityFramework определен класс IdentityDbContext. Это обычный контекст данных, производный от DbContext, который уже содержит свойства, необходимые для управления пользователями и ролями: свойства Users и Roles. Хотя в реальном приложении мы будем иметь дело с

классами, производными от IdentityDbContext. Например, у нас есть такой контекст данных:

```
public class ApplicationContext : IdentityDbContext<ApplicationUser>
{
   public ApplicationContext() : base("IdentityDb") { }

   public static ApplicationContext Create()
   {
      return new ApplicationContext();
   }
}
```

Выведем всех пользователей из бд:

```
public ActionResult Index()
{
   List<ApplicationUser> users = new List<ApplicationUser>();
   using(ApplicationContext db = new ApplicationContext())
   {
      users = db.Users.ToList();
   }

   return View(users);
}
```

Пользователи

Функциональность пользователей в AspNet Identity сосредоточена в классе IdentityUser, который определен в пространстве имен Microsoft.AspNet.Identity.EntityFramework. IdentityUser реализует интерфейс IUser и определяет следующие свойства:

Claims: возвращает коллекцию claims - 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.

Как правило, для управления пользователями определяют класс, производный от IdentityUser:

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

Менеджер пользователей UserManager

Непосредственное управление пользователями осуществляется с помощью класса UserManager<T>, которое находится в пространстве имен Microsoft.AspNet.Identity.

Этот класс определяет множество методов, которые имеют как синхронные, так и асинхронные версии. Перечислим основные:

- ChangePassword(id, old, new) / ChangePasswordAsync(id, old, new): изменяет пароль пользователя;
- Create(user) / CreateAsync(user): создает нового пользователя;
- Delete(user) / DeleteAsync(user): удаляет пользователя;
- Find(user, pass) / FindAsync(user, pass): ищет пользователя по определенному логину и паролю;
- FindById(id) /FindByIdAsync(id): ищет пользователя по id;
- FindByEmail(email) / FindByEmailAsync(email): ищет пользователя по email;

- FindByName(name) / FindByNameAsync(name): ищет пользователя по нику;
- Update(user) / UpdateAsync(user): обновляет пользователя
- Users: возвращает всех пользователей;
- AddToRole(id, name) / AddToRoleAsync(id, name): добавляет для пользователя с определенным id роль name;
- GetRoles(id) / GetRolesAsync (id): получает все роли пользователя по id;
- IsInRole(id, name) / IsInRoleAsync(id, name): возвращает true, если пользователь с данным id имеет роль name;
- RemoveFromRole(id, name) / RemoveFromRoleAsync(id, name): удаляет роль name у пользователя по id.

Создадим класс для управления пользователями:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;

namespace AspNetIdentityApp.Models
{
   public class ApplicationManager : UserManager<ApplicationUser>
   {
      public ApplicationManager(IUserStore<ApplicationUser> store) : base(store)
      {
      }
      public static ApplicationManager Create (IdentityFactoryOptions<ApplicationManager> options,
                                 IOwinContext context)
      {
         ApplicationContext db = context.Get<ApplicationContext>();
         ApplicationManager manager = new ApplicationManager(new UserStore<ApplicationUser>(db));
         return manager;
      }
   }
}
```

Для создания класса используется объект UserStore<ApplicationUser>. Класс UserStore представляет реализацию интерфейса IUserStore<T>. В свою очередь, чтобы создать объект UserStore, необходимо использовать контекст данных ApplicationContext.

Роли в AspNet Identity

Каждая роль в ASP.NET Identity представляет объект интерфейса IRole. В Entity Framework имеется встроенная реализация данного интерфейса в виде класса IdentityRole.

Класс IdentityRole имеет следующие свойства:

- Id: уникальный идентификатор роли;
- Name: название роли;
- Users: коллекция объектов IdentityUserRole, который связывают пользователя и роль.

Для управления ролями в ASP.NET Identity имеется класс RoleManager<T>, где T представляет реализацию интерфейса IRole. Класс RoleManager управляет ролями с помощью следующих синхронных и асинхронных методов:

- Create(role) / CreateAsync(role): создает новую роль с именем role;
- Delete(role) / DeleteAsync(role): удаляет роль с именем role;
- FindById(id) / FindByIdAsync(id): ищет роль по id;
- FindByName(name) / FindByNameAsync(name): ищет роль по названию;
- RoleExists(name) / RoleExistsAsync(name): возвращает true, если роль с данным названием существует;
- Update(role) / UpdateAsync(role): обновляет роль;
- Roles: возвращает набор всех имеющихся ролей.

Создание приложения с ASP.NET Identity с нуля

При создании проекта с типом аутентификации Individual User Accounts в него по умолчанию добавляется все необходимые файлы для работы с AspNet Identity. Однако, как правило, редко востребован весь стандартный функционал. Если нам нужна только регистрация и логин, то остальные файлы и неиспользуемый код будут просто висеть в проекте, либо

их придется удалять. Однако мы можем и выбрать любой другой тип проекта и в него уже добавить вручную функционал AspNet Identity, причем только тот, который нам будет нужен. Это даст нам больший контроль над тем кодом, который размещается в проекте.

Например, создадим обычный проект MVC с типом аутентификации `No Authentication`.

Чтобы добавить в проект AspNet Identity, нам надо добавить следующие NuGet-пакеты:

Microsoft.AspNet.Identity.EntityFramework

Microsoft.AspNet.Identity.OWIN

Microsoft.Owin.Host.SystemWeb

После добавления пакетов надо обновить файл *web.config*: добавим строку подключения:

```
<connectionStrings>
  <add name="IdentityDb" providerName="System.Data.SqlClient"
    connectionString="Data
Source=(localdb)\v11.0;AttachDbFilename=|DataDirectory|\IdentityDb.mdf;Integr
ated Security=True;"/>
</connectionStrings>
```

Теперь добавим классы пользователей и контекста данных в папку Models. Класс контекста будет наследоваться от IdentityDbContext:

```
using System.Data.Entity;
using Microsoft.AspNet.Identity.EntityFramework;

public class ApplicationContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationContext() : base("IdentityDb") { }

    public static ApplicationContext Create()
    {
        return new ApplicationContext();
    }
}
```

Класс пользователей:

```
using Microsoft.AspNet.Identity.EntityFramework;

public class ApplicationUser : IdentityUser
{
    public int Year { get; set; }
    public ApplicationUser()
```

```
    {
    }
}
```

В классе пользователя кроме унаследованных от IdentityUser свойств также определяется и свойство Year для хранения года рождения пользователя.

Поскольку вся работа с пользователями идет не напрямую, а через менеджер пользователей, то также добавим в папку Models соответствующий класс:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;

public class ApplicationUserManager : UserManager<ApplicationUser>
{
    public ApplicationUserManager(IUserStore<ApplicationUser> store)
            : base(store)
    {
    }
    public static ApplicationUserManager Create (IdentityFactoryOp-
tions<ApplicationUserManager> options,
                                            IOwinContext context)
    {
        ApplicationContext db = context.Get<ApplicationContext>();
        ApplicationUserManager manager = new ApplicationUserManager(new Us-
erStore<ApplicationUser>(db));
        return manager;
    }
}
```

Класс менеджера пользователей наследуется от `UserManager`. В конструкторе он принимает объект хранилища пользователей IUserStore. А статический метод `Create()` создает экземпляр класса ApplicationUserManager с помощью объекта контекста`IOwinContext`

Последний шаг в первоначальной настройке проекта для AspNet Identity состоит в добавлении в проект файла запуска приложения OWIN. Итак, добавим в папку App_Start файл *Startup.cs* со следующим содержанием:

```
using Microsoft.Owin;
using Owin;
using AspNetIdentityApp.Models;
using Microsoft.Owin.Security.Cookies;
using Microsoft.AspNet.Identity;

[assembly: OwinStartup(typeof(AspNetIdentityApp.Startup))]

namespace AspNetIdentityApp
{
```

```
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // настраиваем контекст и менеджер

app.CreatePerOwinContext<ApplicationContext>(ApplicationContext.Create);

app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Creat
e);
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthentication-
Types.ApplicationCookie,
                LoginPath = new PathString("/Account/Login"),
            });
        }
    }
}
```

Интерфейс `IAppBuilder` определяет множество методов, в данном случае нам достаточно трех методов. Метод`CreatePerOwinContext` регистрирует в OWIN менеджер пользователей ApplicationUserManager и класс контекста ApplicationContext.

Метод `UseCookieAuthentication` с помощью объекта `CookieAuthenticationOptions` устанавливает аутентификацию на основе куки в качестве типа аутентификации в приложении. А свойство `LoginPath` позволяет установить адрес URL, по которому будут перенаправляться неавторизованные пользователи. Как правило, это адрес */Account/Login*.

Это минимально необходимая настройка проекта для использования AspNet Identity, на основе которой мы уже сможем создавать всю остальную систему авторизации и аутентификации.

Регистрация и создание пользователей в ASP.NET Identity

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

Класс IdentityUser определяет множество свойств, однако нам необязательно всех их устанавливать. И в данном случае для создания пользователей

нам лучше воспользоваться связующей моделью, которая установит все необходимые свойства. Итак, добавим в папку Models класс, который будет представлять пользователя:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace AspNetIdentityApp.Models
{
    public class RegisterModel
    {
        [Required]
        public string Email { get; set; }

        [Required]
        public int Year { 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; }
    }
}
```

Теперь добавим в папку *Controllers* новый контроллер AccountController:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using AspNetIdentityApp.Models;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;

namespace AspNetIdentityApp.Controllers
{
    public class AccountController : Controller
    {
        private ApplicationUserManager UserManager
        {
            get
            {
                return HttpCon-
text.GetOwinContext().GetUserManager<ApplicationUserManager>();
            }
        }

        public ActionResult Register()
        {
            return View();
        }

        [HttpPost]
        public async Task<ActionResult> Register(RegisterModel model)
```

```
        {
            if (ModelState.IsValid)
            {
                ApplicationUser user = new ApplicationUser { UserName = mod-
el.Email, Email = model.Email, Year = model.Year };
                IdentityResult result = await UserManager.CreateAsync(user,
model.Password);
                if (result.Succeeded)
                {
                    return RedirectToAction("Login", "Account");
                }
                else
                {
                    foreach (string error in result.Errors)
                    {
                        ModelState.AddModelError("", error);
                    }
                }
            }
            return View(model);
        }
    }
}
```

Первым делом в контроллере создается свойство UserManager, возвращающее объект ApplicationUserManager. Через него мы будем взаимодействовать с хранилищем пользователей. Для получения хранилища применяется выражение`HttpContext.GetOwinContext().GetUserManager<ApplicationUserManager>()`

Для собственно регистрации определен метод Register в Get- и Post-версиях. Post-версия представляет асинхронный метод, поскольку для создания пользователя здесь используется асинхронный вызов `UserManager.CreateAsync()`. Этот метод возвращает объект **IdentityResult**.

Если создание пользователя прошло успешно, то его свойство `Succeeded` будет равно `true`.

И в конце создадим представление для регистрации *Register.cshtml*:

```
@model AspNetIdentityApp.Models.RegisterModel

@{
    ViewBag.Title = "Register";
}

<h2>Регистрация</h2>


@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

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

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

        <div>
            Год рождения
            <div>
                @Html.EditorFor(model => model.Year)
            </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>
}
```

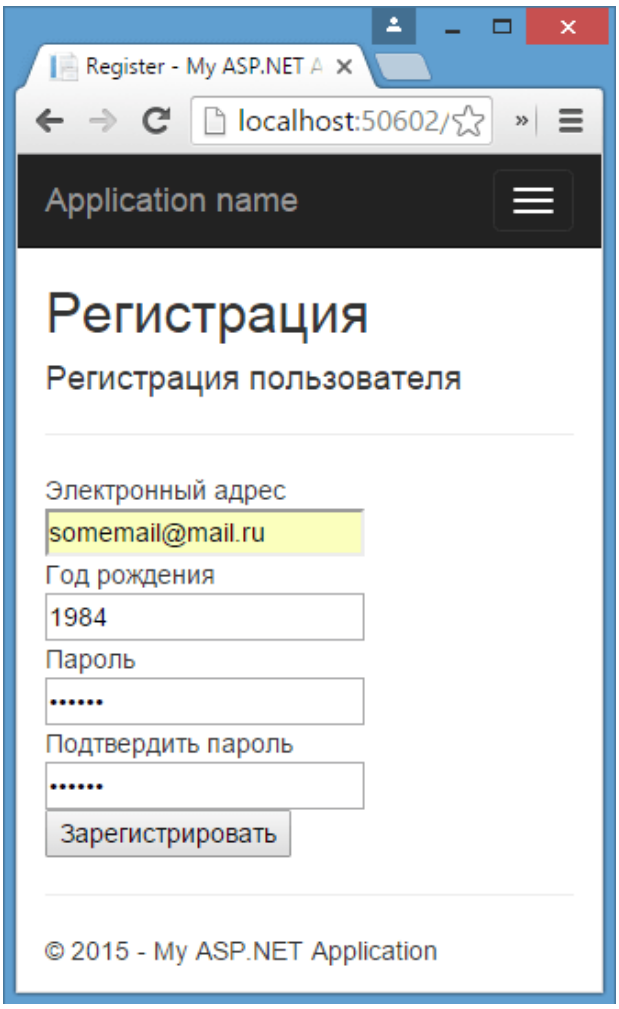

Рисунок 8 - представление для регистрации *Register.cshtml*

И после ввода данных данные о пользователе попадут в таблицу AspNetUsers:

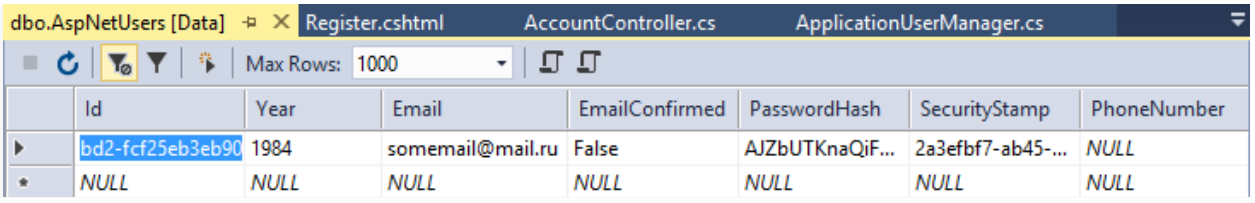

Рисунок 9 - таблицу AspNetUsers

Авторизация пользователей в ASP.NET Identity

Для создания инфраструктуры для входа пользователей на сайт вначале добавим в проект в папку Models специальную модель LoginModel:

```
public class LoginModel
{
    [Required]
    public string Email { get; set; }
    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}
```

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

```
private IAuthenticationManager AuthenticationManager
{
    get
    {
        return HttpContext.GetOwinContext().Authentication;
    }
}

public ActionResult Login(string returnUrl)
{
    ViewBag.returnUrl = returnUrl;
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        ApplicationUser user = await UserManager.FindAsync(model.Email, mod-
el.Password);
        if (user == null)
        {
            ModelState.AddModelError("", "Неверный логин или пароль.");
        }
        else
        {
            ClaimsIdentity claim = await UserManag-
er.CreateIdentityAsync(user,
                                        DefaultAuthentication-
Types.ApplicationCookie);
            AuthenticationManager.SignOut();
            AuthenticationManager.SignIn(new AuthenticationProperties
            {
                IsPersistent = true
            }, claim);
            if (String.IsNullOrEmpty(returnUrl))
                return RedirectToAction("Index", "Home");
            return Redirect(returnUrl);
        }
    }
    ViewBag.returnUrl = returnUrl;
    return View(model);
}
public ActionResult Logout()
{
    AuthenticationManager.SignOut();
    return RedirectToAction("Login");
```

```
}
```

Вначале определяется новое свойство, представляющее объект **IAuthenticationManager**, с помощью которого мы будем управлять входом на сайт. Для этого интерфейс IAuthenticationManager определяет два метода:

`SignIn()`: создает аутентификационные куки

`SignOut()`: удаляет аутентификационные куки

В Get-версии метода Login мы получаем адрес для возврата и передаем его в преставление.

В Post-версии метода Login получаем данные из представления в виде модели LoginModel и по ним пытаемся получить пользователя из бд с помощью метода `UserManager.FindAsync(model.Email, model.Password)`. Этот метод возвращает объект ApplicationUser в случае успеха поиска.

AspNet Identity использует аутентификацию на основе объектов claim. Поэтому нам надо сначала создать объект**ClaimsIdentity**, который представляет реализацию интерфейса IIdentity в AspNet Identity. Для создания ClaimsIdentity применяется метод `CreateIdentityAsync()`

И на финальном этапе вызывается метод `AuthenticationManager.SignIn()`, в который передается объект конфигурации аутентификации `AuthenticationProperties`, а также ранее созданный объект ClaimsIdentity. Свойство `IsPersistent` позволяет сохранять аутентификационные данные в браузере даже после закрытия пользователем браузера.

И метод `Logout` просто удаляет аутентификационные куки в браузере, как бы делая выход из системы.

Представление логина может выглядеть так:

```
@model AspNetIdentityApp.Models.LoginModel

@{
    ViewBag.Title = "Login";
}

<h2>Вход на сайт</h2>


@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div>
```

```
        @Html.ValidationSummary()
        <input type="hidden" name="returnUrl" value="@ViewBag.returnUrl" />
        <p>
            <label>Email</label><br />
            @Html.EditorFor(x => x.Email)
        </p>
        <p>
            <label>Пароль</label><br />
            @Html.EditorFor(x => x.Password)
        </p>
        <p><button type="submit">Войти</button></p>
    </div>
}
```

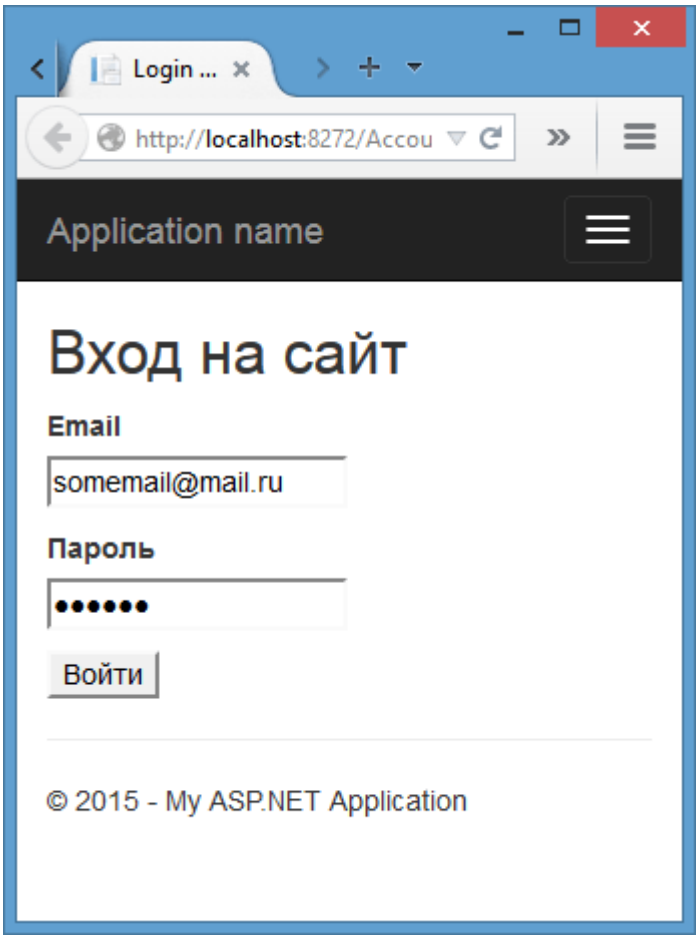

Рисунок 10 - Представление Login

Редактирование и удаление пользователей

Редактирование и удаление пользователей в ASP.NET Identity представляется довольно простой задачей благодаря классу UserManager. Так, возьмем проект из прошлой темы, где у нас уже есть механизм регистрации и логина пользователей. Добавим в контроллер AccountController следующие методы:

```
[HttpGet]
public ActionResult Delete()
{
    return View();
}

[HttpPost]
[ActionName("Delete")]
public async Task<ActionResult> DeleteConfirmed()
{
    ApplicationUser user = await UserManag-
er.FindByEmailAsync(User.Identity.Name);
    if (user != null)
    {
        IdentityResult result = await UserManager.DeleteAsync(user);
        if (result.Succeeded)
        {
            return RedirectToAction("Logout", "Account");
        }
    }
    return RedirectToAction("Index", "Home");
}

public async Task<ActionResult> Edit()
{
    ApplicationUser user = await UserManag-
er.FindByEmailAsync(User.Identity.Name);
    if (user != null)
    {
        EditModel model = new EditModel { Year = user.Year };
        return View(model);
    }
    return RedirectToAction("Login", "Account");
}

[HttpPost]
public async Task<ActionResult> Edit(EditModel model)
{
    ApplicationUser user = await UserManag-
er.FindByEmailAsync(User.Identity.Name);
    if (user != null)
    {
        user.Year = model.Year;
        IdentityResult result = await UserManager.UpdateAsync(user);
        if (result.Succeeded)
        {
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError("", "Что-то пошло не так");
        }
    }
    else
    {
        ModelState.AddModelError("", "Пользователь не найден");
    }

    return View(model);
}
```

Методы Delete и DeleteConfirmed отображают пользователю представление для удаления и принимают выбор пользователя об удалении. Для уда-

ления используется метод `UserManager.DeleteAsync()`. Он возвращает объект **IdentityResult**, который позволяет отследить успех операции.

Метод Edit также отображает представление для редактирования, передавая в него модель EditModel, которую мы далее создадим. В данном случае мы редактируем только значение свойства Year. POST-версия метода принимает данные модели и устанавливает значения ее свойств для пользователя. Редактирование также производится одним методом - методом`UserManager.UpdateAsync()`

Далее создадим модель EditModel:

```
public class EditModel
{
    public int Year { get; set; }
}
```

Пусть редактирование будет производиться только для года рождения пользователя.

Представление для удаления могло бы выглядеть так:

```
@{
    ViewBag.Title = "Delete";
}

<form method="post">
    <h2>Вы уверены, что хотите удалить свой аккаунт?</h2>
    <button type="submit" >Да</button>
</form>
```

И представление для редактирования:

```
@model AspNetIdentityApp.Models.EditModel

@{
    ViewBag.Title = "Edit";
}

<h2>Редактировать данные профиля</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div>
        @Html.ValidationSummary(true, "")
        <div>
            <p>Year: @Html.EditorFor(model => model.Year)</p>
        </div>

        <div>
            <div>
                <input type="submit" value="Сохранить" />
            </div>
```

```
        </div>
    </div>
}
```

Добавление ролей в ASP.NET Identity

Сущность ролей в ASP.NET Identity представлена классом IdentityRole, который реализует интерфейс IRole. И мы можем продолжать использовать IdentityRole, но при необходимости также можем переопределить, добавив в него новые свойства. Итак, возьмем проект из прошлой темы и добавим в него функциональность управления ролями.

Для начала добавим в папку Models новый класс ApplicationRole:

```
using Microsoft.AspNet.Identity.EntityFramework;
public class ApplicationRole: IdentityRole
{
    public ApplicationRole(){}

    public string Description { get; set; }
}
```

Новый класс наследует весь функционал от IdentityRole плюс добавляет новое свойство `Description`, которое будет содержать описание роли.

Для управления ролями используется менеджер ролей RoleManager. Поэтому добавим в папку Models новый класс ApplicationRoleManager:

```
class ApplicationRoleManager : RoleManager<ApplicationRole>
{
    public ApplicationRoleManager(RoleStore<ApplicationRole> store)
                : base(store)
    {}
    public static ApplicationRoleManager Cre-
ate(IdentityFactoryOptions<ApplicationRoleManager> options,
                                            IOwinContext context)
    {
        return new ApplicationRoleManager(new

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

Опять де наследуем функционал от уже имеющегося класса RoleManager. Метод `Create` позволит классу приложения OWIN создавать экземпляры менеджера ролей для обработки каждого запроса, где идет обращение к хранилищу ролей RoleStore.

И теперь нам надо зарегистрировать менеджер ролей в классе приложения OWIN. Поэтому откроем файл *Startup.cs*, который по предыдущим те-

мам у нас находился в папке App_Start, и изменим его содержимое следующим образом:

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {

app.CreatePerOwinContext<ApplicationContext>(ApplicationContext.Create);

app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Creat
e);

        // регистрация менеджера ролей

app.CreatePerOwinContext<ApplicationRoleManager>(ApplicationRoleManager.Creat
e);

        app.UseCookieAuthentication(new CookieAuthenticationOptions
        {
            AuthenticationType = DefaultAuthentication-
Types.ApplicationCookie,
            LoginPath = new PathString("/Account/Login"),
        });
    }
}
```

Благодаря регистрации менеджер ролей будет использовать тот же контекст данных, что и менеджер пользователей.

Если мы создаем систему ролей в приложении после первого запуска приложения, то нам надо будет произвести миграцию, так как контекст данных у нас изменился в силу изменения системы ролей. Поэтому в Visual Studio в окне Package Manager Console введем команду: enable-migrations и нажмем Enter.

Теперь нам надо создать саму миграцию. Там же в консоли Package Manager Console введем команду:

PM> Add-Migration "DataMigration"

Visual Studio автоматически сгенерирует класс миграции:

```
public partial class DataMigration : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.AspNetRoles", "Description", c => c.String());
        AddColumn("dbo.AspNetRoles", "Discriminator", c => c.String(nullable:
false, maxLength: 128));
    }

    public override void Down()
    {
        DropColumn("dbo.AspNetRoles", "Discriminator");
        DropColumn("dbo.AspNetRoles", "Description");
```

```
    }
}
```

И чтобы выполнить миграцию, применим этот класс, набрав в той же консоли команду:

PM> Update-Database

Теперь создадим контроллер, который будет выполнять стандартные действия с ролями:

```
using AspNetIdentityApp.Models;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;

public class RolesController : Controller
{
    private ApplicationRoleManager RoleManager
    {
        get
        {
            return HttpCon-
text.GetOwinContext().GetUserManager<ApplicationRoleManager>();
        }
    }

    public ActionResult Index()
    {
        return View(RoleManager.Roles);
    }

    public ActionResult Create()
    {
        return View();
    }
    [HttpPost]
    public async Task<ActionResult> Create(CreateRoleModel model)
    {
        if (ModelState.IsValid)
        {
            IdentityResult result = await RoleManager.CreateAsync(new Applica-
tionRole
            {
                Name = model.Name,
                Description = model.Description
            });
            if (result.Succeeded)
            {
                return RedirectToAction("Index");
            }
            else
            {
                ModelState.AddModelError("", "Что-то пошло не так");
            }
        }
        return View(model);
```

```
    }

    public async Task<ActionResult> Edit(string id)
    {
        ApplicationRole role = await RoleManager.FindByIdAsync(id);
        if (role != null)
        {
            return View(new EditRoleModel { Id = role.Id,  Name = role.Name,
Description = role.Description });
        }
        return RedirectToAction("Index");
    }
    [HttpPost]
    public async Task<ActionResult> Edit(EditRoleModel model)
    {
        if (ModelState.IsValid)
        {
            ApplicationRole role = await RoleManager.FindByIdAsync(model.Id);
            if (role != null)
            {
                role.Description = model.Description;
                role.Name = model.Name;
                IdentityResult result = await RoleManager.UpdateAsync(role);
                if (result.Succeeded)
                {
                    return RedirectToAction("Index");
                }
                else
                {
                    ModelState.AddModelError("", "Что-то пошло не так");
                }
            }
        }
        return View(model);
    }

    public async Task<ActionResult> Delete(string id)
    {
        ApplicationRole role = await RoleManager.FindByIdAsync(id);
        if (role != null)
        {
            IdentityResult result = await RoleManager.DeleteAsync(role);
        }
        return RedirectToAction("Index");
    }
}
```

Это стандартный CRUD-контроллер, выполняющий чтение, удаление, редактирование и добавление ролей. Вначале для взаимодействия с менеджером ролей мы получаем его объект из контекста OWIN:

```
HttpContext.GetOwinContext().GetUserManager<ApplicationRoleManager>()
```

Затем методы менеджера ролей используются для управления ролями. Обратите внимание, что из представлений в методы Create и Edit мы получа-

ем не объект ApplicationRole, а специальные модели EditRoleModel и CreateRoleModel, который могут выглядеть так:

```
public class EditRoleModel
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
}
public class CreateRoleModel
{
    public string Name { get; set; }
    public string Description { get; set; }
}
```

Использование моделей позволит избежать различных проблем с контекстом данных и управлением объектами, которые могли возникнуть, если бы мы напрямую использовали бы ApplicationRole.

И представления будут выглядеть стандартно. Представление *Index.cshtml*:

```
@model IEnumerable<AspNetIdentityApp.Models.ApplicationRole>

@{
    ViewBag.Title = "Роли";
}

<h2>Роли</h2>
<table class="table">
    <tr>
        <th>Название</th>
        <th>Описание</th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>@item.Name</td>
        <td>@item.Description</td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.Id })
        </td>
    </tr>
}
</table>
```

Представление *Create.cshtml*:

```
@model AspNetIdentityApp.Models.CreateRoleModel

<h2>Добавление роли</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
```

```
    <div>
        @Html.ValidationSummary(true, "")
        <p>Название:  @Html.EditorFor(model => model.Name)</p>
        <p>Описание: @Html.EditorFor(model => model.Description)</p>
        <p><input type="submit" value="Добавить" /></p>
    </div>
}
```

И представление для редактирования *Edit.cshtml*:

```
@model AspNetIdentityApp.Models.EditRoleModel

<h2>Редактирование роли</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div>
        @Html.ValidationSummary(true, "")
        @Html.HiddenFor(model => model.Id)
        <p>Название:  @Html.EditorFor(model => model.Name)</p>
        <p>Описание: @Html.EditorFor(model => model.Description)</p>
        <p><input type="submit" value="Изменить" /></p>
    </div>
}
```