Авторизация и роли в Visual Studio

1 В предыдущей статье мы применили атрибут Authorize для класса контроллера Account, который ограничивает допуск к методам действий для неавторизованных пользователей. В этой статье я покажу вам, как доработать систему авторизации, чтобы осуществить более полный контроль над тем, какие действия можно выполнять определенным пользователям. По традиции, я составил список из трех вопросов, которые у вас сразу могут возникнуть:

Что это?

Авторизация – это процесс предоставления доступа к контроллерам и методам действий для определенных пользователей, как правило, находящихся в определенных ролях (например, допуск к админке должны иметь только администраторы).

Зачем нужно использовать?

Без авторизации вы сможете различать только две категории пользователей: прошедшие и не прошедшие аутентификацию. Большинство приложений имеют список ролей, таких как: пользователь, модератор, администратор и т. д.

Как использовать в рамках MVC?

Роли используются для реализации авторизации через атрибут Authorize, который применяется к контроллерам и методам действий.

Добавление поддержки ролей

ASP.NET Identity содержит строго типизированный базовый класс для доступа и управления ролями, который называется **RoleM** ей *интерфейса IRole*, описывающего механизм хранения данных, используемых для

представления ролей. Entity Framework использует **класс IdentityRole**, являющийся реализацией интерфейса IRole и содержит следующие свойства:

Свойства, определенные в классе IdentityRole

Название	Описание
Id	Уникальный идентификатор роли.
Name	Название роли./p>
Users	Возвращает список объектов IdentityUserRole, представляющих пользователей, которые находятся в данной роли.

Мы не будем использовать напрямую объекты IdentityRole в нашем приложении, вместо этого добавьте файл класса AppRole.cs в папку Models со следующим содержимым:

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

namespace Users.Models
{
    public class AppRole : IdentityRole
    {
        public AppRole() : base() { }

        public AppRole(string name)
            : base(name)
        { }
    }
}
```

Класс RoleManager<T> работает с экземплярами IRole с помощью методов и свойств, перечисленных в таблице ниже:

Пройди тесты C# тест (легкий) .NET тест (средний)

Свойства и методы, определенные в классе `RoleManager<T>`

Название	Описание
CreateAsync(role)	Создает новую роль
DeleteAsync(role)	Удаляет указанную роль
FindByIdAsync(id)	Поиск роли по идентификатору
FindByNameAsync(name)	Поиск роли по названию
RoleExistsAsync(name)	Возвращает `true`, если существует роль с указанным именем
UpdateAsync(role)	Сохраняет изменения в указанной роли
Roles	Список существующих ролей

Эти базовые методы реализуют тот же базовый шаблон, который использует класс `UserManager<T>` для управления пользователями. Добавьте файл `AppRoleManager.cs` в папку `Infrastructure` со следующим содержимым:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using System;
using Users.Models;

namespace Users.Infrastructure
{
    public class AppRoleManager : RoleManager<AppRole>, IDisposable
    {
        public AppRoleManager(RoleStore<AppRole> store)
            : base(store)
        { }

        public static AppRoleManager Create(
            IdentityFactoryOptions<AppRoleManager> options,
            IOwinContext context)
        {
            return new AppRoleManager(new
                RoleStore<AppRole>(context.Get<AppIdentityDbContext>()));
        }
    }
}
```

Этот класс определяет статический метод `Create()`, который позволит `OWIN` создавать экземпляры класса `AppRoleManager` для всех запросов, где требуются данные `Identity`, не раскрывая информации о том, как данные о ролях хранятся в приложении. Чтобы зарегистрировать класс управления ролями в `OWIN`, необходимо отредактировать файл `IdentityConfig.cs`, как показано в примере ниже:

Пройди тесты C# тест (легкий) .NET тест (средний)

```
// ...

namespace Users
{
    public class IdentityConfig
    {
        public void Configuration(IAppBuilder app)
        {
            // ...

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

            // ...
        }
    }
}
```

Это гарантирует, что экземпляры класса `AppRoleManager` используют тот же контекст базы данных `Entity Framework`, что и экземпляры `AppUserManager`.

Создание и удаление ролей

Мы подготовили базовую инфраструктуру для работы с ролями, давайте теперь создадим средство администрирования для работы с ролями. Сначала давайте определим методы действия и представления для управления ролями. Добавьте контроллер `RoleAdmin` в проект приложения с кодом, показанным в примере ниже:

```
using System.Web;
using System.Web.Mvc;
using System.Threading.Tasks;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using System.ComponentModel.DataAnnotations;
using Users.Infrastructure;
using Users.Models;

namespace Users.Controllers
{
    public class RoleAdminController : Controller
    {
        private AppUserManager UserManage
        {
            get
```

```
        {
            return HttpContext.GetOwinContext().GetUserManager<AppUserManager
        }
    }

    private AppRoleManager RoleManager
    {
        get
        {
            return HttpContext.GetOwinContext().GetUserManager<AppRoleManager
        }
    }

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

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

    [HttpPost]
    public async Task<ActionResult> Create([Required]string name)
    {
        if (ModelState.IsValid)
        {
            IdentityResult result
                = await RoleManager.CreateAsync(new AppRole(name));

            if (result.Succeeded)
            {
                return RedirectToAction("Index");
            }
            else
            {
                AddErrorsFromResult(result);
            }
        }
        return View(name);
    }

    [HttpPost]
```

```
        public async Task<ActionResult> Delete(string id)
        {
            AppRole role = await RoleManager.FindByIdAsync(id);
            if (role != null)
            {
                IdentityResult result = await RoleManager.DeleteAsync(role);
                if (result.Succeeded)
                {
                    return RedirectToAction("Index");
                }
                else
                {
                    return View("Error", result.Errors);
                }
            }
            else
            {
                return View("Error", new string[] { "Роль не найдена" });
            }
        }

        private void AddErrorsFromResult(IdentityResult result)
        {
            foreach (string error in result.Errors)
            {
                ModelState.AddModelError("", error);
            }
        }
    }
}
```

Здесь мы применили многие из тех приемов, что использовали в контроллере `Admin`, в том числе добавили свойства `UserManager` и `RoleManager` для более быстрого запроса объектов `AppRoleManager` и `AppUserManager`. Также мы добавили аналогичный метод `AddErrorsFromResult()`, который обрабатывает ошибки в объекте `IdentityResult` и добавляет их в метаданные модели.

Представления для контроллера `RoleAdmin` содержат простую `HTML`-разметку и операторы `Razor`. Нам необходимо отобразить не только список ролей, но и имена всех пользователей, входящих в каждую роль. Класс `IdentityRole` определяет свойство `Users`, которое возвращает коллекцию объектов **`IdentityUserRole`**, описывающих пользователей роли. Каждый о[illegible] во `UserId`, которое возвращает уникальный идентификатор пользователя, с помощью которого мы будем получать имя пользователя.

Добавьте файл класса IdentityHelpers.cs в папку Infrastructure со следующим содержимым:

```
using System.Web;
using System.Web.Mvc;
using Microsoft.AspNet.Identity.Owin;

namespace Users.Infrastructure
{
    public static class IdentityHelpers
    {
        public static MvcHtmlString GetUserName(this HtmlHelper html, string id)
        {
            AppUserManager mgr = HttpContext.Current
                .GetOwinContext().GetUserManager<AppUserManager>();

            return new MvcHtmlString(mgr.FindByIdAsync(id).Result.UserName);
        }
    }
}
```

Этот код содержит определение вспомогательного метода HTML, определенного как расширение класса HtmlHelper. Метод GetUserName() принимает строковый аргумент, содержащий идентификатор пользователя, получает экземпляр класса AppUserManager с помощью метода GetOwinContext().GetUserManager() (где метод GetOwinContext является расширяющим HttpContext), использует метод FindByIdAsync(), чтобы найти экземпляр AppUser, связанный с идентификатором и возвращает значение свойства UserName.

Следующий пример показывает содержимое файла Index.cshtml, находящегося в папке /Views/RoleAdmin:

```
@using Users.Models
@using Users.Infrastructure
@model IEnumerable<AppRole>

@{
    ViewBag.Title = "Роли";
}
```

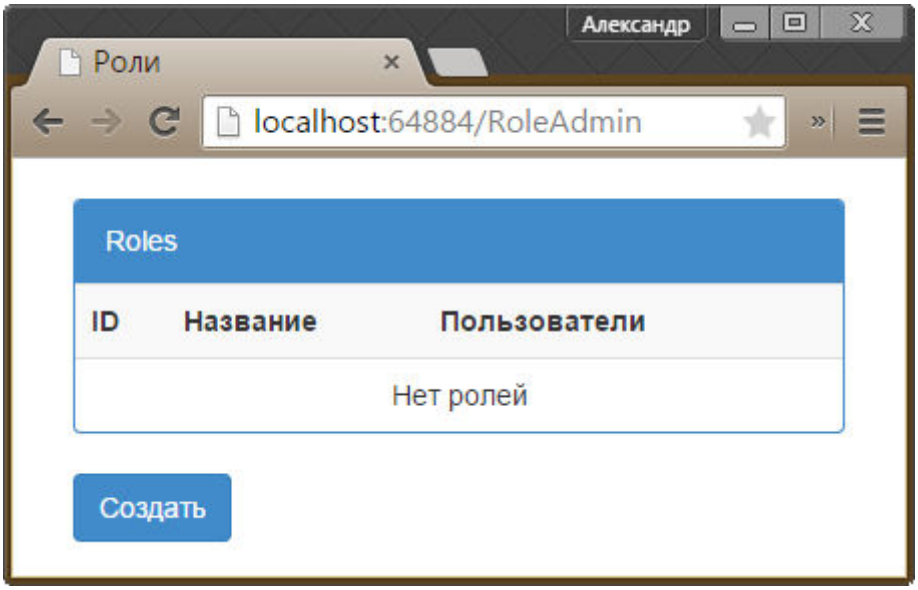


```
<div class="panel panel-primary">
    <div class="panel-heading">Roles</div>
```

```
    <table class="table table-striped">
        <tr>
            <th>ID</th>
            <th>Название</th>
            <th>Пользователи</th>
            <th style="min-width: 150px"></th>
        </tr>
        @if (Model.Count() == 0)
        {
        <tr>
            <td colspan="4" class="text-center">Нет ролей</td>
        </tr>
        }
        else
        {
            foreach (AppRole role in Model) {
            <tr>
                <td>@role.Id</td>
                <td>@role.Name</td>
                <td>
                    @if (role.Users == null || role.Users.Count == 0)
                    {
                        @: Нет пользователей в этой роли
                    }
                    else
                    {
                        <p>@string.Join(", ", role.Users.Select(x =>
                            Html.GetUserName(x.UserId)))
                        </p>
                    }
                </td>
                <td>
                @using (Html.BeginForm("Delete", "RoleAdmin", new { id = role.Id
                {
                    @Html.ActionLink("Изменить", "Edit", new { id = role.Id },
                        new { @class = "btn btn-primary btn-xs", style= "float:le
                    <button class="btn btn-danger btn-xs" type="submit">Удалить<
                }
                </td>
            </tr>
            }
        }
    </table>
</div>
```

```
@Html.ActionLink("Создать", "Create", null, new { @class = "btn btn-primary" })
```

В этом представлении отображается список ролей, определенных в приложении, вместе со списком пользователей в каждой роли. На данный момент мы еще не создали ни одной роли:



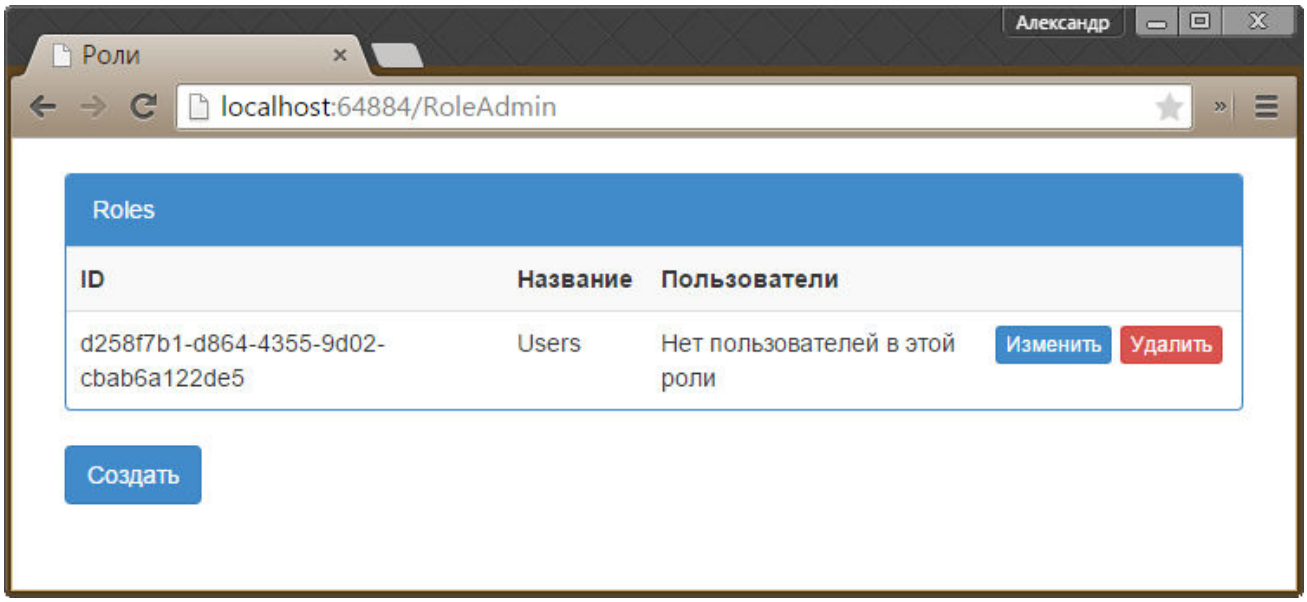
Следующий пример содержит представление `Create.cshtml` в той же папке, которое используется для создания новых ролей:

Пройди тесты C# тест (легкий) .NET тест (средний)

```
@model string
@{
    ViewBag.Title = "Создание роли";
}

<h2>Создать роль</h2>
@Html.ValidationSummary(false)
@using (Html.BeginForm())
{
    <div class="form-group">
        <label>Название</label>
        <input name="name" value="@Model" class="form-control" />
    </div>
    <button type="submit" class="btn btn-primary">Создать</button>
    @Html.ActionLink("Отмена", "Index", null, new { @class = "btn btn-default" }
}
```

Единственная информация, которая требуется для создания новой роли – ее название. Поэтому мы добавили один стандартный элемент `<input>` и кнопку отправки формы `POST`-методу действия `Create`.

Чтобы протестировать функционал создания ролей, запустите приложение и перейдите по адресу `/RoleAdmin/Index` в окне браузера. Чтобы создать новую роль нажмите кнопку «Создать», введите имя в поле ввода в появившейся форме и нажмите вторую кнопку «Создать». Новое представление будет отображать список ролей, сохраненных в базе данных:



Вы можете также удалить роль из приложения нажав кнопку «Удалить».

Редактирование ролей

Для авторизации пользователей недостаточно просто создавать и удалять роли. Мы также должны уметь управлять ролями, назначать и удалять пользователей из роли. Это не сложный процесс, но для его реализации нам необходимо загружать данные о ролях с помощью класса AppRoleManager, а затем вызывать методы, определенные в классе AppUserManager на объектах, связанных с определенной ролью.

Давайте начнем с добавления новых классов модели-представления (view-model) в файл UserViewModels.cs:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace Users.Models
{
    public class CreateModel
    {
        // ...
    }

    public class LoginViewModel
    {
        // ...
    }

    public class RoleEditModel
    {
        public AppRole Role { get; set; }
        public IEnumerable<AppUser> Members { get; set; }
        public IEnumerable<AppUser> NonMembers { get; set; }
    }

    public class RoleModificationModel
    {
        [Required]
        public string RoleName { get; set; }
        public string[] IdsToAdd { get; set; }
        public string[] IdsToDelete { get; set; }
    }
}
```


Класс RoleEditModel содержит информа и ок пользователей в роли в виде коллекции объектов AppUser. Благодаря этому, мы сможем извлечь ID и имя каждого пользователя в роли. Класс RoleModificationModel

будет получать данные от системы привязки модели во время редактирования данных пользователя. Он содержит массив идентификаторов пользователей, а не объектов AppUser, для замены ролей.

Определившись с классами моделей, давайте добавим методы редактирования ролей Edit в контроллер RoleAdmin:

```
using System.Web;
using System.Web.Mvc;
using System.Threading.Tasks;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using System.ComponentModel.DataAnnotations;
using Users.Infrastructure;
using Users.Models;
using System.Linq;
using System.Collections.Generic;

namespace Users.Controllers
{
    public class RoleAdminController : Controller
    {
        // ...

        public async Task<ActionResult> Edit(string id)
        {
            AppRole role = await RoleManager.FindByIdAsync(id);
            string[] memberIDs = role.Users.Select(x => x.UserId).ToArray();

            IEnumerable<AppUser> members
                = UserManager.Users.Where(x => memberIDs.Any(y => y == x.Id));

            IEnumerable<AppUser> nonMembers = UserManager.Users.Except(members);

            return View(new RoleEditModel
            {
                Role = role,
                Members = members,
                NonMembers = nonMembers
            });
        }
```



```
        [HttpPost]
        public async Task<ActionResult> Edit(RoleModificationModel model)
```

```
        public async Task<ActionResult> Edit(RoleModificationModel model)
        {
            IdentityResult result;
            if (ModelState.IsValid)
            {
                foreach (string userId in model.IdsToAdd ?? new string[] { })
                {
                    result = await UserManager.AddToRoleAsync(userId, model.RoleN

                    if (!result.Succeeded)
                    {
                        return View("Error", result.Errors);
                    }
                }
                foreach (string userId in model.IdsToDelete ?? new string[] { })
                {
                    result = await UserManager.RemoveFromRoleAsync(userId,
                    model.RoleName);

                    if (!result.Succeeded)
                    {
                        return View("Error", result.Errors);
                    }
                }
                return RedirectToAction("Index");

            }
            return View("Error", new string[] { "Роль не найдена" });
        }
    }
}
```

Большая часть кода в GET-версии метода Edit отвечает за формирование списков пользователей входящих и не входящих в роль и реализуется с помощью методов LINQ. После группировки пользователей возвращается представление, которому передается объект RoleEditModel.

POST-версия метода Edit отвечает за добавление и удаление пользователей из ролей. Класс AppUserManager наследует ряд вспомогательных методов для работы с ролями из класса UserManager<T>. Эти методы перечислены в таблице ниже:

Пройди тесты C# тест (легкий) .NET тест (средний)

Вспомогательные методы класса `UserManager<T>` для работы с ролями

Название	Описание
`AddToRoleAsync(id, name)`	Добавляет пользователя с указанным идентификатором `id` в роль с указанным именем `name`
`GetRolesAsync(id)`	Возвращает список из имен ролей, в которых находится пользователь с идентификатором `id`
`IsInRoleAsync(id, name)`	Вернет `true`, если пользователь с указанным идентификатором `id` является членом роли с именем `name`
`RemoveFromRoleAsync(id, name)`	Удаляет пользователя с указанным `id` из роли с указанным именем `name`

Странность этих методов заключается в том, что они работают с идентификатором пользователя и именем роли, хотя каждая роль также имеет свой уникальный идентификатор. Именно поэтому класс `RoleModificationModel` содержит строковое свойство `RoleName`.

В примере ниже показан код представления `Edit.cshtml`, находящегося в папке `/Views/RoleAdmin.cshtml`:

```
@using Users.Models
@model RoleEditModel
@{
    ViewBag.Title = "Изменить роль";
}

<h2>Изменить роль</h2>
@Html.ValidationSummary()
@using (Html.BeginForm())
{
    <input type="hidden" name="roleName" value="@Model.Role.Name" />
    <div class="panel panel-primary">
        <div class="panel-heading">Добавить в роль <b>@Model.Role.Name</b></div>
        <table class="table table-striped">
```

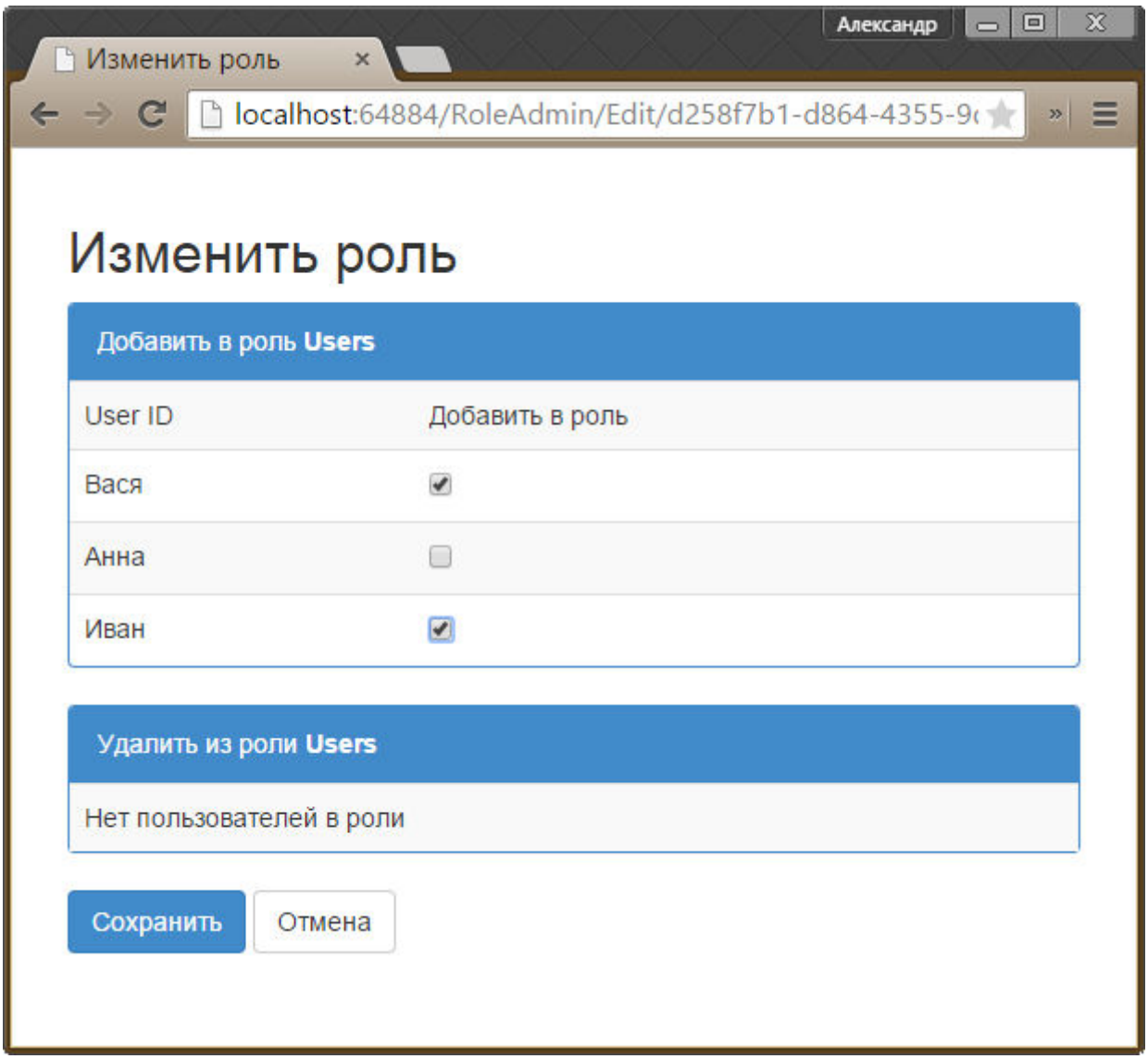
```
            @if (Model.NonMembers.Count() == 0)
            {
                <tr>
                    <td colspan="2">Все пользователи в роли</td>
                </tr>
            }
            else
            {
                <tr>
                    <td>User ID</td>
                    <td>Добавить в роль</td>
                </tr>
                foreach (AppUser user in Model.NonMembers) {
                <tr>
                    <td>@user.UserName</td>
                    <td>
                        <input type="checkbox" name="IdsToAdd" value="@user.Id">
                    </td>
                </tr>
                }
            }
        </table>
    </div>

    <div class="panel panel-primary">
        <div class="panel-heading">Удалить из роли <b>@Model.Role.Name</b></div>
        <table class="table table-striped">
            @if (Model.Members.Count() == 0)
            {
                <tr>
                    <td colspan="2">Нет пользователей в роли</td>
                </tr>
            } else {
                <tr>
                    <td>User ID</td>
                    <td>Удалить из роли</td>
                </tr>
                foreach (AppUser user in Model.Members)
                {
                <tr>
                    <td>@user.UserName</td>
                    <td>
                        <input type="checkbox" name="IdsToDelete" value="@user.Id
                    </td>
```

```
                </tr>
                }
            }
        </table>
    </div>
    <button type="submit" class="btn btn-primary">Сохранить</button>
    @Html.ActionLink("Отмена", "Index", null, new { @class = "btn btn-default" }
}
```

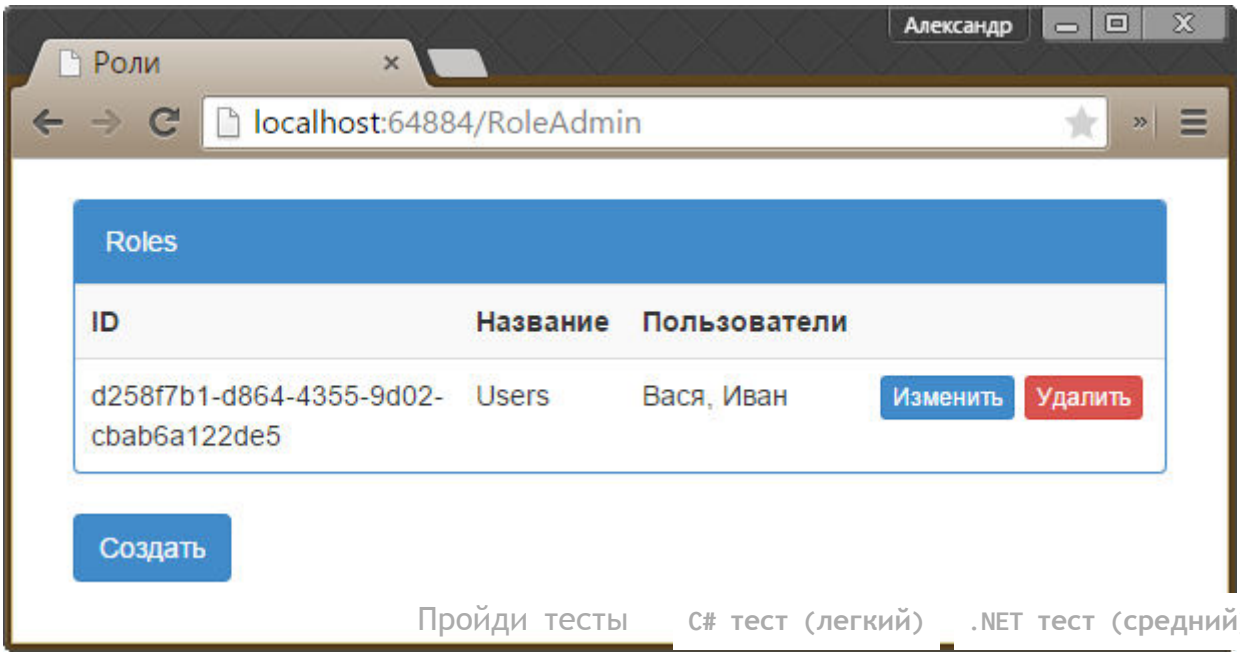
Представление содержит две таблицы: со списком пользователей входящих и не входящих в текущую роль, открытую для редактирования. Напротив каждого пользователя стоит флажок, который позволяет изменить его членство в данной роли.

Давайте протестируем функциональность редактирования ролей. Добавление класса `AppRoleManager` в архитектуру `OWIN` заставит `Entity Framework` удалить базу данных и воссоздать новую схему. Это означает, что пользователи, которых мы создали ранее исчезнут. Поэтому после запуска приложения перейдите по адресу `/Admin/Index` и создайте нескольких пользователей.

Чтобы проверить редактирование ролей, перейдите по адресу `/RoleAdmin/Index` и создайте несколько ролей, затем отредактируйте эти роли, добавив в них нескольких пользователей. На рисунке ниже показан пример приложения (я создал роль `Users`):



Нажмите на кнопке сохранить и перейдите в представление `/RoleAdmin`. Вы увидите список созданных ролей и список пользователей в каждой роли, как показано на рисунке ниже:



Пройди тесты C# тест (легкий) .NET тест (средний)

Использование ролей для авторизации

Теперь, когда у нас есть возможность управления ролями, мы можем использовать их в качестве основы для авторизации через атрибут Authorize. Чтобы проще было тестировать процесс авторизации, давайте добавим метод действия для выхода пользователя из системы в контроллер Account, как показано в примере ниже:

```
[Authorize]
public class AccountController : Controller
{
    // ...

    [Authorize]
    public ActionResult Logout()
    {
        AuthManager.SignOut();
        return RedirectToAction("Index", "Home");
    }

        // ...
}
```

Давайте обновим контроллер Home и добавим новый метод действия, который будет передавать информацию об аутентифицированном пользователе в представление:

Пройди тесты C# тест (легкий) .NET тест (средний)

```
using System.Collections.Generic;
using System.Web.Mvc;

namespace Users.Controllers
{
    public class HomeController : Controller
    {
        [Authorize]
        public ActionResult Index()
        {
            return View(GetData("Index"));
        }

        [Authorize(Roles = "Users")]
        public ActionResult OtherAction()
        {
            return View("Index", GetData("OtherAction"));
        }

        private Dictionary<string, object> GetData(string actionName)
        {
            Dictionary<string, object> dict = new Dictionary<string, object>();

            dict.Add("Action", actionName);
            dict.Add("Пользователь", HttpContext.User.Identity.Name);
            dict.Add("Аутентифицирован?", HttpContext.User.Identity.IsAuthentic
            dict.Add("Тип аутентификации", HttpContext.User.Identity.Authentica
            dict.Add("В роли Users?", HttpContext.User.IsInRole("Users"));

            return dict;
        }
    }
}
```

В этом примере мы оставили атрибут `Authorize` для метода действия `Index` без изменений, но добавили этот атрибут к методу `OtherAction`, задав при этом свойство `Roles`, ограничивающее доступ к этому методу только для пользователей, являющихся членами роли `Users`. Мы также добавили метод `GetData()`, который добавляет некоторую базовую информацию о пользователе, используя свойства, доступные через объект `HttpContext`.

В заключение, нам необходимо добави д в представление `Index.cshtml` из папки `/Views/Home`:

```
...

@Html.ActionLink("Выйти", "Logout", "Account", null, new {@class = "btn btn-prim
```

Атрибут `Authorize` может быть также использован для настройки авторизации на основе списка пользователей. Данную возможность удобно использовать в небольших проектах, но это создаст трудности при расширении приложения, т. к. каждый раз потребуется изменять код в контроллере, когда будет добавляется новый пользователь. Использование ролей для авторизации изолирует приложение от изменений в учетных записях отдельных пользователей и контролирует доступ к приложению через членство ролей.

Для тестирования системы авторизации, запустите приложение и перейдите по адресу `/Home/Index`. Браузер будет перенаправлен на страницу входа в приложение, где вы должны будете ввести данные существующей учетной записи. Метод действия `Index` является доступным для любого авторизованного пользователя. Однако если вы перейдете по адресу `/Index/OtherAction`, доступ будет открыт только тем пользователям, которые являются членами роли `Users`.

Если вы попытаетесь войти под пользователем, находящимся в другой роли, то браузер перенаправит вас снова на форму входа в приложение. Перенаправление уже аутентифицированных пользователей на страницу входа является малополезным решением, поэтому давайте отредактируем контроллер `Account` и добавим возможность перенаправления аутентифированных пользователей, не прошедших авторизацию, на страницу ошибки:

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login(string returnUrl)
    {
        if (HttpContext.User.Identity.IsAuthenticated)
        {
            return View("Error", new string[] { "В доступе отказано" });
        }

        ViewBag.returnUrl = returnUrl;
        return View();
    }

    // ...
}
```

На рисунке ниже наглядно показано поведение нашего приложения, когда пользователю отказано в доступе: