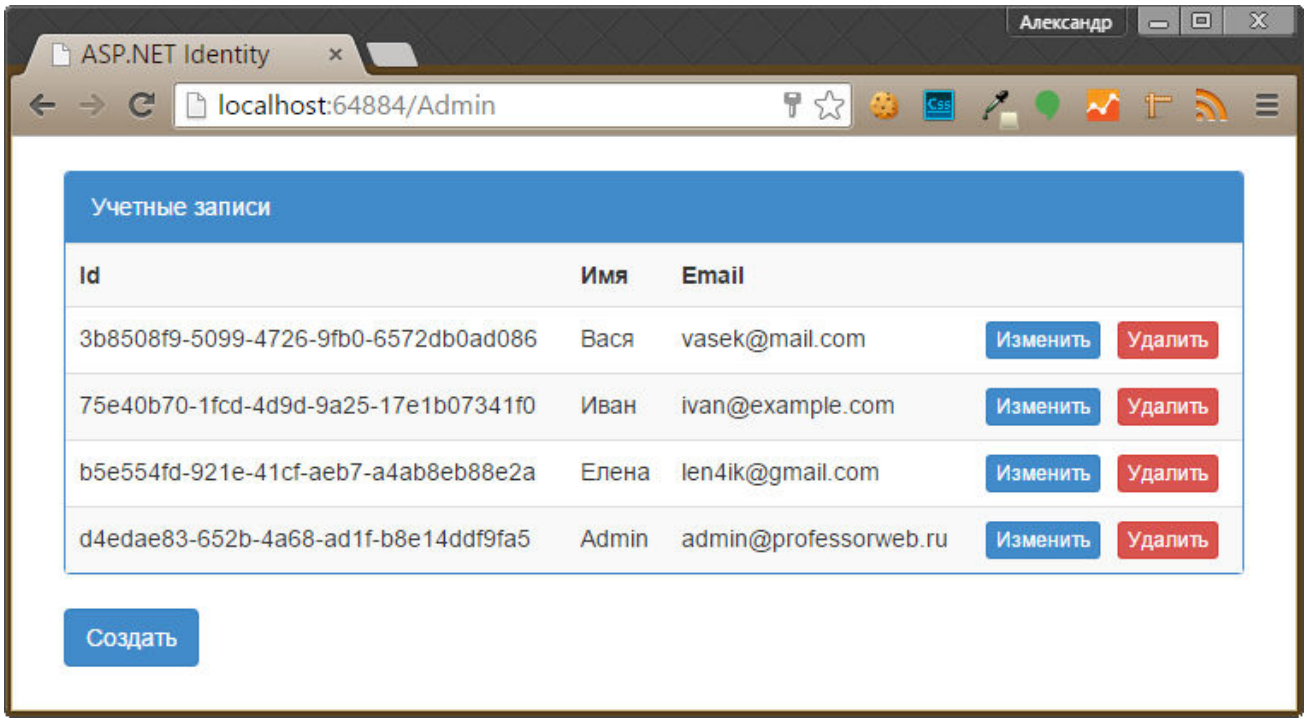
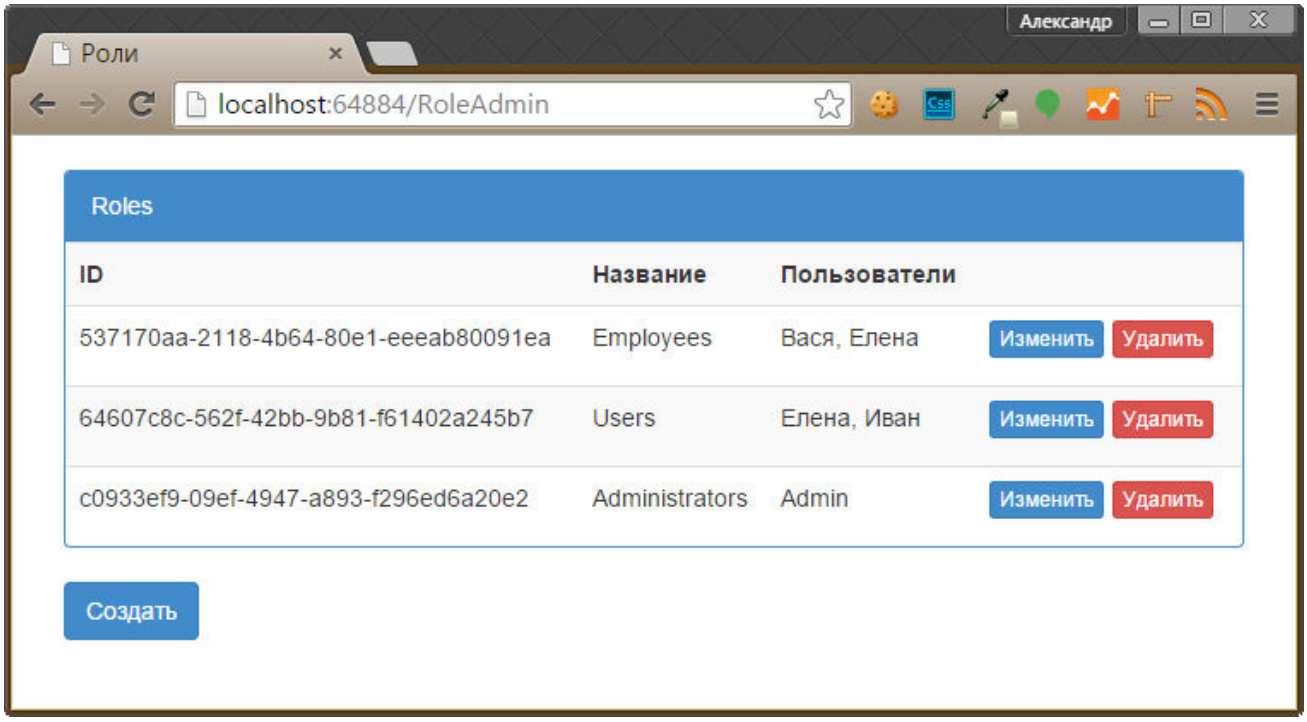
Свойства пользователей в ASP.NET Identity

1 В этой статье мы продолжим работу над тестовым проектом, который создавали и проектировали в предыдущих статьях. Никаких дополнительных изменений вносить в структуру проекта не требуется, но нужно добавить следующих пользователей, используя панель администратора:



На рисунке ниже показан список созданных ролей и список пользователей, входящих в каждую роль. Обратите внимание, что пользователь «Елена» входит сразу в две роли - Users и Employees:



Добавление свойств пользователя

Когда ранее мы создали класс `AppUser` я отметил, что базовый класс `IdentityUser` содержит несколько свойств для описания пользователя, таких как адрес электронной почты и имя. Большинству приложений необходимо хранить больше данных о пользователе, например, индивидуальные настройки приложения, ссылку на аватарку и т. д. - все любые данные, которые пригодятся для выполнения приложения и которые должны сохраняться между сессиями. В старой платформе `ASP.NET Membership` эта возможность реализована через профили пользователей, но `ASP.NET Identity` использует другой подход.

Т.к. `ASP.NET Identity` использует `Entity Framework` для работы с данными пользователя, все что нам нужно — это добавить необходимые свойства в класс пользователя. `Code-First` воссоздаст схему базы данных на основе класса пользователя, добавив новые столбцы в таблицу `Users`.

В следующем примере мы добавили свойство, описывающее город, в котором проживает пользователь:

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

```
using System;
using Microsoft.AspNet.Identity.EntityFramework;
using System.ComponentModel.DataAnnotations;

namespace Users.Models
{
    public enum Cities
    {
        [Display(Name = "Лондон")]
        LONDON,

        [Display(Name = "Париж")]
        PARIS,

        [Display(Name = "Москва")]
        MOSCOW
    }

    public class AppUser : IdentityUser
    {
        public Cities City { get; set; }
    }
}
```

В этом примере мы определили перечисление `Cities` и добавили свойство `City` в класс модели пользователя `AppUser`. Чтобы пользователи могли просматривать и редактировать свое местоположение, давайте добавим несколько методов действий в контроллер `Home`:

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

```
namespace Users.Controllers
{
    public class HomeController : Controller
    {
        // ...

        [Authorize]
        public ActionResult UserProps()
        {
            return View(CurrentUser);
        }

        [Authorize]
        [HttpPost]
        public async Task<ActionResult> UserProps(Cities city)
        {
            AppUser user = CurrentUser;
            user.City = city;
            await UserManager.UpdateAsync(user);
            return View(user);
        }

        private AppUser CurrentUser
        {
            get
            {
                return UserManager.FindByName(HttpContext.User.Identity.Name);
            }
        }

        private AppUserManager UserManager
        {
            get
            {
                return HttpContext.GetOwinContext().GetUserManager<AppUserManager
            }
        }

        // Вспомогательный метод, загружающий название элемента перечисл
        // из атрибута Display
        [NonAction]
        public static string GetCityName<TEnum>(TEnum item)
            where TEnum : struct, IConvertible
        {
```

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

```
        {
            if (!typeof(TEnum).IsEnum)
            {
                throw new ArgumentException("Тип TEnum должен быть перечислени
            }
            else
                return item.GetType()
                    .GetMember(item.ToString())
                    .First()
                    .GetCustomAttribute<DisplayAttribute>()
                    .Name;
        }
    }
}
```

Мы добавили свойство `CurrentUser`, которое возвращает экземпляр класса текущего пользователя `AppUser`, используя класс управления `AppUserManager`. Мы передаем объект `AppUser` в представление в `GET`-версии метода `UserProps`. `POST`-версия этого метода используется для изменения данных о местоположении пользователя.

Ниже показан код представления `UserProps.cshtml`, которое отображает текущее местоположение пользователя и позволяет его редактировать, используя выпадающий список со стандартными значениями:

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

```
@using Users.Controllers
@using System.Linq
@using Users.Models
@model AppUser

@{ ViewBag.Title = "Пользовательские свойства"; }

<div class="panel panel-primary">
    <div class="panel-heading">
        Пользовательские свойства
    </div>
    <table class="table table-striped">
        <tr><th>Текущий город</th><td>@HomeController.GetCityName(Model.City)</
    </table>
</div>

@using (Html.BeginForm())
{
    <div class="form-group">
        <label>Город: </label>
        @Html.DropDownListFor(x => x.City, new SelectList(
                Enum.GetValues(typeof(Cities))
                .OfType<Cities>()
                  .Select(c =>
                  {
                      return new
                      {
                          Id = c,
                          Text = HomeController.GetCityName(c)
                      };
                  }),
                "Id", "Text"
        ))
    </div>
    <button class="btn btn-primary" type="submit">Сохранить</button>
}
```

Внимание, не запускайте приложение в данный момент! Т.к. мы изменили класс модели данных пользователя, Entity Framework обнаружит эти изменения и воссоздаст базу данных с новой структурой, удалив все В ах мы рассмотрим процесс сохранения данных при изменении классов модели.

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

Подготовка к миграции базы данных

Как было сказано выше, стандартным поведением Entity Framework является воссоздание базы данных при любых изменениях в классах модели, влияющих на схему базы данных. Удаление данных во время процесса разработки обычно не является проблемой, но в производственной среде такое поведение является катастрофическим. В этом разделе я покажу как использовать миграцию данных Entity Framework, которая обновляет схему данных Code First без их удаления.

Во-первых, нам необходимо выполнить следующую команду в панели Package Manager Console среды Visual Studio:

```
Enable-Migrations -EnableAutomaticMigrations
```

Эта команда добавит поддержку миграций базы данных и создаст папку Migrations в проекте, содержащую файл Configuration.cs со следующим содержимым:

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

```
namespace Users.Migrations
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration
            : DbMigrationsConfiguration<Users.Infrastructure.AppIdentityDbContext
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
            ContextKey = "Users.Infrastructure.AppIdentityDbContext";
        }

        protected override void Seed(Users.Infrastructure.AppIdentityDbContext co
        {
            //  This method will be called after migrating to the latest version.

            //  You can use the DbSet<T>.AddOrUpdate() helper extension method
            //  to avoid creating duplicate seed data. E.g.
            //
            //    context.People.AddOrUpdate(
            //      p => p.FullName,
            //      new Person { FullName = "Andrew Peters" },
            //      new Person { FullName = "Brice Lambson" },
            //      new Person { FullName = "Rowan Miller" }
            //    );
            //
        }
    }
}
```

Этот класс будет использоваться для переноса существующих данных в базе данных в новую схему. *Метод* `Seed()` используется для обновления существующих записей базы данных. В примере ниже показано использование этого метода для установки значений по умолчанию, в том числе и для нового свойства `City`:

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

```
namespace Users.Migrations
{
```

```
using Microsoft.AspNet.Identity.EntityFramework;
using System;
using System.Data.Entity;
using System.Data.Entity.Migrations;
using System.Linq;
using Users.Infrastructure;
using Users.Models;
using System.Data.Entity.Migrations;
using Microsoft.AspNet.Identity;

internal sealed class Configuration : DbMigrationsConfiguration<Users.Infrast
{
    public Configuration()
    {
        AutomaticMigrationsEnabled = true;
        ContextKey = "Users.Infrastructure.AppIdentityDbContext";
    }

    protected override void Seed(Users.Infrastructure.AppIdentityDbContext co
    {
        AppUserManager userMgr = new AppUserManager(new UserStore<AppUser>(co
        AppRoleManager roleMgr = new AppRoleManager(new RoleStore<AppRole>(co

        string roleName = "Administrators";
        string userName = "Admin";
        string password = "mypassword";
        string email = "admin@professorweb.ru";

        if (!roleMgr.RoleExists(roleName))
        {
            roleMgr.Create(new AppRole(roleName));
        }

        AppUser user = userMgr.FindByName(userName);
        if (user == null)
        {
            userMgr.Create(new AppUser { UserName = userName, Email = email }
                password);
            user = userMgr.FindByName(userName);
        }

        if (!userMgr.IsInRole(user.Id,
        {
            userMgr.AddToRole(user.Id, roleName);
```

```
            }

            foreach (AppUser dbUser in userMgr.Users)
            {
                dbUser.City = Cities.MOSCOW;
            }

            context.SaveChanges();
        }
    }
}
```

Обратите внимание, что большая часть кода в этом примере скопирована из класса `IdentityDbInit`, который мы определили ранее для заполнения базы данных значениями по умолчанию (создание роли администратора и админа). Чуть позже мы обновим этот класс (с добавлением миграций нам не нужен будет этот функционал). Помимо создания администратора, мы инициализировали свойство `City` значением по умолчанию:

```
// ...
foreach (AppUser dbUser in userMgr.Users)
{
    dbUser.City = Cities.MOSCOW;
}
// ...
```

Вы не обязаны устанавливать значения по умолчанию для всех новых свойств — я просто хотел показать, как можно обновить все существующие данные пользователей, используя метод `Seed()` класса `Configuration`. Будьте осторожны при установке таких значений в реальных приложениях, т. к. это приведет к обновлению всех данных пользователей при последующих изменениях класса модели.

Изменение класса контекста базы данных

Как я сказал выше, нам необходимо изменить класс инициализации базы данных `IdentityDbInit`. Сейчас он унаследован от класса **`DropCreateDatabaseIfModelChanges`**, который, как вы уже догадались по названию, удаляет и воссоздает базу данных при изменении схемы классов модели `Entity Framework`. В примере ниже мы изменили базовый класс для `IdentityDbInit`, чтобы он не влиял на базу данных:

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

```
// ...

namespace Users.Infrastructure
{
    public class AppIdentityDbContext : IdentityDbContext<AppUser>
    {
        // ...
    }

    public class IdentityDbInit : NullDatabaseInitializer<AppIdentityDbContext>
    { }
}
```

Мы удалили все методы, определенные ранее, а также изменили базовый класс на **NullDatabaseInitializer**, который игнорирует любые изменения в классах модели данных.

Выполнение миграции

Все что нам остается сделать — это применить миграции. Во-первых выполните следующую команду в панели Package Manager Console:

```
Add-Migration CityProperty
```

Это команда создаст новую миграцию с названием CityProperty (мне нравится указывать в названии миграции изменения, которые она затрагивает — в данном случает мы добавили свойство City). В папку Migrations будет добавлен новый файл класса, с названием, отражающим время, когда была выполнена миграция. Например, мой файл называется 201503262244036_CityProperty.cs. Этот файл будет содержать данные для Entity Framework, описывающие изменения в схеме базы данных:

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

```
namespace Users.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class CityProperty : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.AspNetUsers", "City", c => c.Int(nullable: false));
        }

        public override void Down()
        {
            DropColumn("dbo.AspNetUsers", "City");
        }
    }
}
```

Метод `Up()` описывает изменения, которые должны быть внесены в схему, когда база данных обновляется. В данном случае мы описали добавление столбца `City` в таблицу `AppNetUsers`, в которой сохранятся данные пользователей `ASP.NET Identity`.

Теперь необходимо выполнить миграцию. Без запуска приложения выполните следующую команду в панели `Package Manager Console`:

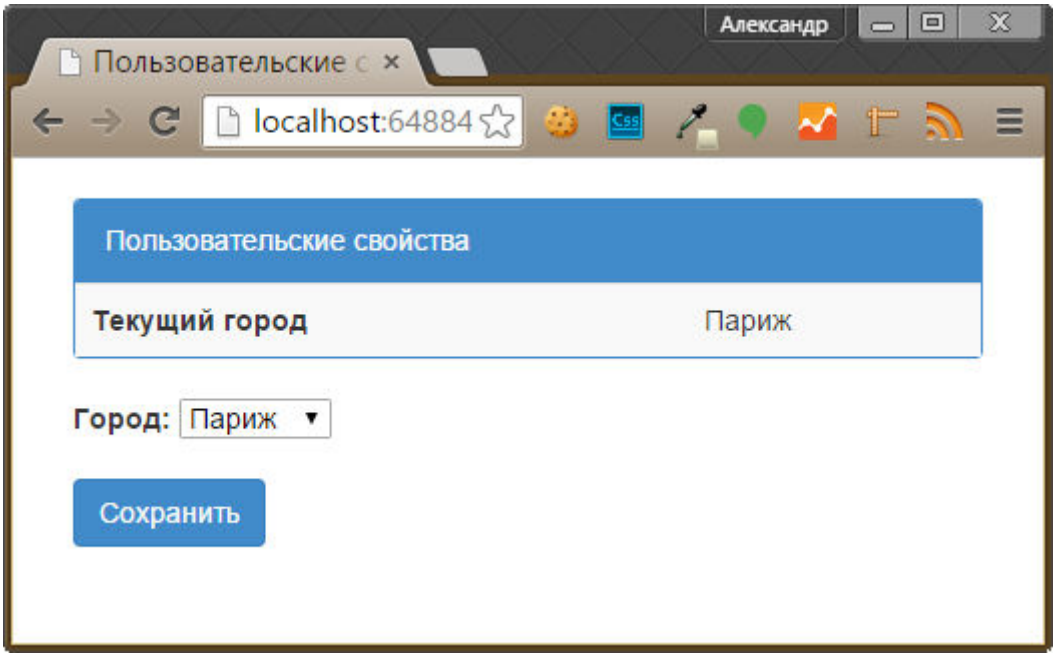
```
Update-Database -TargetMigration CityProperty
```

Схема базы данных будет изменена и код метода `Configuration.Seed()` будет выполнен. Существующие учетные записи пользователей будут сохранены и дополнятся новым столбцом `City`.

Тестирование миграции

Для тестирования миграции запустите приложение, перейдите по адресу `/Home/UserProps` и пройдите аутентификацию. После авторизации вы увидите текущее значение свойства `City` для пользователя и сможете его отредактировать:

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



Определение дополнительного свойства

Теперь, когда миграции базы данных настроены, мы определим еще одно пользовательское свойство для демонстрации обработки последующих изменений и покажем более полезный (и менее опасный) пример использования метода `Configuration.Seed()`. Давайте определим свойство `Country` в классе `AppUser`, как показано в примере ниже:

```
using System;
using Microsoft.AspNet.Identity.EntityFramework;
using System.ComponentModel.DataAnnotations;

namespace Users.Models
{
    public enum Cities
    {
        // ...
    }

    public enum Countries
    {
        [Display(Name = "Не указано")]
        NONE,

        [Display(Name = "Англия")]
        ENG,

        [Display(Name = "Франция")]
```

```
        FRA,

        [Display(Name = "Россия")]
        RUS
    }

    public class AppUser : IdentityUser
    {
        public Cities City { get; set; }
        public Countries Country { get; set; }

        public void SetCountryFromCity(Cities city)
        {
            switch (city)
            {
                case Cities.LONDON:
                    Country = Countries.ENG;
                    break;
                case Cities.PARIS:
                    Country = Countries.FRA;
                    break;
                case Cities.MOSCOW:
                    Country = Countries.RUS;
                    break;
                default:
                    Country = Countries.NONE;
                    break;
            }
        }
    }
}
```

Мы добавили перечисление для определения названий стран и вспомогательный метод, который выбирает значение страны на основе города. Следующий код содержит изменения, которые мы должны внести в класс `Configuration` - метод `Seed()` выбирает страну для свойства `Country` на основе свойства `City`, но только если значение `Country` содержит значение `None`:

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

```
namespace Users.Migrations
{
    // ...

    internal sealed class Configuration : DbMigrationsConfiguration<Users.Infrast
    {
        public Configuration()
        {
            // ...
        }

        protected override void Seed(Users.Infrastructure.AppIdentityDbContext co
        {
            // ...

            foreach (AppUser dbUser in userMgr.Users)
            {
                if (dbUser.Country == Countries.NONE)
                    dbUser.SetCountryFromCity(dbUser.City);
            }

            context.SaveChanges();
        }
    }
}
```

Такая установка свойства является полезной в реальном проекте, т. к. не будет обновлять свойства для объектов, у которых уже была задана ранее страна — т. е. последующие миграции не будут затрагивать это свойство.

Нет смысла определять дополнительные свойства пользователя, если они не доступны в приложении, поэтому давайте добавим отображение страны в представлении `UserProps.cshtml`:

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

```
...
<div class="panel panel-primary">
    ...
    <table class="table table-striped">
        <tr><th>Текущий город</th><td>@HomeController.GetCityName(Model.City)</
        <tr><th>Страна</th><td>@HomeController.GetCityName(Model.Country)</td></
    </table>
</div>
...
```

В следующем примере мы изменили метод действия UserProps контроллера Home и добавили инициализацию свойства Country через метод SetCountryFromCity():

```
public class HomeController : Controller
{
     // ...

    [Authorize]
    [HttpPost]
    public async Task<ActionResult> UserProps(Cities city)
    {
        AppUser user = CurrentUser;
        user.City = city;

        user.SetCountryFromCity(city);

        await UserManager.UpdateAsync(user);
        return View(user);
    }

        // ...
}
```

Давайте теперь запустим процесс миграции, по аналогии с запуском миграции при добавлении свойства City. Выполните следующую команду в панели Package Manager Console:

```
Add-Migration CountryProperty
```

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