

[C# и .NET](#)[Web](#)[Форум](#)[C# 5.0 и .NET 4.5](#)[WPF](#)[ТЕМЫ WPF](#)[SILVERLIGHT 5](#)[РАБОТА С БД](#)[LINQ](#)[ASP.NET](#)[WINDOWS 8/10](#)[ПРОГРАММЫ](#)

Использование ASP.NET Identity

[ASP.NET](#) --- [ASP.NET Identity](#) --- [Использование ASP.NET Identity](#)

1 Теперь, когда мы произвели базовые настройки системы ASP.NET Identity в нашем приложении, мы можем начать использовать эту платформу для управления доступом пользователей в приложение. В этой статье я покажу, как можно использовать Identity для создания инструментов, которые позволят централизованно управлять пользователями. Следующая таблица содержит ответы на три вопроса, которые могут волновать вас на данном этапе изучения ASP.NET Identity:

[Пройди тесты](#)[C# тест \(легкий\)](#)[.NET тест \(средний\)](#)

Вопросы по ASP.NET Identity

Вопрос	Ответ
Что это?	ASP.NET Identity - это API-интерфейс, используемый в веб-приложениях для управления данными пользователей, выполнения проверки подлинности и авторизации.
Почему это должно меня беспокоить?	Большинство приложений требуют, чтобы пользователи создавали учетные записи и получали определенные учетные данные для доступа к контенту и услугам. ASP.NET Identity предоставляет средства для выполнения этих операций.
Как Identity используется в рамках MVC Framework?	ASP.NET Identity не используется непосредственно в рамках MVC, но интегрируется через стандартные функции авторизации MVC.

Список учетных записей пользователей

Панель администрирования пользователей является полезной в большинстве приложений, даже в тех, где допускается пользователям самим управлять и настраивать свой аккаунт. Средства администрирования полезны для демонстрации возможностей Identity, т.к. они включают большой функционал управления пользователями в небольшое количество классов.

Итак, добавьте контроллер Admin в проект, код которого приведен в примере ниже. В дальнейшем в этом контроллере мы будем определять все функции администрирования пользователей.

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

namespace Users.Controllers
{
    public class AdminController : Controller
    {
        public ActionResult Index()
        {
            return View(UserManager.Users);
        }

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

Метод действия `Index` перечисляет всех пользователей, находящихся под управлением `Identity`, хотя на данный момент мы еще не создали ни одного пользователя. Важной частью этого примера является способ, с помощью которого мы получаем экземпляр класса `AppUserManager`, через который будем управлять информацией пользователей. В дальнейшем мы будем повторно использовать экземпляр класса `AppUserManager` в контроллере `Admin`, поэтому я создал отдельное свойство только для чтения `UserManager`, чтобы позже упростить код.

Сборка `Microsoft.Owin.Host.SystemWeb` добавляет вспомогательные методы расширения для класса `HttpContext`, одним из которых является `GetOwinContext()`, возвращающий объект контекста OWIN API для запроса (напомню, объект контекста реализует интерфейс `IOwinContext`). Сами по себе объекты `IOwinContext` не очень интересны в приложениях MVC, но они определяют другой метод расширения `GetUserManager<T>`, возвращающий экземпляр класса `UserManager`, используемый для управления пользователями.

В примере выше я вызвал метод `GetUserManager()`, передав в качестве обобщенного типа класс `AppUserManager`, который мы создали в предыдущей статье:

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

```
// ...  
return HttpContext.GetOwinContext().GetUserManager<AppUserManager>();  
// ...
```

После того, как у нас появится экземпляр класса `AppUserManager`, мы можем начать создавать запросы к хранилищу данных. Свойство `AppUserManager.Users` возвращает перечисление объектов—экземпляров пользовательского класса `AppUser`, с которым можно работать используя LINQ.

В методе действия `Index` мы передаем значение свойства `Users` в метод `View()`, благодаря чему, в представлении можно будет использовать список пользователей. Следующий пример показывает содержимое файла представления `Views/Admin/Index.cshtml`, который вы можете создать щелкнув правой кнопкой мыши по имени метода действия в контроллере и выбрав команду `Add View` из контекстного меню.

```

@using Users.Models
@model IEnumerable<AppUser>

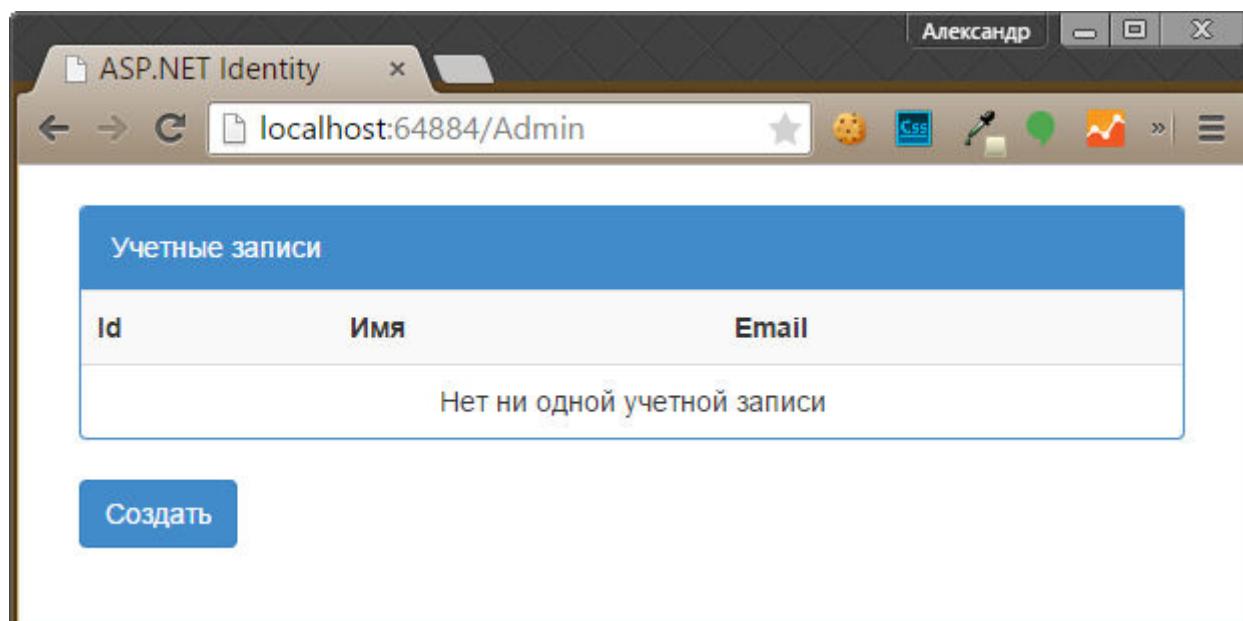
@{
    ViewBag.Title = "ASP.NET Identity";
}

<div class="panel panel-primary">
    <div class="panel-heading">
        Учетные записи
    </div>
    <table class="table table-striped">
        <tr>
            <th>Id</th>
            <th>Имя</th>
            <th>Email</th>
        </tr>
        @if (Model.Count() == 0) {
            <tr>
                <td colspan="3" class="text-center">Нет ни одной учетной записи
            </tr>
        } else {
            foreach (AppUser user in Model)
            {
                <tr>
                    <td>@user.Id</td>
                    <td>@user.UserName</td>
                    <td>@user.Email</td>
                </tr>
            }
        }
    </table>
</div>
@Html.ActionLink("Создать", "Create", null, new { @class = "btn btn-primary" })

```

Это представление содержит таблицу, содержащую строки для всех пользователей и имеющую три столбца с идентификатором, именем и адресом электронной почты пользователя. Если в базе данных еще нет пользователей, то отобразится сообщение, показанное на рисунке ниже:

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



Мы добавили кнопку «Создать» в это представление (стилизованную с помощью bootstrap), которая будет отправлять команду методу действия Create контроллера Admin. Позже мы добавим этот метод действия для добавления пользователей.

Сброс базы данных

Когда вы запустите приложение и перейдете по адресу /Admin/Index, то увидите заметную задержку в несколько секунд перед откликом приложения. Это происходит потому, что Entity Framework в первый раз подключается к базе данных и определяет, что еще нет определенной схемы данных. Code-First использует определенные нами ранее классы (и еще некоторые стандартные классы Identity) для создания схемы базы данных.

Эффект можно увидеть если вы откроете в Visual Studio окно Server Explorer и настроите подключение к базе данных IdentityDB. Эта база данных будет содержать таблицы AspNetUsers и AspNetRoles.

Чтобы сбросить базу данных вы можете щелкнуть по ней правой кнопкой мыши в приложении Microsoft SQL Server Managment Studio и выбрать команду Delete. Для создания базы данных щелкните правой кнопкой мыши по узлу Data Connections в окне Server Explorer и выберите в контекстном меню команду Create New SQL Server Database. В открывшемся модальном окне задайте название для базы IdentityDB.

Теперь, когда вы снова запросите страницу /Admin/Index, Identity опять обнаружит что в база данных пустая и воссоздаст схему данных. Этот прием сброса базы данных может быть полезен на этапе тестирования возможностей Identity и позже мы будем использовать его не раз.

Создание пользователей

В последующих примерах я буду использовать основы валидации модели данных MVC, поэтому для каждого действия в контроллере мы будем создавать **класс модели-представления** (view-model). Добавьте файл класса UserViewModels.cs в папку Models проекта. В примере ниже показано начальное содержимое этого файла (позже мы будем добавлять в него другие классы).
Продолжение: [Продолжение](#)

```

using System.ComponentModel.DataAnnotations;

namespace Users.Models
{
    public class CreateModel
    {
        [Required]
        public string Name { get; set; }

        [Required]
        public string Email { get; set; }

        [Required]
        public string Password { get; set; }
    }
}

```

Первый класс view-model называется CreateModel. Он определяет основные свойства, которые нам понадобятся для создания учетной записи пользователя: имя, адрес электронной почты и пароль. В этом примере используется **атрибут Required** из пространства имен System.ComponentModel.DataAnnotations, который указывает системе валидации модели на то, что эти свойства обязательно должны быть заданы.

Теперь давайте добавим пару методов действий Create в контроллер Admin, которые будут вызываться из представлений. Первый метод будет срабатывать на GET-запросы и возвращать представление для создания пользователя, второй — срабатывать на POST-запросы и непосредственно создавать пользователя в базе данных:

```

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

namespace Users.Controllers
{
    public class AdminController : Controller
    {
        public ActionResult Index()
        {

```

```
        return View(UserManager.Users);
    }

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

    [HttpPost]
    public async Task<ActionResult> Create(CreateModel model)
    {
        if (ModelState.IsValid)
        {
            AppUser user = new AppUser { UserName = model.Name, Email = model.Email };
            IdentityResult result =
                await UserManager.CreateAsync(user, model.Password);

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

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

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



```
}  
}  
}
```

Важной частью этого примера является то, что POST-метод `Create` принимает аргумент типа `CreateModel`, который будет передаваться из формы, отправляемой администратором при создании пользователя. Я использую *свойство* `ModelState.IsValid`, чтобы проверить данные модели, получаемой из формы. Напомню, мы использовали атрибуты `Required`, указывающие на обязательное заполнение всех свойств `CreateModel`. Если модель проходит проверку, то создается новый экземпляр класса `AppUser` и передается в качестве параметра методу `userManager.CreateAsync`:

```
// ...  
AppUser user = new AppUser { UserName = model.Name, Email = model.Email };  
IdentityResult result =  
    await userManager.CreateAsync(user, model.Password);  
// ...
```

Метод `CreateAsync` возвращает объект, реализующий *интерфейс* `IdentityResult`, который описывает результат операции через свойства, указанные в таблице ниже:

Свойства, определенные в интерфейсе `IdentityResult`

Название	Описание
<i>Errors</i>	Возвращает список ошибок в виде строк, которые могли возникнуть при выполнении операции.
<i>Succeeded</i>	Возвращает значение <code>true</code> , если операция прошла успешно.

В примере выше мы проверяем процесс создания пользователя на наличие ошибок с помощью свойства `Succeeded`. Если ошибок нет, срабатывает перенаправление к методу действия `Index`, генерирующему список всех пользователей:

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

```
// ...  
if (result.Succeeded)  
{  
    return RedirectToAction("Index");  
}  
else  
{  
    AddErrorsFromResult(result);  
}  
// ...
```

Если свойство `Succeeded` равно `false`, тогда мы вызываем *метод* `AddErrorsFromResult()`, который перебирает все ошибки из свойства `Errors` и привязывает их к модели данных, используя возможности MVC.

Последний шаг заключается в добавлении представления для создания пользователей. В примере ниже показано содержимое добавленного файла `Views/Admin/Create.cshtml`:

```
@model Users.Models.CreateModel

@{
    ViewBag.Title = "Создание пользователя";
}

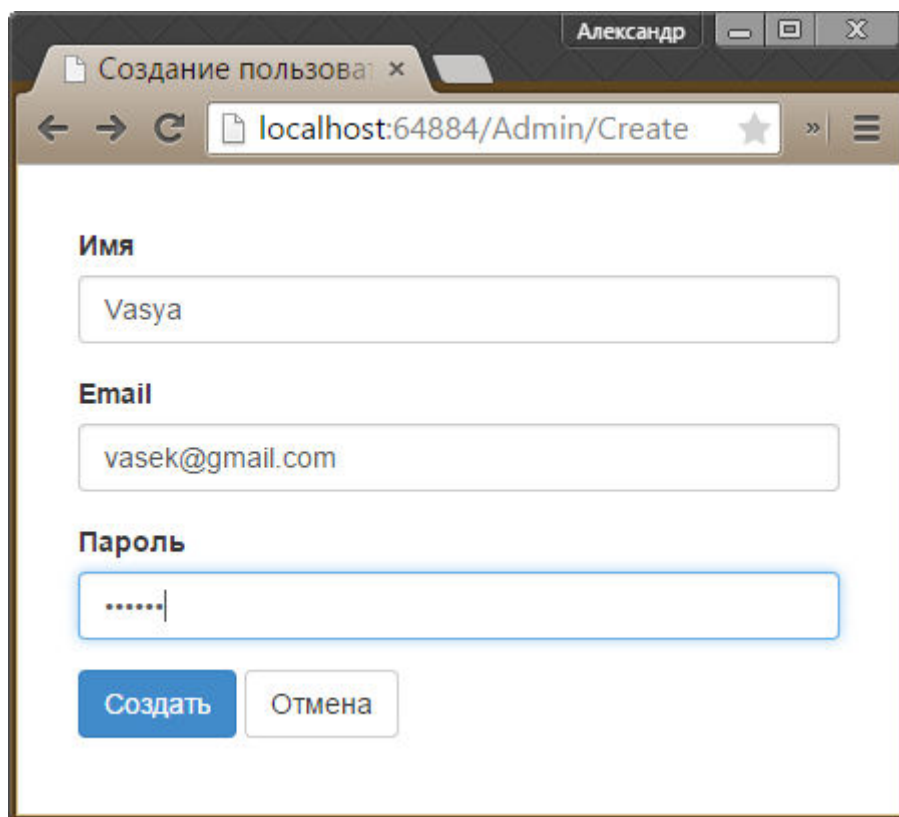
@Html.ValidationSummary(false)
@using (Html.BeginForm())
{
    <div class="form-group">
        <label>Имя</label>
        @Html.TextBoxFor(x => x.Name, new { @class = "form-control" })
    </div>
    <div class="form-group">
        <label>Email</label>
        @Html.TextBoxFor(x => x.Email, new { @class = "form-control" })
    </div>

    <div class="form-group">
        <label>Пароль</label>
        @Html.PasswordFor(x => x.Password, new { @class = "form-control" })
    </div>
    <button type="submit" class="btn btn-primary">Создать</button>

    @Html.ActionLink("Отмена", "Index", null, new { @class = "btn btn-default" })
}
```

В этом представлении нет ничего особенного — это простая форма, которая собирает значения и использует привязку моделей MVC для отправки на сервер POST-методу действия Create.

Чтобы протестировать функциональность создания пользователей, запустите приложение, перейдите по адресу /Admin/Index и нажмите кнопку «Создать»:



Создание пользователя

localhost:64884/Admin/Create

Имя

Vasya

Email

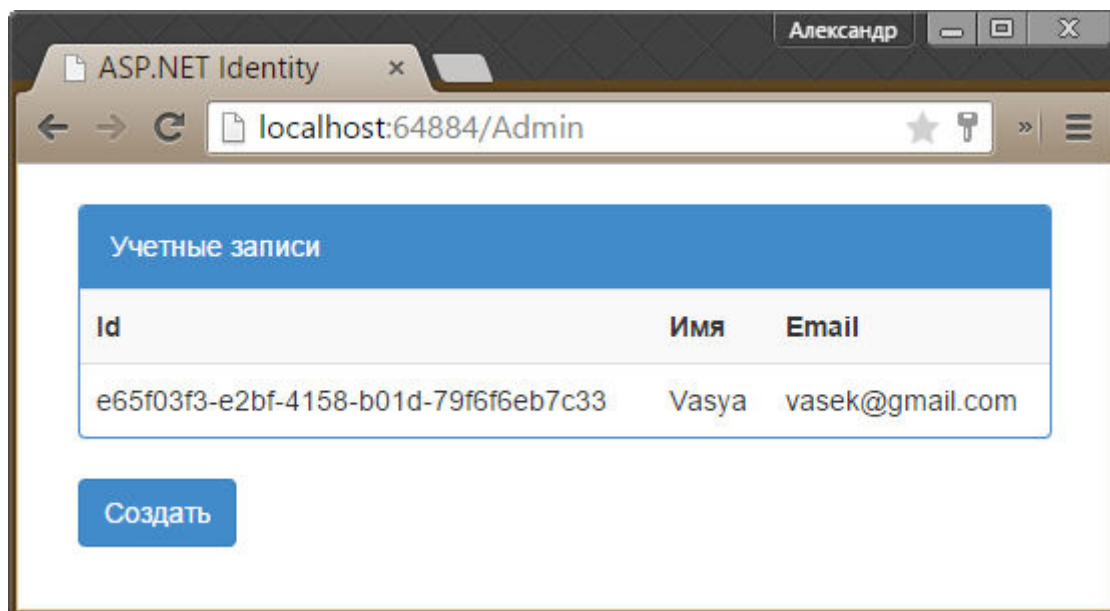
vasek@gmail.com

Пароль

.....

Создать Отмена

Заполните появившуюся форму произвольными значениями и снова нажмите по кнопке «Создать». ASP.NET Identity создаст новую учетную запись пользователя и перенаправит вас на страницу Index, как показано на рисунке ниже. Обратите внимание, что значения ID генерируются случайным образом для каждого пользователя.



ASP.NET Identity

localhost:64884/Admin

Учетные записи

Id	Имя	Email
e65f03f3-e2bf-4158-b01d-79f6f6eb7c33	Vasya	vasek@gmail.com

Создать

Протестируйте также возможность обработки ошибок. Для этого попробуйте создать еще одного пользователя с тем же именем. В браузере высветится ошибка о недопустимости этого имени:

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