

Аутентификация с помощью ASP.NET Identity

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

1

Самым базовым средством работы с ASP.NET Identity является аутентификация пользователей, и в этой статье я покажу как ее использовать. Ниже собраны базовые вопросы, которые могут возникнуть у вас в данный момент:

Что это?

Аутентификация — это проверка учетных данных, предоставляемых пользователем. После успешной аутентификации пользователя, во все последующие запросы добавляется cookie-файл с идентификатором пользователя.

Зачем нужно использовать?

Система аутентификации проверяет подлинность пользователей и является первым шагом к ограничению доступа к важной части приложения.

Как использовать в рамках MVC?

В ASP.NET MVC используется атрибут авторизации, который применяется к контроллерам и методам действий для того, чтобы ограничить доступ неавторизованным пользователям.

Далее я продемонстрирую возможности систем аутентификации и авторизации Identity на учетных записях пользователей, хранящихся в локальной базе данных. Однако, в последующих статьях мы разберем аутентификацию через социальные сети (такие как Facebook, Вконтакте и т.д.)

Процесс аутентификации/авторизации

Платформа ASP.NET Identity полностью интегрирована в ASP.NET. Это означает, что вы можете использовать стандартные средства аутентификации/авторизации MVC вместе с Identity, такие как [атрибут Authorize](#) (Пройди тесты: C# тест (легкий), .NET тест (средний)) и к методу действия Index контроллера Home и реализуем функции, которые позволят

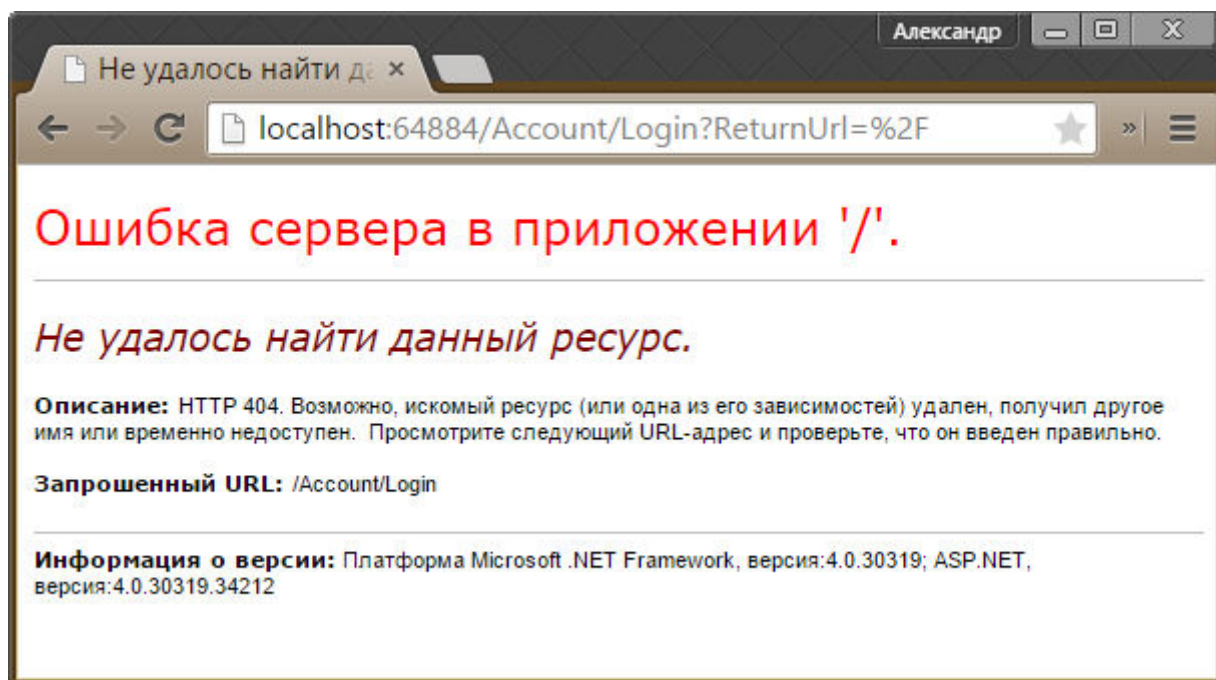
идентифицировать пользователей, чтобы они могли получить к нему доступ. В примере ниже я применил атрибут `Authorize`:

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

namespace Users.Controllers
{
    public class HomeController : Controller
    {
        [Authorize]
        public ActionResult Index()
        {
            Dictionary<string, object> data = new Dictionary<string, object>();
            data.Add("Ключ", "Значение");

            return View(data);
        }
    }
}
```

Атрибут `Authorize` по умолчанию ограничивает доступ к методам действий для пользователей, не прошедших аутентификацию. Если вы запустите приложение и запросите представление `Index` контроллера `Home` (по любому из адресов `/Home/Index`, `/Home` или просто `/`), вы увидите сообщение об ошибке, показанное на рисунке ниже:



Платформа ASP.NET предоставляет полезную информацию о пользователе в объекте `HttpContext`, которую использует атрибут `Authorize`, чтобы проверить состояние текущего запроса и посмотреть, является ли пользователь аутентифицированным. Свойство `HttpContext.User` возвращает реализацию *интерфейса `IPrincipal`*, который определен в пространстве имен `System.Security.Principal`. Интерфейс `IPrincipal` определяет свойства и методы, перечисленные в таблице ниже:

Методы и свойства, определенные в интерфейсе `IPrincipal`

Название	Описание
<i><code>Identity</code></i>	Возвращает реализацию интерфейса <code>IIdentity</code> , описывающую пользователя, связанного с запросом.
<i><code>IsInRole(role)</code></i>	Возвращает <code>true</code> , если пользователь является членом указанной роли.

Интерфейс `IIdentity`, реализация которого возвращается свойством `IPrincipal.Identity`, также содержит несколько полезных свойств:

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

Свойства интерфейса IIdentity

Название	Описание
<i>AuthenticationType</i>	Возвращает строку, описывающую механизм, который используется для аутентификации пользователей.
<i>IsAuthenticated</i>	Возвращает true, если пользователь аутентифицирован.
<i>Name</i>	Возвращает имя текущего пользователя.

ASP.NET Identity содержит модуль, который обрабатывает глобальное событие `AuthenticateRequest` жизненного цикла приложения в котором проверяет cookie в браузере пользователя, для определения того, является ли он аутентифицированным. Позже я покажу как используются эти cookie-файлы. Если пользователь аутентифицирован, ASP.NET задает свойству `IIdentity.IsAuthenticated` значение `true`. (В нашем приложении мы еще не реализовали функцию аутентификации, поэтому свойство `IsAuthenticated` будет всегда возвращать `false`.)

Модуль авторизации проверяет свойство `IsAuthenticated` и, если пользователь не прошел проверку, возвращает HTTP-статус 401 и завершает запрос. В этот момент модуль ASP.NET Identity перехватывает запрос и перенаправляет пользователя на страницу входа `/Account/Login`. Это URL-адрес, который я определил в классе `IdentityConfig` ранее:

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

Браузер перенаправит вас по адресу `/Account/Login`, но т. к. мы еще не добавили контроллер для этого запроса, появится ошибка 404.

Подготовка к реализации системы аутентификации

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

Даже несмотря на то, что наш запрос на данный момент заканчивается ошибкой, мы наглядно показали как вписывается Identity в жизненный цикл приложения

ASP.NET. Следующий шаг — создать контроллер, обрабатывающий запрос к URL /Account/Login, для отображения формы входа в приложение. Добавьте сначала новый класс view-model в файл UserViewModels.cs:

```
using System.ComponentModel.DataAnnotations;

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

    public class LoginViewModel
    {
        [Required]
        public string Name { get; set; }

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

Новый класс модели содержит свойства Name и Password, декларированные атрибутом Required, который говорит системы валидации модели, что эти свойства не должны иметь пустых значений. В реальном проекте не забывайте также добавлять клиентскую проверку при вводе пользователем имени и пароля. В данном проекте мы пропустим эту проверку, т. к. основная наша цель — разобраться в ASP.NET Identity.

Теперь добавьте контроллер Account в наше приложение, с кодом, показанным в примере ниже, который содержит две перегруженные версии метода Login. Обратите внимание, что здесь не реализована логика проверки достоверности модели, т. к. мы возвращаем представление для проверки учетных данных пользователя и входа пользователей в приложение.

```

using System.Threading.Tasks;
using System.Web.Mvc;
using Users.Models;

namespace Users.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        [AllowAnonymous]
        public ActionResult Login(string returnUrl)
        {
            ViewBag.ReturnUrl = returnUrl;
            return View();
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task

```

Хотя этот код еще не относится к аутентификации пользователей, контроллер Account все же содержит некоторую полезную инфраструктуру, не относящуюся к ASP.NET Identity.

Во-первых, обратите внимание, что оба метода Login принимают строковый аргумент returnUrl. Когда пользователь запрашивает URL-адрес с ограниченным доступом, он перенаправляется по адресу /Account/Login со строкой запроса, содержащей адрес страницы с ограниченным доступом. Например, если сейчас вы запросите адрес /Home/Index, вас перенаправит на URL следующего вида:

/Account/Login?ReturnUrl=%2FHome%2FIndex

С помощью параметра returnUrl мы сможем перенаправить пользователя, успешно прошедшего аутентификацию, на страницу, с которой он пришел. Это обеспечивает простой и понятный процесс навигации после аутентификации.

Далее обратите внимание на атрибуты, которые я применил к контроллеру и его методам действий. Функции контроллера Account (такие как смена пароля, например) по умолчанию должны быть доступны только авторизованным пользователям. Для этого мы применили атрибут Authorize к контроллеру Account и добавили **атрибут AllowAnonymous** к некоторым методам действий. Это позволяет ограничить методы действий для авторизованных пользователей по умолчанию, но открыть доступ неавторизованным пользователям для входа в приложение.

Наконец, мы добавили **атрибут ValidateAntiForgeryToken** который работает в связке с классом HtmlHelper, используемом в Razor в представлениях cshtml. Вспомогательный **метод AntiForgeryToken** защищает от **межсайтовой подделки запросов CSRF**, благодаря тому, что генерирует скрытое поле формы с токеном, который проверяется при отправке формы.

Последний подготовительный шаг — создание формы входа в приложение. Добавьте в папку /Views/Account файл представления Login.cshtml:

```
@model Users.Models.LoginViewModel
@{
    ViewBag.Title = "Авторизация на сайте";
}

<h2>Войти</h2>

@Html.ValidationSummary()

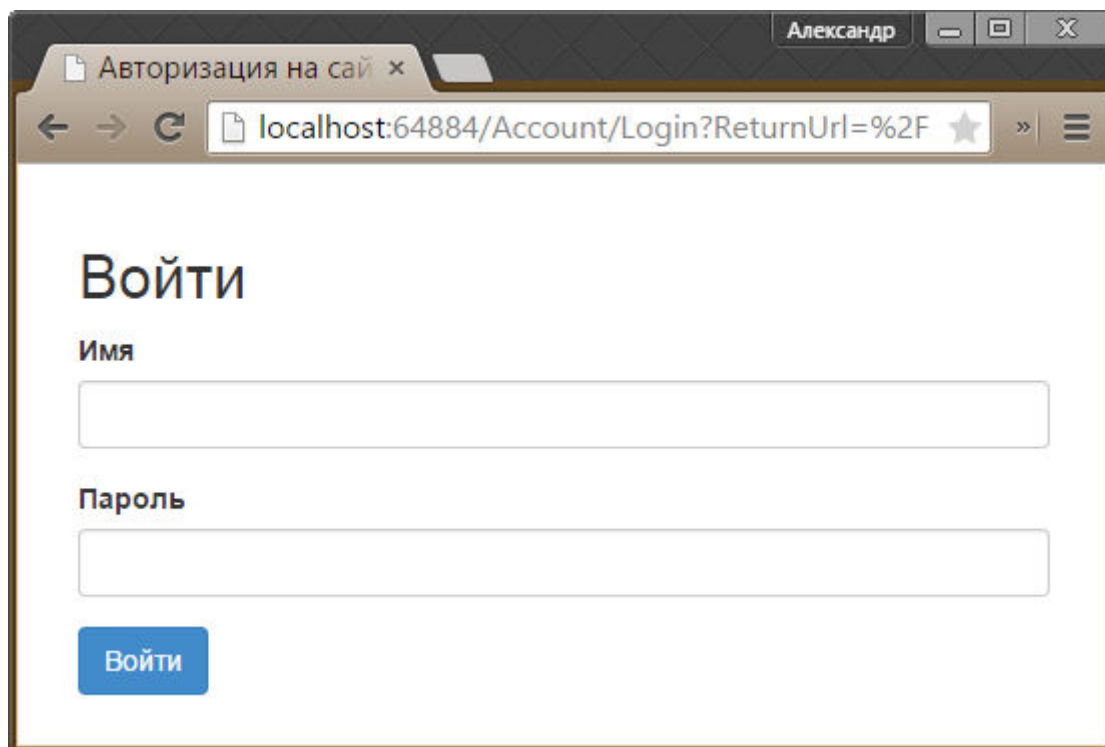
@using (Html.BeginForm()) {
    @Html.AntiForgeryToken();
    <input type="hidden" name="returnUrl" value="@ViewBag.returnUrl" />
    <div class="form-group">
        <label>Имя</label>
        @Html.TextBoxFor(x => x.Name, new { @class = "form-control" })
    </div>
    <div class="form-group">
        <label>Пароль</label>
        @Html.PasswordFor(x => x.Password, new { @class = "form-control" })
    </div>
    <button class="btn btn-primary" type="submit">Войти</button>
}
```

В этой разметке стоит обратить внимание на использование вспомогательного метода Html.AntiForgeryToken и создание скрытого поля <input> с сохранением параметра returnUrl. В остальном — это обычная форма, генерируемая с помощью Razor.

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

Итак, мы завершили подготовку к созданию системы аутентификации. Запустите приложение и перейдите по адресу /Home/Index - система перенаправит вас на

страницу входа:



Александр

Авторизация на сай x

localhost:64884/Account/Login?ReturnUrl=%2F

Войти

Имя

Пароль

Войти

Аутентификация пользователей

Запросы к страницам с ограниченным доступом успешно перехватываются и перенаправляются контроллеру Account, но учетные данные, предоставляемые пользователем, пока не проверяются системой аутентификации. В примере ниже вы можете увидеть, как мы завершим процесс входа пользователей в систему:

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

namespace Users.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        [AllowAnonymous] Пройди тесты C# тест (легкий) .NET тест (средний)
        public ActionResult Login(string returnUrl)
        {

```



```

        ViewBag returnUrl = returnUrl;
        return View();
    }

    [HttpPost]
    [AllowAnonymous]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> Login(LoginViewModel details, string returnUrl)
    {
        AppUser user = await UserManager.FindAsync(details.Name, details.Password);

        if (user == null)
        {
            ModelState.AddModelError("", "Некорректное имя или пароль.");
        }
        else
        {
            ClaimsIdentity ident = await UserManager.CreateIdentityAsync(user,
                DefaultAuthenticationTypes.ApplicationCookie);

            AuthManager.SignOut();
            AuthManager.SignIn(new AuthenticationProperties
            {
                IsPersistent = false
            }, ident);
            return Redirect(returnUrl);
        }

        return View(details);
    }

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

    private AppUserManager UserManager
    {
        get
        {
            return HttpContext.GetOwinContext().GetUserManagers<AppUserManager>().FirstOrDefault();
        }
    }

```



Самая простая часть – это проверка учетных данных, которую мы делаем с помощью метода FindAsync класса AppUserManager:

```
// ...
AppUser user = await UserManager.FindAsync(details.Name, details.Password);
// ...
```

В дальнейшем мы будем многократно обращаться к экземпляру класса AppUserManager, поэтому мы создали отдельное свойство UserManager, который возвращает экземпляр класса AppUserManager с помощью метода расширения GetOwinContext() класса HttpContext.

Метод FindAsync принимает в качестве параметров имя и пароль, введенные пользователем, и возвращает экземпляр класса пользователя (AppUser в примере) если учетная запись с такими данными существует. Если нет учетной записи с таким именем или пароль не совпадает, метод FindAsync возвращает значение null. В этом случае мы добавляем ошибку в метаданные модели, которая будет отображена пользователю.<,/

Если метод FindAsync возвращает объект AppUser, нам нужно создать файл cookie, который будет отправляться браузером в ответ на последующие запросы, благодаря чему пользователь будет автоматически аутентифицироваться в системе:

```
// ...
ClaimsIdentity ident = await UserManager.CreateIdentityAsync(user,
    DefaultAuthenticationTypes.ApplicationCookie);

AuthManager.SignOut();
AuthManager.SignIn(new AuthenticationProperties
{
    IsPersistent = false
}, ident);
return Redirect(returnUrl);
// ...
```

Пройди тесты

С# тест (легкий)

.NET тест (средний)

Первым шагом является создание объекта **ClaimsIdentity**, который идентифицирует пользователя. Класс ClaimsIdentity является частью ASP.NET Identity

и реализует интерфейс `IIdentity`, который был описан ранее. Экземпляры `ClaimsIdentity` создаются путем вызова статического **метода** `CreateIdentityAsync()` класса `userManager`. Этому методу передается объект пользователя (`IdentityUser`) и тип аутентификации в **перечислении** `DefaultAuthenticationTypes`.

Следующий шаг — удаление всех старых аутентифицирующих файлов cookie и создание новых. Мы добавили свойство `AuthManager` в контроллере `Account`, которое возвращает объект, реализующий **интерфейс** `IAuthenticationManager`, который отвечает за выполнение аутентификации в ASP.NET Identity. В таблице ниже перечислено несколько полезных методов этого интерфейса:

Наиболее полезные методы интерфейса `IAuthenticationManager`

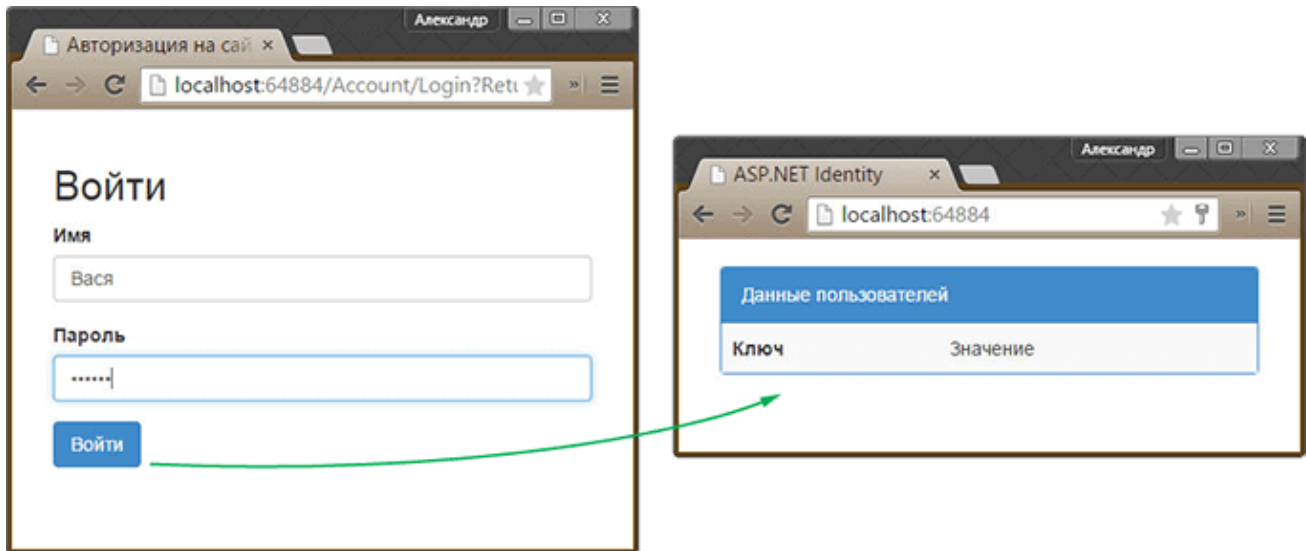
Название	Описание
<code>SignIn(options, identity)</code>	Вход пользователя в систему, что фактически означает создание аутентифицирующего cookie-файла
<code>SignOut()</code>	Выход пользователя из системы, который обычно означает удаление файлов cookie

Аргументами метода `SignIn` являются объект `AuthenticationProperties`, определяющий свойства настройки процесса аутентификации и объект `ClaimsIdentity`. Мы задали **свойству** `IsPersistent` объекта `AuthenticationProperties` значение `true` — это означает, что файлы cookie будут сохраняться между сессиями пользователей. Благодаря этому, если пользователь выйдет из приложения и зайдет, например, на следующий день, он будет автоматически аутентифицирован. (Есть и другие полезные свойства, определенные в классе `AuthenticationProperties`, но свойство `IsPersistent` является единственным, который широко используется на данный момент.)

После воссоздания файлов cookie мы перенаправляем пользователя по URL-адресу, который он просматривал до аутентификации (используя параметр `returnUrl`).

Давайте протестируем процесс аутентификации. Запустите приложение и перейдите по адресу `/Home/Index`. Браузер перенаправит вас на страницу входа, где необходимо ввести данные пользователя, которого мы создали ранее при тестировании панели администратора:

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



Двухфакторная аутентификация

В примере выше мы реализовали однофакторную систему аутентификации. Это означает, что для входа в систему пользователю нужно знать всего лишь пароль.

ASP.NET Identity поддерживает также двухфакторную аутентификацию, когда пользователь для аутентификации должен ввести что-то дополнительное. Наиболее распространенными примерами является ввод закрытого токена Secureid или код проверки подлинности, который может быть отправлен на email-адрес или по СМС (грубо говоря, вторым фактором может быть что угодно, в том числе отпечатки пальцев, сканирование радужной оболочки глаза и распознавание голоса :)

Второй фактор значительно повышает безопасность приложений, однако, зачастую не требуется. В данном руководстве я не буду показывать как реализуется двухфакторная аутентификация в Identity, т.к. она требует много подготовительных работ, таких как создание инфраструктуры, которая описывает работу с электронной почтой и реализует логику проверки сообщений. Пример реализации такой системы аутентификации вы можете увидеть в статье [Two-factor authentication using SMS and email with ASP.NET Identity](#).

