Применение ASP.NET Core Identity

В настоящей главе будет показано, как применять систему ASP.NET Core Identity для аутентификации и авторизации пользовательских учетных записей, созданных в предыдущей главе. В табл. 29.1 приведена сводка по главе.

Таблица 29.1. Сводка по главе

Задача	Решение	Листинг
Ограничение доступа к методу действия	Применяйте атрибут `Authorize`	29.1
Аутентификация пользователей	Создайте контроллер `Account`, который получает пользовательские учетные данные и проверяет их с использованием класса `UserManager`	29.2–29.5
Создание и управление ролями	Используйте класс `RoleManager`	29.6–29.10
Авторизация доступа к действию	Добавьте пользовательские учетные записи к ролям и применяйте атрибут `Authorize`, чтобы указать, какие роли могут иметь доступ к методам действий	29.11–29.18
Обеспечение наличия учетной записи администратора	Поместите в базу данных начальные данные для автоматического создания учетной записи	29.19–29.24

Подготовка проекта для примера

В главе будет продолжена работа с проектом `Users`, созданным в главе 28. В качестве подготовительных шагов запустим приложение, перейдем на URL вида `/Admin` и, щелкая на кнопке Create (Создать), добавим в базу данных пользовательские учетные записи из табл. 29.2.

Таблица 29.2. Пользовательские учетные записи, требующиеся для настоящей главы

Имя пользователя	Адрес электронной почты	Пароль
`Joe`	`joe@example.com`	`secret123`
`Alice`	`alice@example.com`	`secret123`
`Bob`	`bob@example.com`	`secret123`

По завершении запрос URL вида /Admin должен привести к отображению списка пользователей, включая описанные в табл. 29.2 (не имеет значения, если вы создадите дополнительных пользователей; важно, чтобы присутствовали пользователи, перечисленные в таблице), как показано на рис. 29.1.

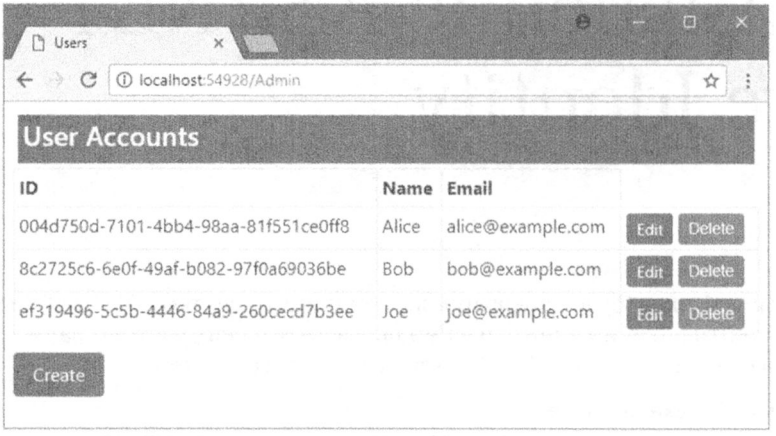

Рис. 29.1. Выполнение примера приложения

Аутентификация пользователей

Наиболее фундаментальной работой для системы ASP.NET Core Identity является аутентификация пользователей. Основной инструмент для ограничения доступа к методам действий — атрибут Authorize, который сообщает инфраструктуре MVC о том, что обрабатываться должны только запросы от аутентифицированных пользователей. В листинге 29.1 атрибут Authorize применяется к действию Index контроллера Home.

Листинг 29.1. Ограничение доступа в файле HomeController.cs из папки Controllers

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {

  public class HomeController : Controller {

    [Authorize]
    public ViewResult Index() =>
      View(new Dictionary<string, object> { ["Placeholder"] = "Placeholder" });
  }
}
```

После запуска приложения браузер отправит запрос на стандартный URL, который будет нацелен на метод действия, декорированный атрибутом Authorize. Пока что у пользователей нет никакой возможности аутентифицировать себя, поэтому в результате возникает ошибка (рис. 29.2).

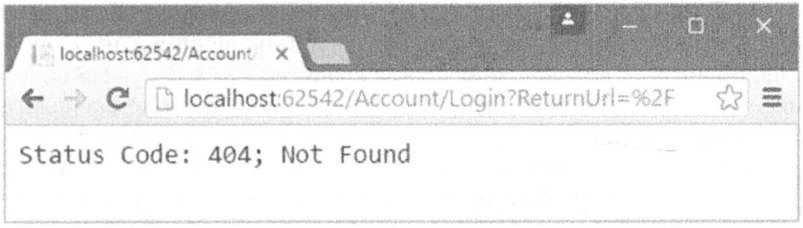

Рис. 29.2. Нацеливание на защищенный метод действия

Атрибут `Authorize` не указывает, как пользователь должен быть аутентифицирован, и не имеет прямой ссылки на ASP.NET Core Identity. Службы Identity и промежуточное ПО охватывают всю платформу ASP.NET Core, что делает их интеграцию в приложения MVC простой и бесшовной. Задача решается модификацией объектов контекста, которые описывают HTTP-запросы, и снабжением инфраструктуры MVC исходом процесса аутентификации без необходимости в предоставлении любых других деталей.

Платформа ASP.NET предоставляет информацию о пользователе через объект `HttpContext`, который используется атрибутом `Authorize` для проверки состояния текущего запроса и выяснения, был ли пользователь аутентифицирован. Свойство `HttpContext.User` возвращает реализацию интерфейса `IPrincipal`, который определен в пространстве имен `System.Security.Principal`. Интерфейс `IPrincipal` определяет свойство и метод, показанные в табл. 29.3.

Таблица 29.3. Избранные члены, определяемые интерфейсом `IPrincipal`

Имя	Описание
`Identity`	Возвращает реализацию интерфейса `IIdentity`, который описывает пользователя, ассоциированного с запросом
`IsInRole(role)`	Возвращает `true`, если пользователь является членом указанной роли. Управление авторизацией с помощью ролей объясняется в разделе "Авторизация пользователей с помощью ролей" далее в главе

Реализация интерфейса `IIdentity`, возвращаемая свойством `IPrincipal.Identity`, предлагает базовую, но полезную информацию о текущем пользователе через свойства, которые описаны в табл. 29.4.

Таблица 29.4. Избранные свойства, определяемые интерфейсом `IIdentity`

Имя	Описание
`AuthenticationType`	Возвращает строку, которая описывает механизм, используемый для аутентификации пользователя
`IsAuthenticated`	Возвращает `true`, если пользователь был аутентифицирован
`Name`	Возвращает имя текущего пользователя

Совет. В главе 30 рассматривается класс реализации, который ASP.NET Core Identity применяет для интерфейса `IIdentity`.

Промежуточное ПО ASP.NET Core Identity применяет cookie-наборы, посылаемые браузером, для выяснения, был ли пользователь аутентифицирован. Если пользователь успешно прошел аутентификацию, тогда свойство `IIdentity.IsAuthenticated` устанавливается в `true`. Поскольку пример приложения пока не располагает механизмом аутентификации, свойство `IsAuthenticated` всегда возвращает `false`, что приводит к ошибке аутентификации. В результате клиент перенаправляется на URL вида `/Account/Login`, который является стандартным URL для предоставления учетных данных аутентификации.

Браузер запрашивает URL вида `/Account/Login`, но из-за того, что в проекте он не соответствует какому-либо контроллеру или действию, сервер возвращает ответ `404 - Not Found` (404 — не найдено), давая в итоге сообщение об ошибке, показанное на рис. 29.2.

Изменение URL для входа

Хотя `/Account/Login` — стандартный URL, на который клиенты перенаправляются, когда требуется авторизация, в методе `ConfigureServices()` класса `Startup` можно указать собственный URL, изменив параметр конфигурации при настройке служб ASP.NET Core Identity:

```
...
services.ConfigureApplicationCookie(opts =>
  opts.LoginPath = "/Users/Login");
...
```

При генерации своих URL система Identity не может полагаться на систему маршрутизации, так что цель перенаправления должна указываться буквально. В случае изменения схемы маршрутизации, используемой приложением, потребуется также обеспечить изменение настройки Identity, чтобы URL по-прежнему достигал целевого контроллера.

Подготовка к реализации аутентификации

Несмотря на то что запрос заканчивается выводом сообщения об ошибке, он иллюстрирует, каким образом система ASP.NET Core Identity вписывается в стандартный жизненный цикл запросов ASP.NET. Следующий шаг заключается в реализации контроллера, который будет получать запросы для URL вида `/Account/Login` и аутентифицировать пользователя. Добавим новый класс модели в файл `UserViewModels.cs` (листинг 29.2).

Листинг 29.2. Добавление нового класса модели в файле `UserViewModels.cs` из папки `Models`

```
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; }
  }
```

```
  public class LoginModel {
    [Required]
    [UIHint("email")]
    public string Email { get; set; }

    [Required]
    [UIHint("password")]
    public string Password { get; set; }
  }
}
```

Новый класс модели имеет свойства `Email` и `Password`, декорированные атрибутом `Required`, что позволяет применять проверку достоверности моделей для контроля, предоставил ли пользователь значения. Свойства также декорированы атрибутом `UIHint`, который гарантирует, что элементы `input`, визуализируемые вспомогательной функцией дескриптора в представлении, будут иметь соответствующим образом установленные атрибуты `type`.

Совет. В реальном проекте для выяснения, предоставил ли пользователь значения для имени и пароля, перед отправкой формы серверу можно использовать проверку достоверности на стороне клиента, которая была описана в главе 27.

Добавим в папку `Controllers` файл класса по имени `AccountController.cs` и поместим в него определение контроллера, приведенное в листинге 29.3.

Листинг 29.3. Содержимое файла `AccountController.cs` из папки `Controllers`

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;

namespace Users.Controllers {
  [Authorize]
  public class AccountController : Controller {
    [AllowAnonymous]
    public IActionResult Login(string returnUrl) {
      ViewBag.returnUrl = returnUrl;
      return View();
    }
    [HttpPost]
    [AllowAnonymous]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Login(LoginModel details,
          string returnUrl) {
      return View(details);
    }
  }
}
```

В листинге 29.3 логика аутентификации не была реализована, потому что мы собираемся определить представление и затем пройти через процесс проверки пользовательских учетных данных и входа пользователей в приложение.

Хотя контроллер Account пока еще не аутентифицирует пользователей, он содержит удобную инфраструктуру, заслуживающую объяснения отдельно от кода ASP.NET Core Identity, который вскоре будет добавлен в метод действия Login().

Первым делом обратите внимание, что обе версии метода действия Login() принимают аргумент по имени returnUrl. Когда пользователь запрашивает ограниченный URL, он перенаправляется на URL вида /Account/Login со строкой запроса, в которой указан URL, куда пользователь должен быть направлен после того, как он успешно пройдет аутентификацию. Удостовериться в этом можно, запустив приложение и запросив URL вида /Home/Index. Браузер будет перенаправлен примерно так:

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

Значение параметра ReturnUrl строки запроса делает возможным такое перенаправление пользователя, что навигация между открытыми и защищенными частями приложения превращается в простой и гладкий процесс.

Далее обратите внимание на атрибуты, которые были применены в контроллере Account. Контроллеры, управляющие пользовательскими учетными записями, содержат функциональность, которая должна быть доступна только аутентифицированным пользователям, подобную сбросу пароля. С этой целью к классу контроллера был применен атрибут Authorize, а к индивидуальным методам действий — атрибут AllowAnonymous. В итоге доступ к методам действий по умолчанию ограничивается аутентифицированными пользователями, но пользователям, не прошедшим аутентификацию, разрешено входить в приложение. Кроме того, применяется описанный в главе 24 атрибут ValidateAntiForgeryToken, который работает в сочетании со вспомогательной функцией дескриптора для элемента form в целях противодействия подделке межсайтовых запросов.

Последний подготовительный шаг связан с созданием представления, которое будет визуализироваться для сбора учетных данных от пользователя. Создадим папку Views/Account и добавим в нее файл представления по имени Login.cshtml с разметкой из листинга 29.4.

Листинг 29.4. Содержимое файла Login.cshtml из папки Views/Account

```
@model LoginModel

<div class="bg-primary m-1 p-1 text-white"><h4>Log In</h4></div>

<div class="text-danger" asp-validation-summary="All"></div>

<form asp-action="Login" method="post">
  <input type="hidden" name="returnUrl" value="@ViewBag.returnUrl" />
  <div class="form-group">
    <label asp-for="Email"></label>
    <input asp-for="Email" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Password"></label>
    <input asp-for="Password" class="form-control" />
  </div>
  <button class="btn btn-primary" type="submit">Log In</button>
</form>
```

Единственный примечательный аспект данного представления — скрытый элемент input, который сохраняет аргумент returnUrl. Во всех остальных отношениях

это стандартное представление Razor, но оно завершает подготовку к аутентификации и демонстрирует способ перехвата и перенаправления запросов, не прошедших аутентификацию. Запустим приложение, чтобы протестировать новый контроллер. Когда браузер запрашивает стандартный URL приложения, он перенаправляется на URL вида `/Account/Login`, что выдает содержимое, показанное на рис. 29.3.

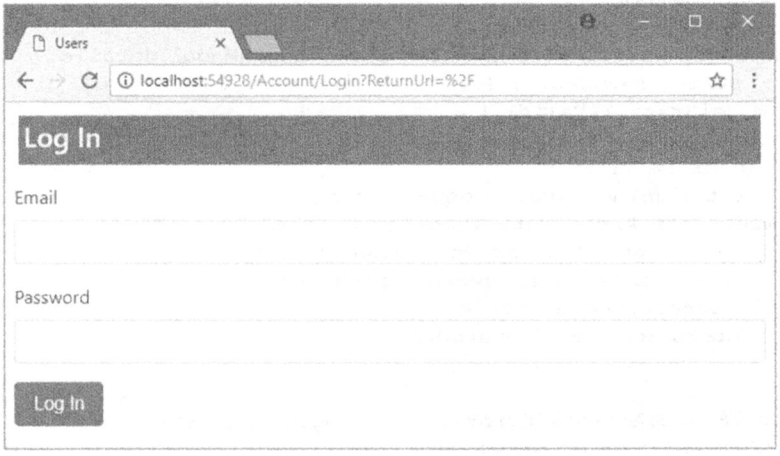

Рис. 29.3. Вывод пользователю приглашения предоставить свои учетные данные

Добавление аутентификации пользователей

Запросы к защищенным методам действий корректно перенаправляются контроллеру `Account`, но учетные данные, предоставляемые пользователем, пока еще не используются для аутентификации. В листинге 29.5 завершается реализация действия `Login` за счет применения служб ASP.NET Core Identity для аутентификации пользователя с участием деталей, хранящихся в базе данных.

Листинг 29.5. Добавление аутентификации в файле `AccountController.cs` из папки `Controllers`

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;

namespace Users.Controllers {

  [Authorize]
  public class AccountController : Controller {
    private UserManager<AppUser> userManager;
    private SignInManager<AppUser> signInManager;

    public AccountController(UserManager<AppUser> userMgr,
          SignInManager<AppUser> signinMgr) {
      userManager = userMgr;
      signInManager = signinMgr;
    }
```

```
    [AllowAnonymous]
    public IActionResult Login(string returnUrl) {
      ViewBag.returnUrl = returnUrl;
      return View();
    }
    [HttpPost]
    [AllowAnonymous]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Login(LoginModel details,
            string returnUrl) {
      if (ModelState.IsValid) {
        AppUser user = await userManager.FindByEmailAsync(details.Email);
        if (user != null) {
          await signInManager.SignOutAsync();
          Microsoft.AspNetCore.Identity.SignInResult result =
            await signInManager.PasswordSignInAsync(
              user, details.Password, false, false);
          if (result.Succeeded) {
            return Redirect(returnUrl ?? "/");
          }
        }
        ModelState.AddModelError(nameof(LoginModel.Email),
          "Invalid user or password");
      }
      return View(details);
    }
  }
}
```

Простейшей частью является получение объекта AppUser, который представляет пользователя, что делается посредством метода FindByEmailAsync() класса UserManager<AppUser>:

```
...
AppUser user = await userManager.FindByEmailAsync(details.Email);
...
```

Метод FindByEmailAsync() находит пользовательскую учетную запись, используя адрес электронной почты, который применялся при ее создании. Есть также альтернативные методы поиска по идентификатору, по имени и по входу. Адрес электронной почты используется для входа из-за того, что такой подход принят в большинстве веб-приложений, доступных через Интернет, и он набирает популярность также в корпоративных приложениях.

Если учетная запись с указанным пользователем адресом электронной почты существует, тогда производится аутентификация с применением класса SignInManager<AppUser>, для которого добавляется аргумент конструктора, распознаваемый с помощью внедрения зависимостей. Класс SignInManager используется для выполнения двух шагов аутентификации:

```
...
await signInManager.SignOutAsync();
Microsoft.AspNetCore.Identity.SignInResult result =
  await signInManager.PasswordSignInAsync(user, details.Password,
    false, false);
...
```

Метод `SignOutAsync()` аннулирует любой имеющийся у пользователя сеанс, а метод `PasswordSignIn()` проводит саму аутентификацию. В качестве аргументов метод `PasswordSignInAsync()` получает объект пользователя, предоставленный пользователем пароль, булевское значение, управляющее постоянством cookie-набора аутентификации (отключено), и признак, должна ли учетная запись блокироваться в случае некорректного пароля (отключено). Результатом метода `PasswordSignInAsync()` будет объект `SignInResult`, в котором определено булевское свойство `Succeeded`, указывающее на успешность аутентификации.

В рассматриваемом примере проверяется свойство `Succeeded`; если оно равно `true`, то пользователь перенаправляется на местоположение `returnUrl`, а если `false`, тогда добавляется ошибка проверки достоверности и затем представление `Login` отображается заново, чтобы пользователь смог повторить попытку.

Как часть процесса аутентификации система Identity добавляет к ответу cookie-набор, который браузер затем включает в любые последующие запросы, чтобы идентифицировать сеанс пользователя и ассоциированную с ним учетную запись. Вы не обязаны создавать или управлять этим cookie-набором напрямую, т.к. он поддерживается автоматически промежуточным ПО Identity.

Учет двухфакторной аутентификации

В настоящей главе выполнялась однофакторная аутентификация, при которой пользователь может быть аутентифицирован с применением одиночной порции информации, известной ему заранее: пароля.

Система ASP.NET Core Identity поддерживает также двухфакторную аутентификацию, при которой пользователю нужно кое-что дополнительное, обычно получаемое в момент, когда он желает пройти аутентификацию. Наиболее распространенным примером может служить значение из маркера SecureID или код аутентификации, который отправляется в виде сообщения электронной почты либо текстового сообщения. (Строго говоря, в качестве второго фактора может выступать что угодно, включая снятие отпечатков пальцев, сканирование радужной оболочки глаза и распознавание голоса, хотя в большинстве веб-приложений такие варианты востребованы редко.)

Защита в итоге усиливается, потому что злоумышленнику необходимо знать пароль пользователя *и* иметь доступ к тому, что предоставляется как второй фактор, например, к учетной записи электронной почты или сотовому телефону.

Двухфакторная аутентификация в книге не рассматривается по двум причинам. Во-первых, она требует большой подготовительной работы, связанной с настройкой инфраструктуры, которая распространяет сообщения электронной почты и текстовые сообщения второго фактора, а также реализации логики проверки, что выходит за рамки тематики настоящей книги.

Во-вторых, двухфакторная аутентификация вынуждает пользователя помнить о необходимости прохождения второго шага аутентификации, для чего держать поблизости, например, сотовый телефон или маркер безопасности, что в случае веб-приложений не всегда оказывается подходящим. Я более десяти лет на различных работах проносил с собой маркер SecureID того или иного вида и потерял счет, сколько раз не мог войти в систему работодателя из-за того, что забывал свой маркер дома.

Если вы заинтересованы в двухфакторной аутентификации, тогда рекомендуется опираться на стороннего поставщика вроде Google, который позволяет пользователю самостоятельно выбрать, желает ли он иметь дополнительную защиту (и смириться с неудобствами), обеспечиваемую двухфакторной аутентификацией. Использование сторонней аутентификации демонстрируется в главе 30.

Тестирование аутентификации

Чтобы протестировать аутентификацию пользователей, запустим приложение и запросим URL вида `/Home/Index`. После перенаправления на URL вида `/Account/Login` введем учетные данные одного из пользователей, перечисленных в начале главы (например, адрес электронной почты `joe@example.com` и пароль `secret123`). Щелкнем на кнопке Log In (Вход) и браузер будет перенаправлен обратно на `/Home/Index`, но на этот раз он отправит cookie-набор аутентификации, который предоставит доступ к методу действия (рис. 29.4).

Совет. Для просмотра cookie-наборов, применяемых при идентификации аутентифицированных запросов, можно использовать инструменты разработчика, встроенные в браузер.

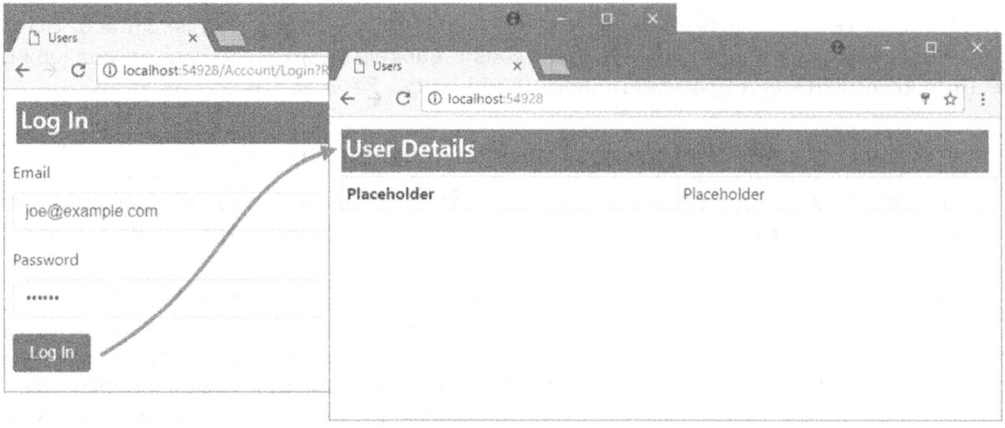

Рис. 29.4. Аутентификация пользователя

Авторизация пользователей с помощью ролей

В предыдущем разделе атрибут `Authorize` применялся в самой базовой форме, которая позволяет любому аутентифицированному пользователю выполнять метод действия. Его также можно использовать для уточнения авторизации, чтобы получить более детальный контроль над тем, какие пользователи могут выполнять те или иные действия, основываясь на принадлежности пользователя к роли.

Роль — всего лишь произвольная метка, которая определяется с целью представления разрешения на выполнение набора действий внутри приложения. Практически каждое приложение проводит различие между пользователями, которые могут выполнять административные функции, и пользователями, которые не могут. В мире ролей цель достигается за счет создания и назначения пользователям роли `Administrators`. Пользователи могут принадлежать ко многим ролям, а связанные с ролями разрешения могут быть настолько крупнозернистыми или мелкозернистыми, насколько это желательно. Таким образом, с применением разных ролей можно проводить различие между администраторами, которым позволено выполнять базовые задачи, подобные созданию новых учетных записей, и администраторами, которым разрешено выполнять более критичные операции вроде доступа к данным о платежах.

Система ASP.NET Core Identity берет на себя ответственность за управление набором ролей, определенных в приложении, и отслеживание членства пользователей в них. Но ей ничего не известно о том, что означает каждая роль; такая информация содержится внутри части MVC приложения, где доступ к методам действий ограничивается на основе членства в ролях.

Для доступа и управления ролями в ASP.NET Core Identity предусмотрен строго типизированный базовый класс по имени `RoleManager<T>`, где `T` — класс, который представляет роли в механизме хранения. Инфраструктура Entity Framework Core использует для представления ролей класс `IdentityRole`, в котором определены свойства, перечисленные в табл. 29.5.

Таблица 29.5. Избранные свойства класса `IdentityRole`

Имя	Описание
`Id`	Определяет уникальный идентификатор для роли
`Name`	Определяет имя роли
`Users`	Возвращает коллекцию объектов `IdentityUserRole`, которые представляют члены роли

При желании расширить встроенную функциональность, которая описана в главе 30 для объектов пользователей, можно создать класс роли, специфичный для приложения, но здесь будет применяться класс `IdentityRole`, т.к. он делает все, в чем нуждается большинство приложений. Когда конфигурировалось приложение в главе 28, системе ASP.NET Core Identity уже было указано на необходимость использования класса `IdentityRole` для представления ролей, что демонстрирует следующий оператор в методе `ConfigureServices()` класса `Startup`:

```
...
services.AddIdentity<AppUser, IdentityRole>(opts => {
  opts.User.RequireUniqueEmail = true;
  // opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
  opts.Password.RequiredLength = 6;
  opts.Password.RequireNonAlphanumeric = false;
  opts.Password.RequireLowercase = false;
  opts.Password.RequireUppercase = false;
  opts.Password.RequireDigit = false;
}).AddEntityFrameworkStores<AppIdentityDbContext>()
    .AddDefaultTokenProviders();
...
```

Параметры типов в методе `AddIdentity()` указывают классы, которые будут применяться для представления пользователей и ролей. В примере приложения для представления пользователей используется класс `AppUser`, а для представления ролей — встроенный класс `IdentityRole`.

Создание и удаление ролей

Чтобы продемонстрировать применение ролей, мы создадим инструмент администрирования для управления ими, начав с методов действий, которые могут создавать и удалять роли. Добавим в папку `Controllers` файл класса по имени `RoleAdminController.cs` и определим в нем контроллер, как показано в листинге 29.6.

Листинг 29.6. Содержимое файла RoleAdminController.cs из папки Controllers

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc;

namespace Users.Controllers {
  public class RoleAdminController : Controller {
    private RoleManager<IdentityRole> roleManager;
    public RoleAdminController(RoleManager<IdentityRole> roleMgr) {
      roleManager = roleMgr;
    }
    public ViewResult Index() => View(roleManager.Roles);
    public IActionResult Create() => View();
    [HttpPost]
    public async Task<IActionResult> Create([Required]string name) {
      if (ModelState.IsValid) {
        IdentityResult result
          = await roleManager.CreateAsync(new IdentityRole(name));
        if (result.Succeeded) {
          return RedirectToAction("Index");
        } else {
          AddErrorsFromResult(result);
        }
      }
      return View(name);
    }
    [HttpPost]
    public async Task<IActionResult> Delete(string id) {
      IdentityRole role = await roleManager.FindByIdAsync(id);
      if (role != null) {
        IdentityResult result = await roleManager.DeleteAsync(role);
        if (result.Succeeded) {
          return RedirectToAction("Index");
        } else {
          AddErrorsFromResult(result);
        }
      } else {
        ModelState.AddModelError("", "No role found");
      }
      return View("Index", roleManager.Roles);
    }
    private void AddErrorsFromResult(IdentityResult result) {
      foreach (IdentityError error in result.Errors) {
        ModelState.AddModelError("", error.Description);
      }
    }
  }
}
```

Управление ролями производится с использованием класса `RoleManager<T>`, где `T` — тип, предназначенный для представления ролей (встроенный класс `IdentityRole` в текущем приложении). Конструктор `RoleAdminController` объявляет зависимость

от RoleManager<IdentityRole>, распознаваемую посредством внедрения зависимостей при создании объекта контроллера.

В классе RoleManager<T> определены методы и свойства, перечисленные в табл. 29.6, которые позволяют создавать и управлять ролями.

Таблица 29.6. Члены, определяемые классом RoleManager<T>

Имя	Описание
CreateAsync(role)	Создает новую роль
DeleteAsync(role)	Удаляет указанную роль
FindByIdAsync(id)	Находит роль по ее идентификатору
FindByNameAsync(name)	Находит роль по ее имени
RoleExistsAsync(name)	Возвращает true, если роль с указанным именем существует
UpdateAsync(role)	Сохраняет изменения в указанной роли
Roles	Возвращает перечисление ролей, которые были определены

Метод действия Index() нового контроллера отображает все роли в приложении. Метод действия Create() применяется для отображения формы и получения ее данных, которые используются при создании новой роли с помощью метода CreateAsync(). Метод действия Delete() получает запрос POST и принимает уникальный идентификатор роли, который применяется для ее удаления из приложения через метод DeleteAsync(), находя объект роли с применением метода FindByIdAsync().

Создание представлений

Чтобы отобразить детали ролей в приложении, создадим папку Views/RoleAdmin и добавим в нее файл Index.cshtml с разметкой из листинга 29.7.

Листинг 29.7. Содержимое файла Index.cshtml из папки Views/RoleAdmin

```
@model IEnumerable<IdentityRole>

<div class="bg-primary m-1 p-1"><h4>Roles</h4></div>

<div class="text-danger" asp-validation-summary="ModelOnly"></div>

<table class="table table-sm table-bordered table-bordered">
  <tr><th>ID</th><th>Name</th><th>Users</th><th></th></tr>
  @if (Model.Count() == 0) {
    <tr><td colspan="4" class="text-center">No Roles</td></tr>
  } else {
    foreach (var role in Model) {
      <tr>
        <td>@role.Id</td>
        <td>@role.Name</td>
        <td identity-role="@role.Id"></td>
        <td>
          <form asp-action="Delete" asp-route-id="@role.Id" method="post">
            <a class="btn btn-sm btn-primary" asp-action="Edit"
               asp-route-id="@role.Id">Edit</a>
```

```
          <button type="submit" class="btn btn-sm btn-danger">
            Delete
          </button>
        </form>
      </td>
    </tr>
  }
}
</table>
<a class="btn btn-primary" asp-action="Create">Create</a>
```

Для отображения деталей о ролях в приложении представление использует таблицу. В третьей колонке применяется специальный атрибут элемента:

```
...
<td identity-role="@role.Id"></td>
...
```

Нужно отобразить список пользователей, которые являются членами каждой роли, что требует включения в представление очень большого объема кода. Чтобы сохранить представление простым, добавим в папку `Infrastructure` файл класса по имени `RoleUsersTagHelper.cs` и поместим в него определение класса вспомогательной функции дескриптора, приведенное в листинге 29.8.

Листинг 29.8. Содержимое файла `RoleUsersTagHelper.cs` из папки `Infrastructure`

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Users.Models;

namespace Users.Infrastructure {
  [HtmlTargetElement("td", Attributes = "identity-role")]
  public class RoleUsersTagHelper : TagHelper {
    private UserManager<AppUser> userManager;
    private RoleManager<IdentityRole> roleManager;
    public RoleUsersTagHelper(UserManager<AppUser> usermgr,
          RoleManager<IdentityRole> rolemgr) {
      userManager = usermgr;
      roleManager = rolemgr;
    }
    [HtmlAttributeName("identity-role")]
    public string Role { get; set; }
    public override async Task ProcessAsync(TagHelperContext context,
          TagHelperOutput output) {
      List<string> names = new List<string>();
      IdentityRole role = await roleManager.FindByIdAsync(Role);
      if (role != null) {
        foreach (var user in userManager.Users) {
          if (user != null
              && await userManager.IsInRoleAsync(user, role.Name)) {
            names.Add(user.UserName);
          }
```

```
        }
      }
      output.Content.SetContent(names.Count == 0 ?
        "No Users" : string.Join(", ", names));
    }
  }
}
```

Класс `RoleUsersTagHelper` оперирует на элементах `td` посредством атрибута `identity-role`, который используется для получения имени обрабатываемой роли. Объекты `RoleManager<IdentityRole>` и `UserManager<AppUser>` позволяют отправлять запросы базе данных Identity для построения списка имен пользователей в роли. В листинге 29.9 к файлу импортирования представлений добавляется класс вспомогательной функции дескриптора `RoleUsersTagHelper` и выражение `@using`, чтобы на типы EF Core можно было ссылаться внутри представлений, не указывая пространство имен.

Листинг 29.9. Добавление класса вспомогательной функции дескриптора в файле `_ViewImports.cshtml` из папки

```
@using Users.Models
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper Users.Infrastructure.*, Users
```

Добавим в папку `Views/RoleAdmin` файл представления по имени `Create.cshtml` с разметкой из листинга 29.10 для поддержки добавления новых ролей.

Листинг 29.10. Содержимое файла `Create.cshtml` из папки `Views/RoleAdmin`

```
@model string

<div class="bg-primary m-1 p-1"><h4>Create Role</h4></div>

<div asp-validation-summary="All" class="text-danger"></div>

<form asp-action="Create" method="post">
  <div class="form-group">
    <label for="name"></label>
    <input name="name" class="form-control" />
  </div>
  <button type="submit" class="btn btn-primary">Create</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

Для создания роли из данных формы требуется только имя — вот почему в `Create.cshtml` есть возможность применения `string` в качестве класса модели представления. Мы хотим воспользоваться преимуществами проверки достоверности модели, чтобы гарантировать предоставление пользователем значения, когда форма отправлена, но для такой простой задачи не стоит создавать отдельный класс модели. Взглянув на метод `Create()`, принимающий запросы `POST` в листинге 29.6, вы заметите, что атрибут проверки достоверности `Required` применяется прямо к параметру. Результат будет таким же, как в случае применения этого атрибута в классе модели, и появляется возможность задействовать встроенный процесс проверки достоверности моделей.

Тестирование создания и удаления ролей

Чтобы протестировать новый контроллер, запустим приложение и перейдем на URL вида `/RoleAdmin`. Щелкнем на кнопке Create (Создать), введем имя в элементе `input` и щелкнем на второй кнопке Create. Новая роль будет сохранена в базе данных и отображена после перенаправления браузера на действие `Index` (рис. 29.5). Щелкнув на кнопке Delete (Удалить), роль можно удалить из приложения.

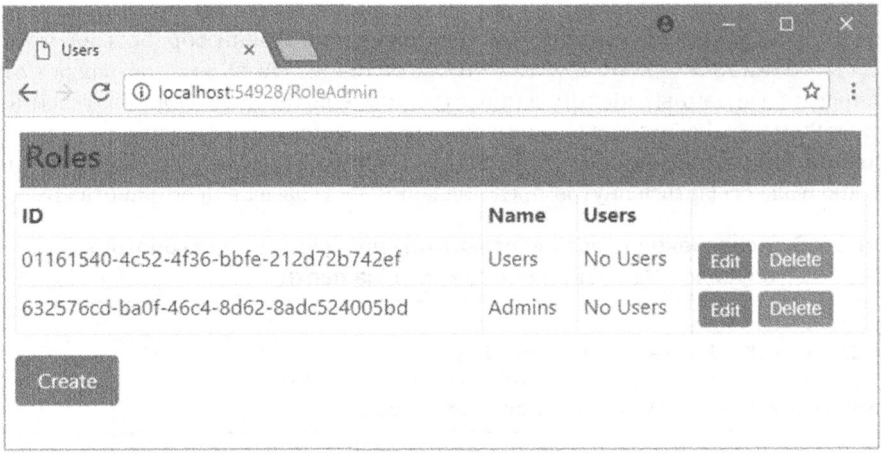

Рис. 29.5. Создание новой роли

Управление членством в ролях

Следующий шаг — обеспечение возможности добавления и удаления пользователей из ролей. Сам процесс несложен; он предусматривает взятие данных роли из класса `RoleManager` и их ассоциирование с деталями индивидуальных пользователей.

Для начала понадобится определить несколько классов моделей представлений, которые будут представлять членство в роли и получать новый набор инструкций относительно членства от пользователя. В листинге 29.11 показаны добавления, внесенные в файл `UserViewModels.cs` из папки `Models`.

Листинг 29.11. Добавление моделей представлений в файле `UserViewModels.cs` из папки `Models`

```
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using Microsoft.AspNetCore.Identity;

namespace Users.Models {

  public class CreateModel {
    [Required]
    public string Name { get; set; }
    [Required]
    public string Email { get; set; }
    [Required]
    public string Password { get; set; }
  }
```

```
  public class LoginModel {
    [Required]
    [UIHint("email")]
    public string Email { get; set; }
    [Required]
    [UIHint("password")]
    public string Password { get; set; }
  }
  public class RoleEditModel {
    public IdentityRole 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 RoleId { get; set; }
    public string[] IdsToAdd { get; set; }
    public string[] IdsToDelete { get; set; }
  }
}
```

Класс `RoleEditModel` представляет роль и детали о пользователях в системе, категоризированные по членству в роли. Класс `RoleModificationModel` представляет набор изменений роли.

В листинге 29.12 к контроллеру `RoleAdmin` добавляются новые методы действий, которые используют модели представлений из листинга 29.11 для управления членством в ролях.

Листинг 29.12. Добавление методов действий в файле `RoleAdminController.cs` из папки `Controllers`

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Collections.Generic;

namespace Users.Controllers {
  public class RoleAdminController : Controller {
    private RoleManager<IdentityRole> roleManager;
    private UserManager<AppUser> userManager;

    public RoleAdminController(RoleManager<IdentityRole> roleMgr,
        UserManager<AppUser> userMrg) {
      roleManager = roleMgr;
      userManager = userMrg;
    }

    // ...для краткости другие методы действий не показаны...

    public async Task<IActionResult> Edit(string id) {
      IdentityRole role = await roleManager.FindByIdAsync(id);
      List<AppUser> members = new List<AppUser>();
```

```
    List<AppUser> nonMembers = new List<AppUser>();
    foreach (AppUser user in userManager.Users) {
      var list = await userManager.IsInRoleAsync(user, role.Name)
        ? members : nonMembers;
      list.Add(user);
    }
    return View(new RoleEditModel {
      Role = role,
      Members = members,
      NonMembers = nonMembers
    });
  }
  [HttpPost]
  public async Task<IActionResult> Edit(RoleModificationModel model) {
    IdentityResult result;
    if (ModelState.IsValid) {
      foreach (string userId in model.IdsToAdd ?? new string[] { }) {
        AppUser user = await userManager.FindByIdAsync(userId);
        if (user != null) {
          result = await userManager.AddToRoleAsync(user,
            model.RoleName);
          if (!result.Succeeded) {
            AddErrorsFromResult(result);
          }
        }
      }
      foreach (string userId in model.IdsToDelete ?? new string[] { }) {
        AppUser user = await userManager.FindByIdAsync(userId);
        if (user != null) {
          result = await userManager.RemoveFromRoleAsync(user,
            model.RoleName);
          if (!result.Succeeded) {
            AddErrorsFromResult(result);
          }
        }
      }
    }
    if (ModelState.IsValid) {
      return RedirectToAction(nameof(Index));
    } else {
      return await Edit(model.RoleId);
    }
  }
  private void AddErrorsFromResult(IdentityResult result) {
    foreach (IdentityError error in result.Errors) {
      ModelState.AddModelError("", error.Description);
    }
  }
 }
}
```

Большая часть кода в версии GET метода действия Edit() отвечает за генерацию наборов членов и не членов выбранной роли. После того как все пользователи катего-

ризированы, новый экземпляр класса RoleEditModel передается методу View(), так что данные могут быть отображены с применением стандартного представления.

Версия POST метода действия Edit() отвечает за добавление и удаление пользователей в и из ролей. Класс UserManager<T> предлагает методы для работы с ролями, которые описаны в табл. 29.7.

Таблица 29.7. Методы, связанные с ролями, которые определяет класс UserManager<T>

Имя	Описание
AddToRoleAsync(user, name)	Добавляет идентификатор пользователя к роли с указанным именем
GetRolesAsync(user)	Возвращает список имен ролей, членом которых является пользователь
IsInRoleAsync(user, name)	Возвращает true, если пользователь имеет членство в роли с указанным именем
RemoveFromRoleAsync(user, name)	Удаляет пользователя как члена из роли с указанным именем

Причудливость методов, относящихся к ролям, связана с тем, что они оперируют с именами ролей, хотя роли имеют также и уникальные идентификаторы. По указанной причине класс модели представления RoleModificationModel имеет свойство RoleName.

В листинге 29.13 приведено содержимое файла представления Edit.cshtml, добавленного в папку Views/RoleAdmin, которое позволяет пользователю редактировать членство в роли.

Листинг 29.13. Содержимое файла Edit.cshtml из папки Views/RoleAdmin

```
@model RoleEditModel

<div class="bg-primary m-1 p-1 text-white"><h4>Edit Role</h4></div>

<div asp-validation-summary="All" class="text-danger"></div>
<form asp-action="Edit" method="post">
  <input type="hidden" name="roleName" value="@Model.Role.Name" />
  <input type="hidden" name="roleId" value="@Model.Role.Id" />
  <h6 class="bg-info p-1 text-white">Add To @Model.Role.Name</h6>
  <table class="table table-bordered table-sm">
    @if (Model.NonMembers.Count() == 0) {
      <tr><td colspan="2">All Users Are Members</td></tr>
    } else {
      @foreach (AppUser user in Model.NonMembers) {
        <tr>
          <td>@user.UserName</td>
          <td>
            <input type="checkbox" name="IdsToAdd" value="@user.Id">
          </td>
        </tr>
      }
    }
  </table>
```

```
  <h6 class="bg-info p-1 text-white">Remove From @Model.Role.Name</h6>
  <table class="table table-bordered table-sm">
    @if (Model.Members.Count() == 0) {
      <tr><td colspan="2">No Users Are Members</td></tr>
    } else {
      @foreach (AppUser user in Model.Members) {
        <tr>
          <td>@user.UserName</td>
          <td>
            <input type="checkbox" name="IdsToDelete" value="@user.Id">
          </td>
        </tr>
      }
    }
  </table>
  <button type="submit" class="btn btn-primary">Save</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

Представление `Edit.cshtml` содержит две таблицы: одну для пользователей, не являющихся членами выбранной роли, и еще одну для пользователей, принадлежащих роли. Рядом с именем каждого пользователя отображается флажок, который позволяет изменять членство. Таблицы находятся внутри формы, которая отправляется методу действия `Edit()` и привязана к классу модели `RoleModificationModel`, обеспечивая легкий доступ к списку изменений членства в роли.

Тестирование редактирования членства в роли

Чтобы протестировать поддержку членства в ролях, запустим приложение, перейдем на URL вида `/RoleAdmin` и создадим новую роль по имени `Users`. Щелкнем на кнопке Edit (Редактировать); все пользователи в приложении отобразятся внутри списка не членов (рис. 29.6).

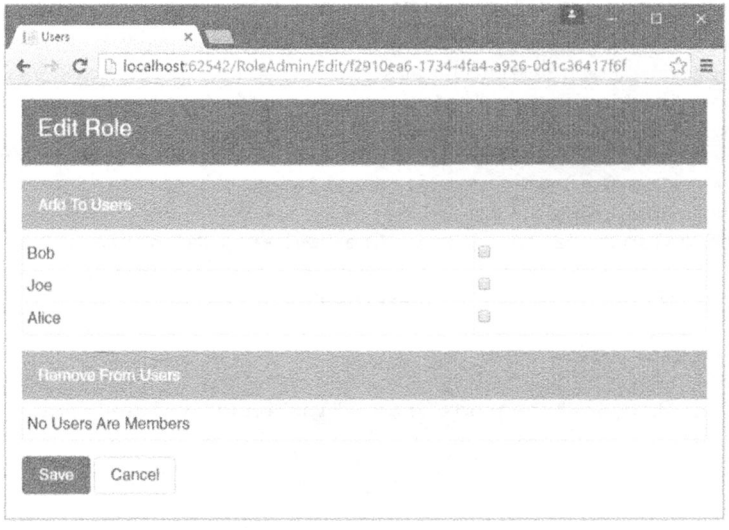

Рис. 29.6. Просмотр и редактирование членства в ролях

Отметим флажки для пользователей Alice и Joe (две учетные записи из числа добавленных в систему Identity в начале главы) и щелкнем на кнопке Save (Сохранить). В списке членов роли Users появятся пользователи Alice и Joe, как показано на рис. 29.7.

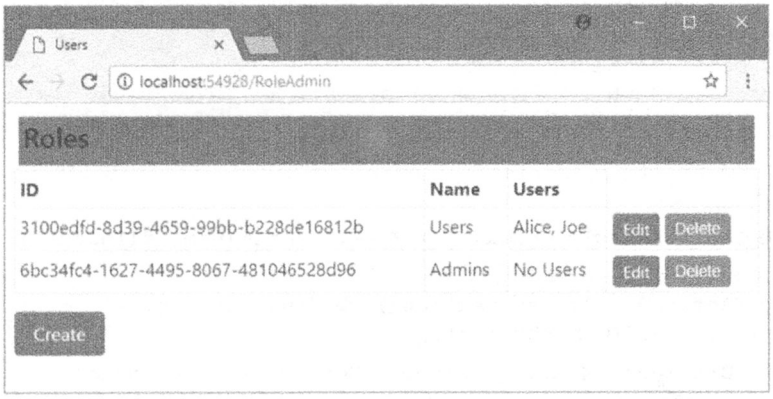

Рис. 29.7. Управление членством в ролях

Использование ролей для авторизации

Теперь, когда в приложении присутствуют роли, их можно применять в качестве основы для авторизации посредством атрибута Authorize. Чтобы облегчить тестирование авторизации на основе ролей, добавим в контроллер Account метод Logout(), который сделает возможным выход и последующий вход от имени другого пользователя для демонстрации работы членства в ролях (листинг 29.14).

Листинг 29.14. Добавление метода Logout() в файле AccountController.cs из папки Controllers

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;

namespace Users.Controllers {

  [Authorize]
  public class AccountController : Controller {
    private UserManager<AppUser> userManager;
    private SignInManager<AppUser> signInManager;

    // ...для краткости другие методы действий не показаны...

    [Authorize]
    public async Task<IActionResult> Logout() {
      await signInManager.SignOutAsync();
      return RedirectToAction("Index", "Home");
    }
  }
}
```

Обновим контроллер Home, добавив новый метод действия и передав представлению информацию об аутентифицированном пользователе (листинг 29.15).

Листинг 29.15. Добавление метода действия и информации об учетной записи в файле HomeController.cs из папки Controllers

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {

  public class HomeController : Controller {

    [Authorize]
    public IActionResult Index() => View(GetData(nameof(Index)));

    [Authorize(Roles = "Users")]
    public IActionResult OtherAction() => View("Index",
      GetData(nameof(OtherAction)));

    private Dictionary<string, object> GetData(string actionName) =>
      new Dictionary<string, object> {
        ["Action"] = actionName,
        ["User"] = HttpContext.User.Identity.Name,
        ["Authenticated"] = HttpContext.User.Identity.IsAuthenticated,
        ["Auth Type"] = HttpContext.User.Identity.AuthenticationType,
        ["In Users Role"] = HttpContext.User.IsInRole("Users")
      };
  }
}
```

Атрибут Authorize для метода действия Index() не изменялся, но при его применении к методу OtherAction() было установлено свойство Roles с целью указания на то, что доступ к этому методу должен быть разрешен только членам роли Users. Кроме того, определен метод GetData(), который добавляет базовые сведения об удостоверении пользователя с использованием свойств, доступных через объект HttpContext.

Совет. Атрибут Authorize можно также применять для авторизации доступа на основе списка индивидуальных пользовательских имен. Такая возможность привлекательна в небольших проектах, но требует изменения кода в контроллерах всякий раз, когда изменяется набор авторизуемых пользователей, и обычно означает необходимость в повторном проходе через цикл тестирования и развертывания. Использование ролей для авторизации изолирует приложение от изменений в отдельных пользовательских учетных записях и позволяет управлять доступом к приложению посредством информации о членстве в ролях, хранящейся в системе ASP.NET Core Identity.

Последнее изменение касается файла Index.cshtml из папки Views/Home, который применяется обоими действиями в контроллере Home, и связано с добавлением ссылки, нацеленной на метод Logout() контроллера Account (листинг 29.16).

Листинг 29.16. Добавление ссылки на метод `Logout()` в файле `Index.cshtml` из папки `Views/Home`

```
@model Dictionary<string, object>

<div class="bg-primary m-1 p-1 text-white"><h4>User Details</h4></div>

<table class="table table-sm table-bordered m-1 p-1">
  @foreach (var kvp in Model) {
    <tr><th>@kvp.Key</th><td>@kvp.Value</td></tr>
  }
</table>

@if (User?.Identity?.IsAuthenticated ?? false) {
  <a asp-controller="Account" asp-action="Logout"
     class="btn btn-danger">Logout</a>
}
```

Чтобы протестировать аутентификацию, запустим приложение и перейдем на URL вида `/Home/Index`. Браузер будет перенаправлен так, что можно ввести пользовательские учетные данные. Совершенно не имеет значения, какой пользователь из табл. 29.2 будет выбран, поскольку атрибут `Authorize`, примененный к действию `Index`, разрешает доступ любому аутентифицированному пользователю.

Однако если запросить URL вида `/Home/OtherAction`, то выбор пользователя из табл. 29.2 будет иметь значение, т.к. членами роли `Users`, требующейся для доступа к методу `OtherAction()`, являются только пользователи `Alice` и `Joe`. В случае входа от имени пользователя `Bob` браузер будет перенаправлен на URL вида `/Account/AccessDenied`, который используется, когда пользователь не имеет возможности получить доступ к методу действия. Для обработки такой ситуации в контроллер `Account` добавлен метод `AccessDenied()`, поэтому теперь есть действие, обрабатывающее запрос (листинг 29.17).

Совет. Устанавливая конфигурационное свойство `AccessDeniedPath`, можно изменять URL вида `/Account/AccessDenied`. Во врезке "Изменение URL для входа" ранее в главе был приведен похожий пример.

Листинг 29.17. Добавление метода действия в файле `AccountController.cs` из папки `Controllers`

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;

namespace Users.Controllers {

  [Authorize]
  public class AccountController : Controller {
    private UserManager<AppUser> userManager;
    private SignInManager<AppUser> signInManager;

    public AccountController(UserManager<AppUser> userMgr,
           SignInManager<AppUser> signinMgr) {
```

```
      userManager = userMgr;
      signInManager = signinMgr;
    }
    // ...для краткости другие методы действий не показаны...
    [AllowAnonymous]
    public IActionResult AccessDenied() {
      return View();
    }
  }
}
```

Чтобы снабдить действие `AccessDenied` представлением для отображения, создадим в папке `Views/Account` файл по имени `AccessDenied.cshtml` с содержимым из листинга 29.18.

Листинг 29.18. Содержимое файла `AccessDenied.cshtml` из папки `Views/Account`

```
<div class="bg-danger mb-1 p-2 text-white"><h4>Access Denied</h4></div>
<a asp-action="Index" asp-controller="Home" class="btn btn-primary">OK</a>
```

Запустим приложение, запросим URL вида `/Account/Login` и войдем от имени `bob@example.com`. Когда процесс аутентификации завершится, браузер будет перенаправлен на URL вида `/Home/Index`, который отобразит детали учетной записи, как показано слева на рис. 29.8, проясняя, что пользователь `Bob` не является членом роли `Users`. Затем запросим URL вида `/Home/OtherAction`, который нацелен на действие, защищенное с помощью доступа на основе ролей. Пользователь `Bob` не имеет требуемого членства в роли, поэтому браузер будет перенаправлен на URL вида `/Account/AccessDenied`, как демонстрируется справа на рис. 29.8.

Совет. Роли загружаются во время входа пользователя, т.е. изменение ролей для пользователя, который в текущий момент аутентифицирован, вступит в силу только после того, как пользователь выйдет и затем аутентифицируется заново.

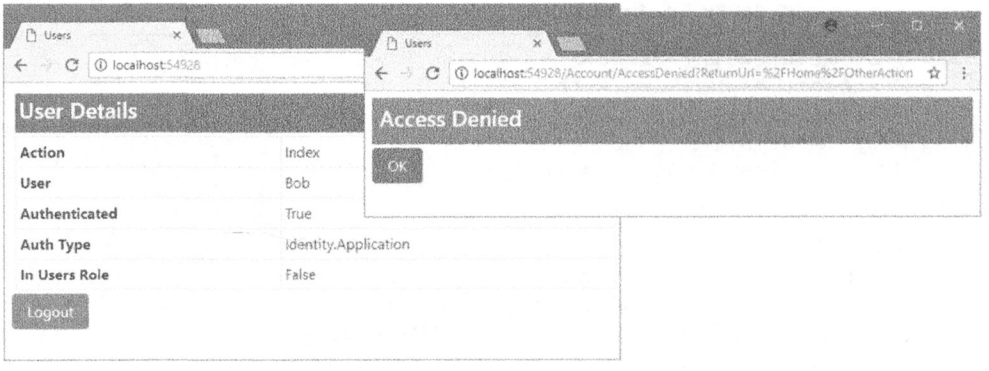

Рис. 29.8. Применение авторизации на основе ролей

Помещение в базу данных начальной информации

В рассматриваемом проекте осталась одна проблема — доступ к контроллерам Admin и RoleAdmin не ограничен. Это классическая проблема курицы и яйца. Дело в том, что для ограничения доступа необходимо создать пользователей и роли, но контроллеры Admin и RoleAdmin являются инструментами управления пользователями, и если защитить их с помощью атрибута Authorize, тогда не будет никаких учетных данных, которые предоставят к ним доступ, особенно при первом развертывании приложения.

Проблема решается за счет помещения в базу данных начальной информации при запуске приложения. В листинге 29.19 приведено содержимое файла appsettings.json с добавленными конфигурационными данными, указывающими детали для учетной записи, которая будет создана.

Листинг 29.19. Добавление конфигурационных данных в файле appsettings.json из папки Users

```
{
  "Data": {
    "AdminUser": {
      "Name": "Admin",
      "Email": "admin@example.com",
      "Password": "secret",
      "Role": "Admins"
    },
    "SportStoreIdentity": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;
        Database=IdentityUsers;
        Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
}
```

В категории Data:AdminUser предоставлены четыре значения, требующиеся для создания учетной записи и ее назначения роли, которая обеспечит возможность использования административных инструментов.

Внимание! Помещение паролей в текстовые конфигурационные файлы означает необходимость добавления в процесс развертывания приложения возможности изменять пароль стандартной учетной записи и инициализировать новую базу данных при начальном развертывании.

Добавим в класс AppIdentityDbContext статический метод, как показано в листинге 29.20. Код для создания стандартной учетной записи вовсе не обязан находиться в данном классе, но для меня это место выглядит вполне естественным, и я поступаю так в своих проектах. Вы можете применять отдельный класс, как делается в приложении SportsStore.

Листинг 29.20. Добавление метода в файле AppIdentityDbContext.cs из папки Models

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Threading.Tasks;

namespace Users.Models {

  public class AppIdentityDbContext : IdentityDbContext<AppUser> {

    public AppIdentityDbContext(DbContextOptions<AppIdentityDbContext>
      options)
        : base(options) { }

    public static async Task CreateAdminAccount(
        IServiceProvider serviceProvider,
        IConfiguration configuration) {

      UserManager<AppUser> userManager =
        serviceProvider.GetRequiredService<UserManager<AppUser>>();
      RoleManager<IdentityRole> roleManager =
        serviceProvider.GetRequiredService<RoleManager<IdentityRole>>();

      string username = configuration["Data:AdminUser:Name"];
      string email = configuration["Data:AdminUser:Email"];
      string password = configuration["Data:AdminUser:Password"];
      string role = configuration["Data:AdminUser:Role"];

      if (await userManager.FindByNameAsync(username) == null) {
        if (await roleManager.FindByNameAsync(role) == null) {
          await roleManager.CreateAsync(new IdentityRole(role));
        }

        AppUser user = new AppUser {
          UserName = username,
          Email = email
        };

        IdentityResult result = await userManager
          .CreateAsync(user, password);
        if (result.Succeeded) {
          await userManager.AddToRoleAsync(user, role);
        }
      }
    }
  }
}
```

Метод `CreateAdminAccount()` принимает объект реализации `IServiceProvider`, который применяется для получения объектов `UserManager` и `RoleManager`, а также объект реализации `IConfiguration`, используемый для извлечения данных из файла `appsetting.json`. Код в методе `CreateAdminAccount()` проверяет, существует ли пользователь, и если нет, то создает его и назначает указанной роли, которая также

при необходимости создается. В листинге 29.21 в класс Startup добавлен оператор, вызывающий метод CreateAdminAccount() после того, как остальная часть приложения настроена и сконфигурирована.

Листинг 29.21. Вызов метода базы данных в файле Startup.cs из папки Users

```
...
public void Configure(IApplicationBuilder app) {
  app.UseStatusCodePages();
  app.UseDeveloperExceptionPage();
  app.UseStaticFiles();
  app.UseAuthentication();
  app.UseMvcWithDefaultRoute();
  AppIdentityDbContext.CreateAdminAccount(app.ApplicationServices,
    Configuration).Wait();
}
...
```

Поскольку здесь производится доступ к службе с ограниченной областью действия через поставщик IApplicationBuilder.ApplicationServices, потребуется также отключить проверку области действия в классе Program (листинг 29.22).

Листинг 29.22. Отключение проверки области действия в файле Program.cs из папки Users

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace Users {
  public class Program {
    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseDefaultServiceProvider(options =>
          options.ValidateScopes = false)
        .Build();
  }
}
```

Располагая надежной стандартной учетной записью в базе данных Identity, можно применять атрибут Authorize для защиты контроллеров Admin и RoleAdmin. В листинге 29.23 приведены изменения, внесенные в контроллер Admin.

Листинг 29.23. Ограничение доступа в файле AdminController.cs из папки Controllers

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {

  [Authorize(Roles = "Admins")]
  public class AdminController : Controller {
    // ...для краткости операторы не показаны...
  }
}
```

В листинге 29.24 показано соответствующее изменение, внесенное в контроллер RoleAdmin.

Листинг 29.24. Ограничение доступа в файле RoleAdminController.cs из папки Controllers

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Collections.Generic;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {
  [Authorize(Roles = "Admins")]
  public class RoleAdminController : Controller {
    // ...для краткости операторы не показаны...
  }
}
```

Теперь следует запустить приложение и запросить URL вида /Admin или /RoleAdmin. Если ранее уже был осуществлен вход от имени какого-то другого пользователя, тогда понадобится выйти. В противном случае будет предложено предоставить учетные данные, так что можно ввести admin@example.com и пароль secret и получить доступ к административным функциям.

Резюме

В главе объяснялось, как использовать систему ASP.NET Core Identity для аутентификации и авторизации пользователей. Вы узнали, каким образом собирать и проверять пользовательские учетные данные и ограничивать доступ к методам действий на основе ролей, членом которых является пользователь.