ГЛАВА 29

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.

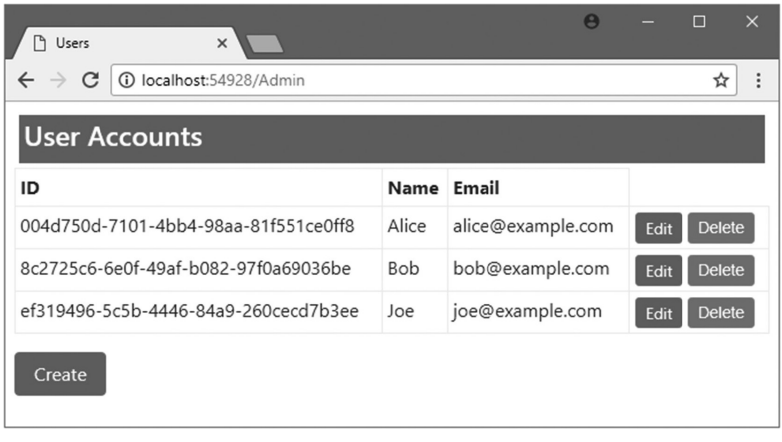

Рис. 29.1. Выполнение примера приложения

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).

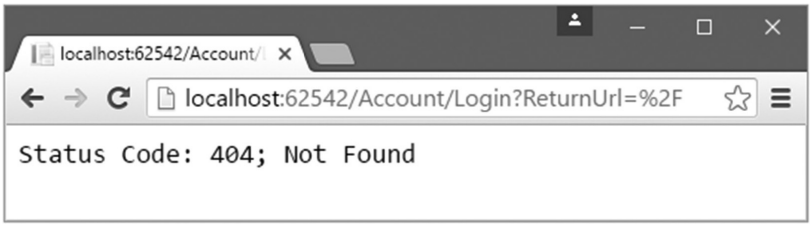

Рис. 29.2. Нацеливание на защищенный метод действия

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.

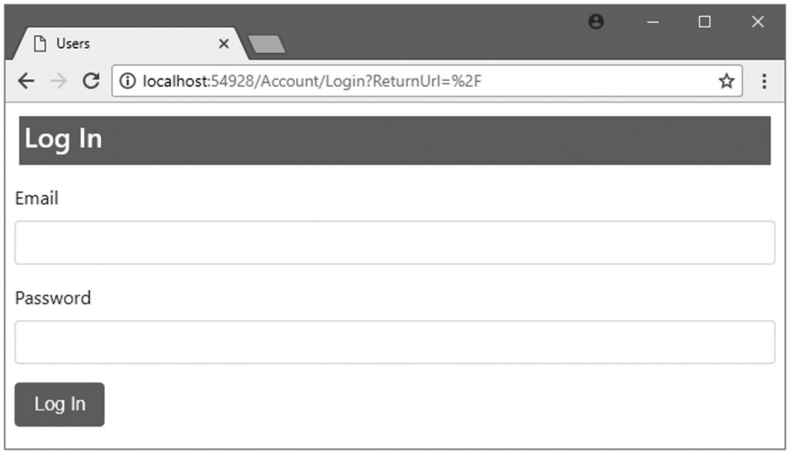

Рис. 29.3. Вывод пользователю приглашения предоставить свои учетные данные

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.

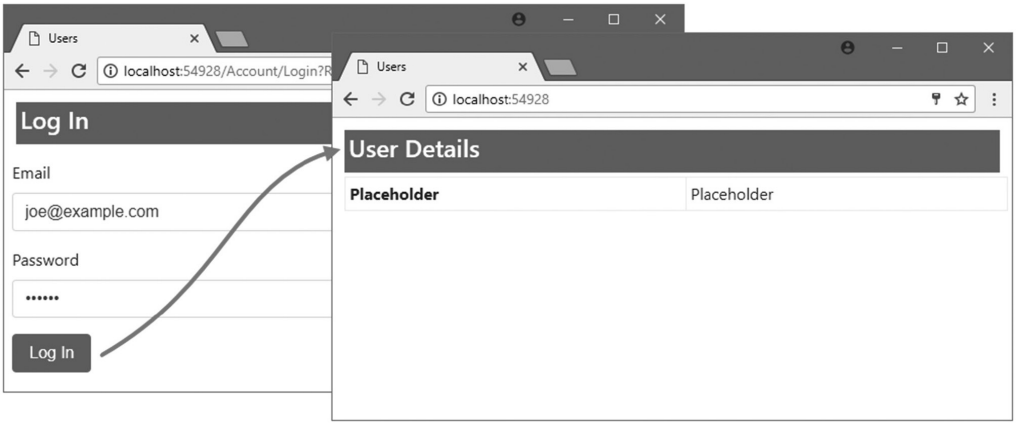

Рис. 29.4. Аутентификация пользователя

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.