Введение в ASP.NET Core Identity

Система ASP.NET Core Identity представляет собой API-интерфейс от Microsoft, предназначенный для управления пользователями в приложениях ASP.NET. В настоящей главе демонстрируется процесс настройки ASP.NET Core Identity и создания простого инструмента администрирования пользователей, который управляет индивидуальными пользовательскими учетными записями, хранящимися в базе данных.

Система ASP.NET Core Identity поддерживает другие виды пользовательских учетных записей, такие как записи, хранящиеся с использованием Active Directory, но здесь они не рассматриваются, потому что редко применяются вне корпораций (где реализации Active Directory оказываются настолько замысловатыми, что очень трудно отыскать полезные общие примеры).

На заметку! Настоящая глава требует наличия установленного средства SQL Server LocalDB для Visual Studio. Чтобы добавить LocalDB, понадобится запустить программу установки Visual Studio и выбрать вариант SQL Server Express 2016 LocalDB в разделе Individual Components (Индивидуальные компоненты).

В главе 29 будет показано, как выполнять аутентификацию и авторизацию с помощью таких пользовательских учетных записей, а в главе 30 — каким образом выйти за рамки основ и применять ряд более сложных приемов. В табл. 28.1 приведена сводка, позволяющая поместить систему ASP.NET Core Identity в контекст.

Таблица 28.1. Помещение системы ASP.NET Core Identity в контекст

Вопрос	Ответ
Что это такое?	Система ASP.NET Core Identity — это API-интерфейс для управления пользователями и запоминания пользовательских данных в хранилищах, таких как реляционные базы данных, посредством Entity Framework Core
Чем она полезна?	Управление пользователями является важной возможностью для большинства приложений, и ASP.NET Core Identity предлагает готовую хорошо протестированную платформу, которая не требует создания специальных версий распространенных функций

Окончание табл. 28.1

Вопрос	Ответ
Как она используются?	Система Identity используется через службы и промежуточное ПО, добавляемое в классе `Startup`, и посредством классов, которые действуют в качестве шлюзов между приложением и функциональностью Identity
Существуют ли какие-то скрытые ловушки или ограничения?	В Microsoft скомпенсировали жесткость ранних API-интерфейсов управления пользователями ASP.NET, сделав систему Identity настолько гибкой и конфигурируемой, что выяснение того, что возможно и что необходимо, может оказаться проблематичным. В книге мы лишь слегка коснемся поверхности этой глубокой и сложной системы
Существуют ли альтернативы?	Можно было бы реализовать собственные API-интерфейсы, но такая задача обычно требует большого объема работы и вдобавок чревата созданием уязвимостей защиты, если не выполнять ее крайне аккуратно

В табл. 28.2 приведена сводка по главе.

Таблица 28.2. Сводка по главе

Задача	Решение	Листинг
Добавление Identity в проект	Добавьте пакеты и промежуточное ПО для ASP.NET Identity Core и Entity Framework Core, создайте класс пользователя и класс контекста базы данных, а также создайте миграцию базы данных	28.1–28.13
Чтение данных о пользователе	Выполните запрос к базе данных Identity с применением класса контекста	28.14, 28.15
Создание пользовательской учетной записи	Вызовите метод `UserManager.CreateAsync()`	28.18–28.18
Изменение стандартной политики проверки паролей	Установите параметры паролей в классе `Startup`	28.19
Реализация специальной проверки паролей	Реализуйте интерфейс `IPasswordValidator` либо унаследуйте класс от класса `PasswordValidator`	28.20–28.22
Изменение политики проверки учетных записей	Установите параметры пользовательских учетных записей в классе `Startup`	28.23
Реализация специальной проверки учетных записей	Реализуйте интерфейс `IUserValidator` либо унаследуйте класс от класса `UserValidator`	28.24–28.26
Удаление пользовательской учетной записи	Вызовите метод `UserManager.DeleteAsync()`	28.27, 28.28
Редактирование пользовательской учетной записи	Вызовите метод `UserManager.UpdateAsync()`	28.29–28.31

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `Users` с применением шаблона ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core. Примеру приложения требуются инструменты командной строки Entity Framework Core, которые должны быть добавлены в проект путем ручного редактирования файла `.csproj`. Щелкнем правой кнопкой мыши на элементе проекта `Users` в окне Solution Explorer, выберем в контекстном меню пункт Edit Users.csproj file (Редактировать файл `Users.csproj`) и добавим элемент, показанный в листинге 28.1.

Листинг 28.1. Добавление пакета в файле `Users.csproj` из папки `Users`

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <DotNetCliToolReference
       Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
       Version="2.0.0" />
  </ItemGroup>

</Project>
```

В листинге 28.2 приведен код класса `Startup`, который настраивает MVC Framework и компоненты промежуточного ПО, как было описано в главе 14.

Листинг 28.2. Содержимое файла `Startup.cs` из папки `Users`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace Users {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Создание контроллера и представления

Создадим папку `Controllers`, добавим файл класса по имени `HomeController.cs` и поместим в него определение контроллера из листинга 28.3. Этот контроллер будет применяться для описания деталей пользовательских учетных записей и данных, а его метод действия `Index()` передает словарь значений стандартному представлению через метод `View()`.

Листинг 28.3. Содержимое файла `HomeController.cs` из папки `Controllers`

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

namespace Users.Controllers {

  public class HomeController : Controller {

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

Чтобы снабдить контроллер представлением, создадим папку `Views/Home` и добавим в нее файл представления по имени `Index.cshtml` с разметкой, приведенной в листинге 28.4.

Листинг 28.4. Содержимое файла `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>
```

Представление отображает содержимое словаря модели в таблице. Для поддержки представления создадим папку `Views/Shared` и добавим в нее файл по имени `_Layout.cshtml` с разметкой, показанной в листинге 28.5.

Листинг 28.5. Содержимое файла `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <title>Users</title>
  <meta name="viewport" content="width=device-width" />
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  @RenderBody()
</body>
</html>
```

При стилизации HTML-элементов представление полагается на CSS-пакет Bootstrap. Создадим в корневой папке проекта файл `bower.json` с использованием шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел `dependencies` (листинг 28.6).

Листинг 28.6. Добавление пакета Bootstrap в файле `bower.json` из папки `Users`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Последний подготовительный шаг связан с созданием файла `_ViewImports.cshtml` в папке `Views`, в котором настраиваются встроенные вспомогательные функции дескрипторов для применения в представлениях (листинг 28.7).

Листинг 28.7. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Наконец, создадим в папке `Views` файл запуска представления по имени `_ViewStart.cshtml` с содержимым из листинга 28.8. Он обеспечит использование компоновки, созданной в листинге 28.5, всеми представлениями в приложении.

Листинг 28.8. Содержимое файла `_ViewStart.cshtml` из папки `Views`

```
@{
  Layout = "_Layout";
}
```

В результате запуска приложения появится вывод, приведенный на рис. 28.1.

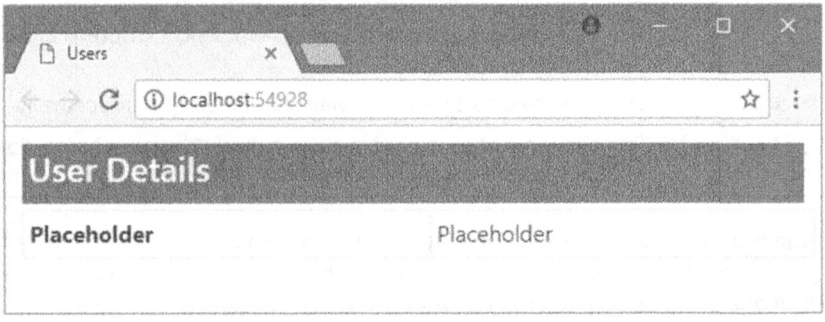

Рис. 28.1. Выполнение примера приложения

Настройка ASP.NET Core Identity

Процесс настройки системы Identity затрагивает почти все части приложения, требуя новых классов моделей, изменений в конфигурации, а также контроллеров и действий для поддержки операций аутентификации и авторизации. В последующих разделах мы пройдем по процессу настройки Identity в базовой конфигурации, чтобы продемонстрировать разнообразные шаги, которые с ним связаны. Задействовать систему Identity в приложении можно многими разными способами, но конфигурация, применяемая в настоящей главе, предусматривает использование самых простых и ходовых параметров.

Создание класса пользователя

Первый шаг заключается в определении класса, предназначенного для представления пользователя в приложении, который называется *классом пользователя*. Класс пользователя является производным от класса `IdentityUser`, определенного в пространстве имен `Microsoft.AspNetCore.Identity.EntityFrameworkCore`. Класс `IdentityUser` обеспечивает базовое представление пользователя, которое можно расширять, добавляя свойства к производному классу, как будет описано в главе 30. В табл. 28.3 перечислены наиболее полезные встроенные свойства, которые определены в `IdentityUser`, включая применяемые в текущей главе.

Таблица 28.3. Свойства класса `IdentityUser`

Имя	Описание
`Id`	Это свойство содержит уникальный идентификатор пользователя
`UserName`	Это свойство возвращает имя пользователя
`Claims`	Это свойство возвращает коллекцию заявок (claim) пользователя, которая будет описана в главе 30
`Email`	Это свойство содержит адрес электронной почты пользователя
`Logins`	Это свойство возвращает коллекцию входов пользователя, используемую для сторонней аутентификации, как объясняется в главе 30
`PasswordHash`	Это свойство возвращает хешированную форму пароля пользователя, которая применяется в разделе "Реализация возможности редактирования" далее в главе
`Roles`	Это свойство возвращает коллекцию ролей, к которым принадлежит пользователь (глава 29)
`PhoneNumber`	Это свойство возвращает телефонный номер пользователя
`SecurityStamp`	Это свойство возвращает значение, которое изменяется, когда меняется удостоверение пользователя, например, из-за смены пароля

Индивидуальные свойства в настоящий момент неважны. Важно то, что класс `IdentityUser` предоставляет доступ к базовой информации о пользователе, включающей имя пользователя, адрес электронной почты, телефонный номер, хеш пароля, членство в ролях и т.д. При желании хранить дополнительные сведения о пользователе придется добавить свойства в класс, производный от `IdentityUser`, который будет использоваться для представления пользователей в приложении.

Чтобы создать класс пользователя для приложения, создадим папку Models и добавим в нее файл класса по имени AppUser.cs с определением класса AppUser, приведенным в листинге 28.9.

Листинг 28.9. Содержимое файла AppUser.cs из папки Models

```
using Microsoft.AspNetCore.Identity;

namespace Users.Models {

  public class AppUser : IdentityUser {
    // Для базовой установки Identity
    // дополнительные члены не требуются
  }
}
```

На данный момент имеется все, что нужно, хотя мы еще вернемся к классу AppUser в главе 30 при рассмотрении способа добавления свойств данных о пользователе, специфичных для приложения.

Конфигурирование импортирования представлений

Несмотря на то что это не относится непосредственно к настройке ASP.NET Core Identity, в следующем разделе мы будем работать с объектами AppUser в представлениях. Чтобы упростить написание представлений, добавим пространство имен Users.Models в файл импортирования представлений (листинг 28.10).

Листинг 28.10. Добавление пространства имен в файле _ViewImports.cshtml из папки Views

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

Создание класса контекста базы данных

Следующий шаг предусматривает создание класса контекста базы данных Entity Framework Core, который оперирует с классом AppUser. Класс контекста унаследован от IdentityDbContext<T>, где T — класс пользователя (AppUser в рассматриваемом примере). Добавим в папку Models файл класса по имени AppIdentityDbContext.cs и определим в нем класс, как показано в листинге 28.11.

Листинг 28.11. Содержимое файла AppIdentityDbContext.cs из папки Models

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace Users.Models {

  public class AppIdentityDbContext : IdentityDbContext<AppUser> {

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

Класс контекста базы данных может быть расширен для изменения способа, которым база данных настраивается и используется, но в случае элементарного приложения ASP.NET Core Identity простого определения класса вполне достаточно, чтобы начать и получить заполнитель для любой настройки в будущем.

На заметку! Не переживайте, если роль упомянутых выше классов не особенно ясна. Если ранее вы не имели дела с инфраструктурой Entity Framework Core, тогда можете трактовать ее как "черный ящик". После того, как основные строительные блоки окажутся на месте (и вы можете их копировать в свои проекты, чтобы все заработало), потребность в их редактировании будет возникать редко.

Конфигурирование настройки строки подключения к базе данных

Первый шаг по конфигурированию ASP.NET Core Identity связан с определением строки подключения к базе данных. По соглашению строка подключения помещается в файл `appsettings.json`, который затем загружается при запуске приложения и доступен в классе `Startup`, как объяснялось в главе 14. Создадим в корневой папке проекта файл `appsettings.json` с использованием шаблона элемента ASP.NET Configuration File (Файл конфигурации ASP.NET) и добавим в него конфигурационные настройки из листинга 28.12.

Листинг 28.12. Содержимое файла `appsettings.json` из папки `Users`

```
{
  "Data": {
    "SportStoreIdentity": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=
IdentityUsers;Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
}
```

В строке подключения присутствует параметр `localdb`, который обеспечивает удобную поддержку баз данных для разработчиков. В качестве имени базы данных указывается `IdentityUsers`.

На заметку! Ширина печатной страницы не позволяет соблюдать правильный формат строки подключения, которая должна выглядеть как одиночная неразрывная строка. В редакторе Visual Studio это не проблема, но в листинге строку пришлось разбить на части. При добавлении строки подключения в свой проект удостоверьтесь, что вводите ее в одной строке.

Имея строку подключения к базе данных, можно обновить класс `Startup` для получения конфигурационных данных (листинг 28.13).

Листинг 28.13. Получение настроек приложения в файле `Startup.cs` из папки `Users`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
```

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;

namespace Users {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));

      services.AddIdentity<AppUser, IdentityRole>()
        .AddEntityFrameworkStores<AppIdentityDbContext>()
        .AddDefaultTokenProviders();

      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseAuthentication();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Для создания базовой установки ASP.NET Core Identity требуются три набора изменений. Сначала настраивается инфраструктура Entity Framework Core, которая предоставляет приложениям MVC службы доступа к данным:

```
...
services.AddDbContext<AppIdentityDbContext>(options =>
  options.UseSqlServer(Configuration[
    "Data:SportStoreIdentity:ConnectionString"]));
...
```

Метод `AddDbContext()` добавляет службы, требующиеся для Entity Framework Core, а метод `UseSqlServer()` настраивает поддержку, необходимую для хранения данных с применением Microsoft SQL Server. Метод `AddDbContext()` позволяет использовать ранее созданный класс контекста базы данных и указать, что он будет копироваться с базой данных SQL Server, строка подключения для которой получается из конфигурации приложения (файл `appsettings.json` в примере приложения).

Понадобится также настроить службы для ASP.NET Core Identity, что делается следующим образом:

```
...
services.AddIdentity<AppUser, IdentityRole>()
  .AddEntityFrameworkStores<AppIdentityDbContext>()
  .AddDefaultTokenProviders();
...
```

Метод AddIdentity() имеет параметры типов, которые указывают класс, применяемый для представления пользователей, и класс, используемый для представления ролей. Здесь задается класс AppUser для пользователей и класс IdentityRole для ролей. Метод AddEntityFrameworkStores() указывает, что система Identity должна использовать инфраструктуру Entity Framework Core для сохранения и извлечения своих данных с применением созданного ранее класса контекста базы данных. Метод AddDefaultTokenProviders() использует стандартную конфигурацию для поддержки операций, требующих маркера, таких как изменение пароля.

Последнее изменение в классе Startup касается добавления системы ASP.NET Core Identity в конвейер обработки запросов. Это позволяет ассоциировать пользовательские учетные данные с запросами на основе cookie-наборов или переписывания URL, т.е. детали пользовательских учетных записей не включаются прямо в HTTP-запросы, отправляемые приложению, или в ответы, которые оно генерирует:

```
...
app.UseAuthentication();
...
```

Создание базы данных Identity

Почти все на месте, и осталось лишь фактически создать базу данных, которая будет использоваться для хранения данных Identity. Откроем окно командной строки или окно PowerShell, перейдем в папку проекта Users (ту, что содержит файл Startup.cs) и введем следующую команду:

```
dotnet ef migrations add Initial
```

Как объяснялось при настройке базы данных для приложения SportsStore, инфраструктура Entity Framework Core управляет изменениями в схемах баз данных через средство, которое называется *миграции*. В случае модификации классов модели, применяемых для генерации схемы, можно сгенерировать файл миграции, который содержит команды SQL, предназначенные для обновления базы данных. Приведенная выше команда создает файлы миграции, которые будут настраивать базу данных Identity.

Когда команда dotnet ef завершится, в окне Solution Explorer появится папка Migrations. При просмотре содержимого файлов из папки Migrations обнаружатся команды SQL, которые будут использоваться для создания начальной базы данных. Чтобы задействовать файлы миграции для создания базы данных, введем такую команду:

```
dotnet ef database update
```

Выполнение команды может занять некоторое время, а после ее завершения база данных будет создана и готова к применению.

Использование ASP.NET Core Identity

После проведения базовой настройки можно приступать к применению ASP.NET Core Identity, чтобы добавить в пример приложения поддержку управления пользователями. В последующих разделах будет продемонстрировано, как использовать API-интерфейс Identity для создания инструментов администрирования, которые делают возможным централизованное управление пользователями.

Инструменты централизованного администрирования пользователей полезны практически во всех приложениях, даже в тех, которые позволяют пользователям создавать и управлять собственными учетными записями. Всегда найдутся пользователи, которым требуется, скажем, пакетное создание учетных записей и поддержка задач, предполагающих инспектирование и корректировку пользовательских данных. С точки зрения настоящей главы инструменты администрирования удобны из-за того, что они организуют множество основных функций управления пользователями в небольшое число классов, превращая их в полезные примеры для демонстрации фундаментальных возможностей системы ASP.NET Core Identity.

Перечисление пользовательских учетных записей

Отправной точкой текущего раздела является перечисление всех пользовательских учетных записей в базе данных, что позволит увидеть эффект от кода, который будет добавлен в приложение позже. Первым делом добавим в папку `Controllers` файл класса по имени `AdminController.cs` и определим в нем контроллер, как показано в листинге 28.14, который будет применяться для реализации функциональности администрирования пользователей.

Листинг 28.14. Содержимое файла `AdminController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;

namespace Users.Controllers {

  public class AdminController : Controller {
    private UserManager<AppUser> userManager;

    public AdminController(UserManager<AppUser> usrMgr) {
      userManager = usrMgr;
    }

    public ViewResult Index() => View(userManager.Users);
  }
}
```

Метод действия `Index()` перечисляет пользователей, управляемых системой Identity; разумеется, в текущий момент пользователи отсутствуют, но вскоре они появятся. Доступ к пользовательским данным осуществляется через объект `UserManager<AppUser>`, который конструктор контроллера получает посредством внедрения зависимостей.

С помощью объекта `UserManager<AppUser>` можно запрашивать хранилище данных. Свойство `Users` возвращает перечисление объектов пользователей (экземпляров класса `AppUser` в рассматриваемом приложении), с которым удобно работать, используя LINQ. Внутри метода действия значение свойства `Users`, которое будет перечислять всех пользователей в базе данных, передается методу `View()`, так что он сможет отобразить детали учетных записей. Чтобы снабдить метод действия представлением, создадим папку `Views/Admin`, добавим файл по имени `Index.cshtml` и поместим в него разметку из листинга 28.15.

Листинг 28.15. Содержимое файла `Index.cshtml` из папки `Views/Admin`

```
@model IEnumerable<AppUser>

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

<table class="table table-sm table-bordered">
  <tr><th>ID</th><th>Name</th><th>Email</th></tr>

  @if (Model.Count() == 0) {
    <tr><td colspan="3" class="text-center">No User Accounts</td></tr>
  } else {
    foreach (AppUser user in Model) {
      <tr>
        <td>@user.Id</td>
        <td>@user.UserName</td>
        <td>@user.Email</td>
      </tr>
    }
  }
</table>

<a class="btn btn-primary" asp-action="Create">Create</a>
```

Представление `Index.cshtml` содержит таблицу, в которой для каждого пользователя предусмотрена строка с колонками, отображающими уникальный идентификатор, имя пользователя и адрес электронной почты. Если пользователи в базе данных отсутствуют, тогда выводится соответствующее сообщение, как показано на рис. 28.2, для чего понадобится запустить приложение и запросить URL вида `/Admin`.

В представление включена ссылка Create (Создать), стилизованная под кнопку, которая нацелена на действие `Create` контроллера `Admin`. Это действие будет поддерживать добавление пользователей.

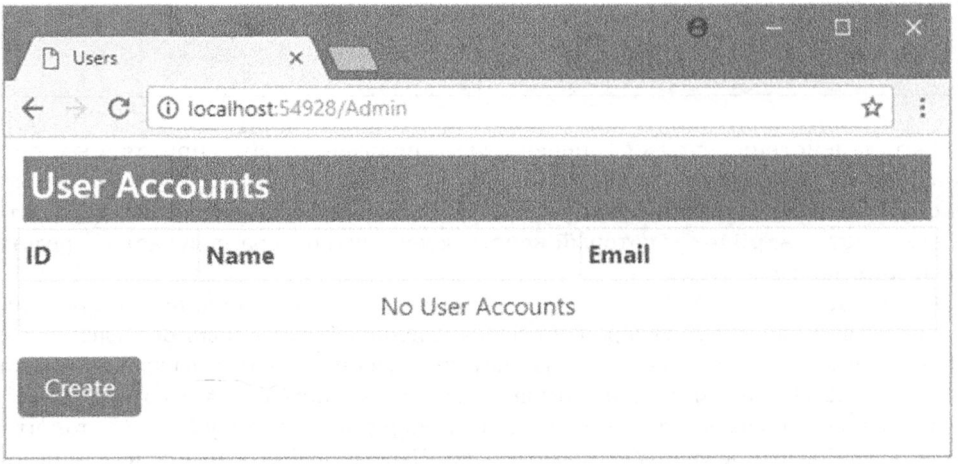

Рис. 28.2. Отображение (пустого) списка пользователей

Переустановка базы данных

Увидеть созданную для системы Identity базу данных можно, открыв окно SQL Server Object Explorer (Проводник объектов SQL Server) в Visual Studio. Если вы впервые используете окно SQL Server Object Explorer, то должны выбрать в меню Tools (Сервис) пункт Connect to Database (Подключиться к базе данных), чтобы сообщить среде Visual Studio о базе данных, с которой нужно работать. В качестве источника данных выберите Microsoft SQL Server, для имени сервера укажите `(localdb)\mssqllocaldb`, оставьте отмеченным флажок Use Windows Authentication (Использовать аутентификацию Windows) и щелкните на стрелке, раскрывающей поле Select Or Enter a Database Name (Выберите или введите имя базы данных). Спустя несколько секунд отобразится список доступных баз данных LocalDB, в котором должна быть возможность выбора базы данных `IdentityUsers`, относящейся к примеру приложения. Щелкните на кнопке OK, после чего в окне SQL Server Object Explorer появится новая запись. Среда Visual Studio запомнит базу данных, так что описанный процесс необходимо выполнить только один раз.

Для просмотра базы данных понадобится раскрыть элемент `(localdb)\mssqllocaldb⇨Databases⇨IdentityUsers` в окне SQL Server Object Explorer. Вы увидите таблицы, которые были созданы файлами миграции, с именами вроде `AspNetUsers` и `AspNetRoles`. После добавления пользователей, как будет показано в следующем разделе, в базу данных можно отправлять запросы для просмотра содержимого таблиц.

Чтобы удалить базу данных, щелкните правой кнопкой мыши на элементе `IdentityUsers` в окне SQL Server Object Explorer и выберите в контекстном меню пункт Delete (Удалить). В диалоговом окне Delete Database (Удаление базы данных) отметьте оба имеющихся флажка и щелкните на кнопке OK для удаления базы данных.

Чтобы заново создать базу данных, откройте окно консоли диспетчера пакетов и введите следующую команду:

```
dotnet ef database update
```

База данных будет воссоздана и готова к применению, когда вы снова запустите приложение.

Создание пользователей

В отношении входных данных, получаемых приложением, будет использоваться проверка достоверности моделей MVC, и легче всего это организовать за счет создания простых моделей представлений для каждой операции, поддерживаемой контроллером. Создадим в папке `Models` файл класса по имени `UserViewModels.cs` с содержимым, приведенным в листинге 28.16.

Листинг 28.16. Содержимое файла `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; }
  }
}
```

Начальная модель называется CreateModel и определяет основные свойства, которые требуются для создания пользовательской учетной записи: имя пользователя, адрес электронной почты и пароль. Атрибут Required из пространства имен System.ComponentModel.DataAnnotations применяется для указания на обязательность значений всех трех свойств, определяемых моделью.

В листинге 28.17 к контроллеру Admin добавлена пара методов действий Create(), на которые нацелена ссылка в представлении Index из предыдущего раздела; они используют стандартный прием для отображения пользователю представления в случае запроса GET и обработки данных формы при поступлении запроса POST.

Листинг 28.17. Определение методов действий Create() в файле AdminController.cs из папки Controllers

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

namespace Users.Controllers {

  public class AdminController : Controller {
    private UserManager<AppUser> userManager;

    public AdminController(UserManager<AppUser> usrMgr) {
      userManager = usrMgr;
    }

    public ViewResult Index() => View(userManager.Users);

    public ViewResult Create() => View();

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

        if (result.Succeeded) {
          return RedirectToAction("Index");
        } else {
          foreach (IdentityError error in result.Errors) {
            ModelState.AddModelError("", error.Description);
          }
        }
      }
      return View(model);
    }
  }
}
```

Важной частью листинга является метод действия Create(), принимающий аргумент CreateModel; этот метод будет вызываться, когда администратор отправляет

данные формы. Свойство ModelState.IsValid применяется для проверки того, что данные содержат обязательные значения, и если содержат, тогда создается новый экземпляр класса AppUser, который передается асинхронному методу UserManager.CreateAsync():

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

Результатом, возвращаемым из метода CreateAsync(), будет объект IdentityResult, который описывает исход операции посредством свойств, перечисленных в табл. 28.4.

Таблица 28.4. Свойства класса IdentityResult

Имя	Описание
Succeeded	Возвращает true, если операция выполнилась успешно
Errors	Возвращает последовательность объектов IdentityError, описывающих ошибки, которые возникли при попытке выполнения операции. Класс IdentityError предлагает свойство Description со сводкой по проблеме

В методе действия Create() с помощью свойства Succeeded выясняется, была ли создана новая запись о пользователе в базе данных. Если свойство Succeeded возвращает true, тогда клиент перенаправляется на действие Index, так что отобразится список пользователей:

```
...
if (result.Succeeded) {
  return RedirectToAction("Index");
} else {
  foreach (IdentityError error in result.Errors) {
    ModelState.AddModelError("", error.Description);
  }
}
...
```

Если свойство Succeeded возвращает false, тогда производится перечисление последовательности объектов IdentityError, которую предоставляет свойство Errors. Свойство Description используется для создания ошибки проверки достоверности на уровне модели с помощью метода ModelState.AddModelError(), как объяснялось в главе 27.

Чтобы снабдить новые методы действий представлением, создадим в папке Views/Admin файл представления по имени Create.cshtml и добавим в него разметку, показанную в листинге 28.18.

Листинг 28.18. Содержимое файла Create.cshtml из папки Views/Admin

```
@model CreateModel

<div class="bg-primary m-1 p-1 text-white"><h4>Create User</h4></div>
<div asp-validation-summary=" All" class="text-danger"></div>
<form asp-action="Create" method="post">
```

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

В представлении `Create.cshtml` нет ничего особо примечательного — в нем присутствует простая форма для сбора значений, которые инфраструктура MVC привяжет к свойствам объекта модели, переданного методу действия `Create()`, а также сводка по возможным ошибкам проверки достоверности.

Тестирование функциональности создания

Чтобы протестировать возможность создания новой пользовательской учетной записи, запустим приложение, перейдем на URL вида `/Admin` и щелкнем на кнопке Create (Создать). Заполним форму значениями из табл. 28.5.

Таблица 28.5. Значения для создания примера пользователя

Имя	Значение
Name	Joe
Email	joe@example.com
Password	Secret123$

Совет. Существуют домены, зарезервированные для целей тестирования, в числе которых `example.com`. Полный список таких доменов доступен по адресу `https://tools.ietf.org/html/rfc2606`.

После ввода значений щелкнем на кнопке Create. Система ASP.NET Core Identity создаст пользовательскую учетную запись, которая будет отображаться после перенаправления браузера на метод действия `Index()`, что видно на рис. 28.3. (Вы получите другой идентификатор, т.к. идентификаторы для пользовательских учетных записей генерируются случайным образом.)

Щелкнем на кнопке Create еще раз и введем в элементах формы те же самые значения из табл. 28.5. На этот раз после отправки формы будет получена ошибка, о которой сообщается в сводке по проверке достоверности модели (рис. 28.4).

Рис. 28.3. Добавление новой пользовательской учетной записи

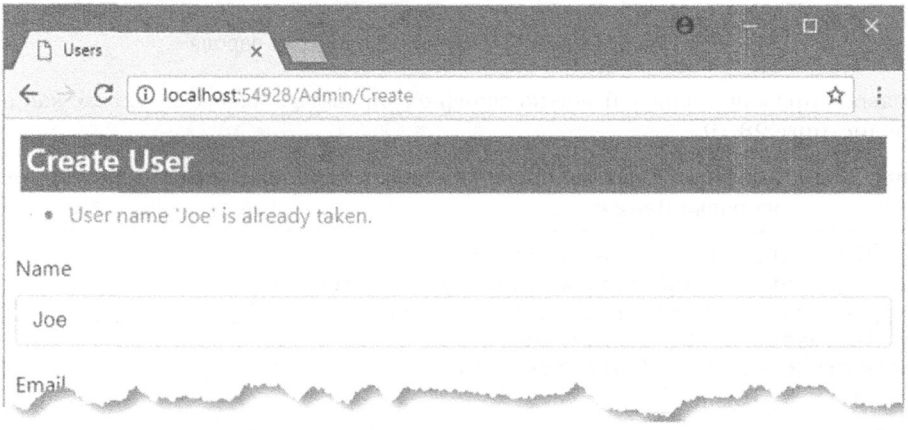

Рис. 28.4. Ошибка при попытке создать нового пользователя

Проверка паролей

Одним из наиболее распространенных требований, особенно в корпоративных приложениях, является принудительное применение политики проверки паролей. Чтобы взглянуть на стандартную политику, запустим приложение, запросим URL вида `/Admin/Create` и заполним форму данными, приведенными в табл. 28.6; их важное отличие от данных из предыдущего раздела связано со значением, вводимым в поле пароля.

Таблица 28.6. Значения для создания примера пользователя

Имя	Значение
Name	Alice
Email	alice@example.com
Password	secret

Когда форма отправляется серверу, система Identity проверяет пароль-кандидат и генерирует ошибки, если он не удовлетворяет требованиям (рис. 28.5).

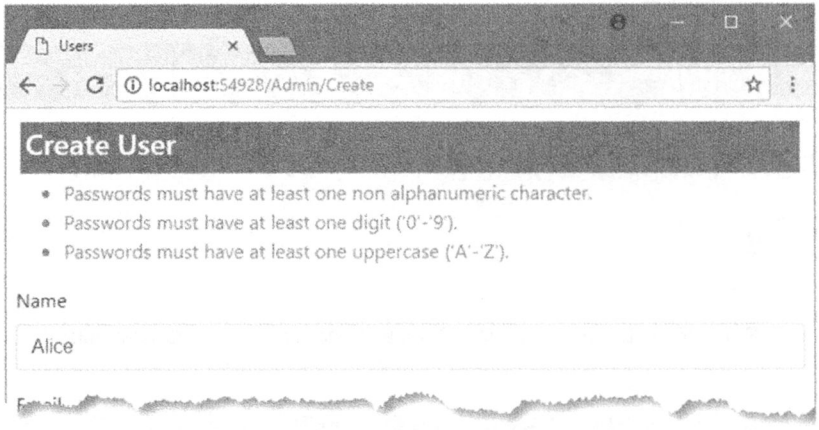

Рис. 28.5. Ошибки в результате проверки пароля

Правила проверки паролей можно сконфигурировать в классе `Startup`, как показано в листинге 28.19.

Листинг 28.19. Конфигурирование правил проверки паролей в файле `Startup.cs` из папки `Users`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;

namespace Users {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));
      services.AddIdentity<AppUser, IdentityRole>(opts => {
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
      }).AddEntityFrameworkStores<AppIdentityDbContext>()
          .AddDefaultTokenProviders();
      services.AddMvc();
    }
```

```
    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseAuthentication();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Метод `AddIdentity()` может использоваться с функцией, принимающей объект `IdentityOptions`, свойство `Password` которого возвращает экземпляр класса `PasswordOptions`. Класс `PasswordOptions` предоставляет свойства для управления политикой проверки паролей, описанные в табл. 28.7.

Таблица 28.7. Свойства класса `PasswordOptions`

Имя	Описание
`RequiredLength`	Это свойство типа `int` применяется для указания минимальной длины паролей
`RequireNonAlphanumeric`	Установка этого свойства типа `bool` в `true` требует, чтобы пароли содержали хотя бы один символ, не являющийся буквой или цифрой
`RequireLowercase`	Установка этого свойства типа `bool` в `true` требует, чтобы пароли содержали хотя бы один символ нижнего регистра
`RequireUppercase`	Установка этого свойства типа `bool` в `true` требует, чтобы пароли содержали хотя бы один символ верхнего регистра
`RequireDigit`	Установка этого свойства типа `bool` в `true` требует, чтобы пароли содержали хотя бы один цифровой символ

В листинге 28.19 указано, что пароли должны иметь минимальную длину в шесть символов, а другие ограничения отключены. Это не должно делаться в реальном проекте, но позволяет получить эффективную демонстрацию. Запустив приложение, перейдя на URL вида `/Admin/Create` и повторив отправку формы, можно обнаружить, что пароль `secret` теперь принимается и новая пользовательская учетная запись успешно создается (рис. 28.6).

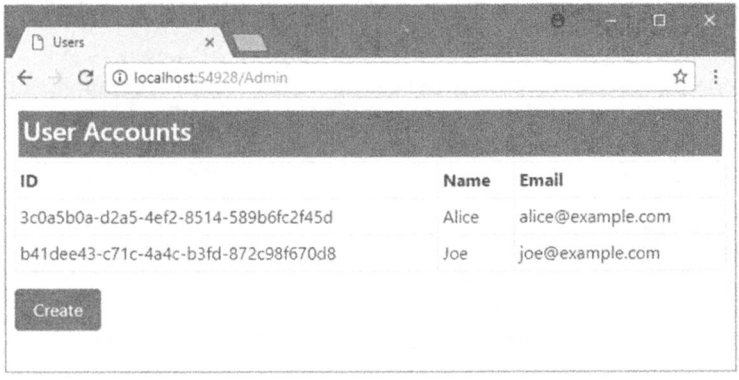

Рис. 28.6. Изменение политики проверки паролей

Реализация специального класса проверки паролей

Для большинства приложений вполне достаточно и встроенной проверки паролей, но может возникнуть необходимость в реализации специальной политики, особенно при разработке корпоративного производственного приложения, в котором сложные политики проверки паролей — обычное явление. Функциональность проверки паролей определяется интерфейсом `IPasswordValidator<T>` из пространства имен `Microsoft.AspNetCore.Identity`, где `T` — класс пользователя, специфичный для приложения (`AppUser` в рассматриваемом примере):

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Identity {
  public interface IPasswordValidator<TUser> where TUser : class {
    Task<IdentityResult> ValidateAsync(UserManager<TUser> manager,
      TUser user, string password);
  }
}
```

Для проверки пароля вызывается метод `ValidateAsync()`, которому передаются данные контекста через объект `UserManager` (позволяющий выполнять запросы к базе данных Identity), представляющий объект пользователя и пароль-кандидат. В результате возвращается объект `IdentityResult`, создаваемый с использованием статического свойства `Success`, если проблемы отсутствуют, или вызывается статический метод `Failed()`, которому передается массив объектов `IdentityError`, описывающих возникшие во время проверки проблемы.

Чтобы продемонстрировать применение специальной политики проверки, создадим папку `Infrastructure` и добавим в нее файл класса по имени `CustomPasswordValidator.cs` с определением из листинга 28.20.

Листинг 28.20. Содержимое файла `CustomPasswordValidator.cs` из папки `Infrastructure`

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
namespace Users.Infrastructure {
  public class CustomPasswordValidator : IPasswordValidator<AppUser> {
    public Task<IdentityResult> ValidateAsync(UserManager<AppUser> manager,
          AppUser user, string password) {
      List<IdentityError> errors = new List<IdentityError>();
      if (password.ToLower().Contains(user.UserName.ToLower())) {
        errors.Add(new IdentityError {
          Code = "PasswordContainsUserName",
          Description = "Password cannot contain username"
        });
      }
      if (password.Contains("12345")) {
        errors.Add(new IdentityError {
          Code = "PasswordContainsSequence",
          Description = "Password cannot contain numeric sequence"
        });
      }
```

```
        return Task.FromResult(errors.Count == 0 ?
          IdentityResult.Success : IdentityResult.Failed(errors.ToArray()));
      }
    }
}
```

Класс `CustomPasswordValidator` проверяет, не содержит ли пароль имя пользователя или последовательность `12345`.

В листинге 28.21 класс `CustomPasswordValidator` регистрируется как средство проверки паролей для объектов `AppUser`.

Листинг 28.21. Регистрация специального класса проверки паролей в файле Startup.cs из папки Users

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;

namespace Users {

  public class Startup {
    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {

      services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();

      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));
      services.AddIdentity<AppUser, IdentityRole>(opts => {
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
      }).AddEntityFrameworkStores<AppIdentityDbContext>()
          .AddDefaultTokenProviders();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseAuthentication();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Чтобы протестировать специальную политику, запустим приложение, запросим URL вида `/Admin/Create` и заполним форму значениями, приведенными в табл. 28.8.

Таблица 28.8. Значения для создания примера пользователя

Имя	Значение
Name	Bob
Email	bob@example.com
Password	bob12345

Пароль в табл. 28.8 нарушает оба правила проверки, навязываемые специальным классом, и вызывает отображение сообщений об ошибках, как показано на рис. 28.7.

Рис. 28.7. Использование специального класса проверки паролей

Можно также реализовать специальную политику проверки, основанную на встроенном классе, который применяется по умолчанию. Стандартный класс называется `PasswordValidator` и определен в пространстве имен `Microsoft.AspNetCore.Identity`.

В листинге 28.22 специальный класс проверки изменен так, что теперь он унаследован от класса `PasswordValidator` и построен на основе поддерживаемых им базовых проверках.

Листинг 28.22. Наследование от встроенного класса проверки в файле CustomPasswordValidator.cs из папки Infrastructure

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
using System.Linq;

namespace Users.Infrastructure {

  public class CustomPasswordValidator : PasswordValidator<AppUser> {
    public override async Task<IdentityResult> ValidateAsync(
        UserManager<AppUser> manager, AppUser user, string password) {
```

```
            IdentityResult result = await base.ValidateAsync(manager,
              user, password);
            List<IdentityError> errors = result.Succeeded ?
              new List<IdentityError>() : result.Errors.ToList();
            if (password.ToLower().Contains(user.UserName.ToLower())) {
              errors.Add(new IdentityError {
                Code = "PasswordContainsUserName",
                Description = "Password cannot contain username"
              });
            }
            if (password.Contains("12345")) {
              errors.Add(new IdentityError {
                Code = "PasswordContainsSequence",
                Description = "Password cannot contain numeric sequence"
              });
            }
            return errors.Count == 0 ? IdentityResult.Success
              : IdentityResult.Failed(errors.ToArray());
        }
    }
}
```

Для тестирования объединенной проверки запустим приложение и заполним данными из табл. 28.9 форму, возвращаемую для URL вида `/Admin/Create`.

Таблица 28.9. Значения для создания примера пользователя

Имя	Значение
Name	Bob
Email	bob@example.com
Password	12345

После отправки формы отобразится сочетание сообщений об ошибках специальной и встроенной проверки (рис. 28.8).

Рис. 28.8. Объединение специальной и встроенной проверки паролей

Проверка деталей, связанных с пользователем

При создании учетной записи проверка выполняется также в отношении имени пользователя и адреса электронной почты. Чтобы взглянуть на работу встроенной проверки, запустим приложение, запросим URL вида `/Admin/Create` и заполним форму данными из табл. 28.10.

Таблица 28.10. Значения для создания примера пользователя

Имя	Значение
`Name`	`Bob!`
`Email`	`alice@example.com`
`Password`	`secret`

В результате отправки формы будет получено сообщение об ошибке (рис. 28.9).

Рис. 28.9. Ошибка при проверке пользовательской учетной записи

Проверку можно конфигурировать в классе `Startup` с использованием свойства `IdentityOptions.User`, которое возвращает экземпляр класса `UserOptions`. Свойства `UserOptions` описаны в табл. 28.11.

Таблица 28.11. Свойства класса `UserOptions`

Имя	Описание
`AllowedUserNameCharacters`	Это свойство типа `string` содержит все разрешенные символы, которые могут применяться в имени пользователя. В стандартном значении указаны символы `a-z`, `A-Z`, `0-9`, переноса, точки, подчеркивания и `@`. Данное свойство не является регулярным выражением, и каждый разрешенный символ должен задаваться явно в строке
`RequireUniqueEmail`	Установка этого свойства типа `bool` в `true` требует, чтобы для новых учетных записей указывались адреса электронной почты, которые не использовались ранее

В листинге 28.23 конфигурация приложения изменена так, чтобы уникальные адреса электронной почты были обязательными и в именах пользователей допускались только алфавитные символы нижнего регистра.

Листинг 28.23. Изменение настроек проверки пользовательских учетных записей в файле `Startup.cs` из папки `Users`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;

namespace Users {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();

      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));

      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();

      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseAuthentication();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

После повторной отправки данных из предыдущего теста легко заметить, что адрес электронной почты сейчас приводит к ошибке, а символы в имени пользователя по-прежнему отклоняются (рис. 28.10).

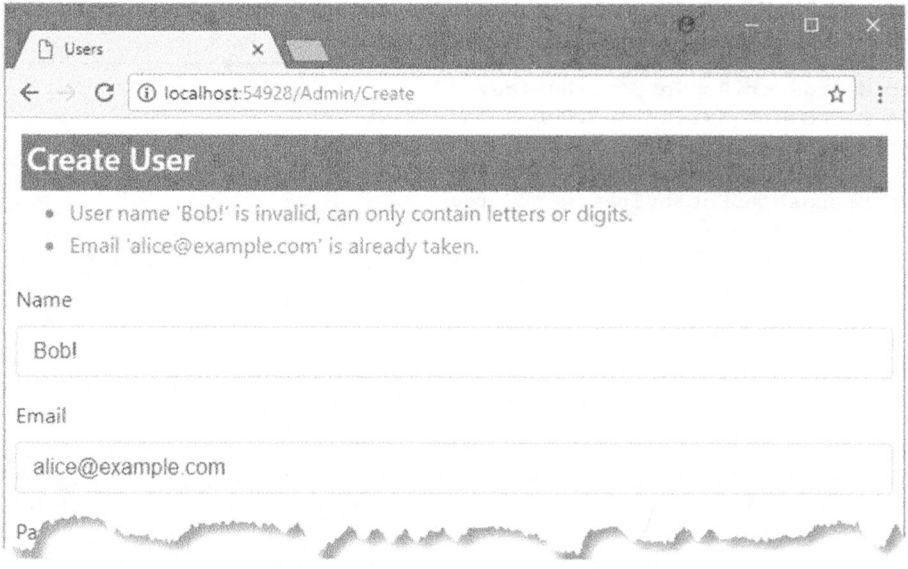

Рис. 28.10. Изменение настроек проверки пользовательских учетных записей

Реализация специальной проверки пользователей

Функциональность проверки задается с помощью интерфейса `IUserValidator<T>`, который определен в пространстве имен `Microsoft.AspNetCore.Identity`:

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Identity {

  public interface IUserValidator<TUser> where TUser : class {
    Task<IdentityResult> ValidateAsync(UserManager<TUser> manager,
      TUser user);
  }
}
```

Метод `ValidateAsync()` вызывается для выполнения проверки. В результате возвращается объект того же самого класса `IdentityResult`, который применялся при проверке паролей. Чтобы продемонстрировать работу специального класса проверки, добавим в папку `Infrastructure` файл класса по имени `CustomUserValidator.cs` с определением, представленным в листинге 28.24.

Листинг 28.24. Содержимое файла `CustomUserValidator.cs` из папки `Infrastructure`

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

```
namespace Users.Infrastructure {
  public class CustomUserValidator : IUserValidator<AppUser> {
    public Task<IdentityResult> ValidateAsync(UserManager<AppUser> manager,
        AppUser user) {
      if (user.Email.ToLower().EndsWith("@example.com")) {
        return Task.FromResult(IdentityResult.Success);
      } else {
        return Task.FromResult(IdentityResult.Failed(new IdentityError {
          Code = "EmailDomainError",
          Description = "Only example.com email addresses are allowed"
        }));
      }
    }
  }
}
```

Класс `CustomUserValidator` проверяет домен адреса электронной почты, чтобы удостовериться в том, что он является частью домена `example.com`. В листинге 28.25 специальный класс регистрируется как средство проверки для объектов `AppUser`.

Листинг 28.25. Регистрация специального класса проверки пользователей в файле `Startup.cs` из папки `Users`

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddTransient<IPasswordValidator<AppUser>,
    CustomPasswordValidator>();
  services.AddTransient<IUserValidator<AppUser>,
    CustomUserValidator>();

  services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
      Configuration["Data:SportStoreIdentity:ConnectionString"]));

  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();

  services.AddMvc();
}
...
```

Для тестирования специального класса проверки запустим приложение и заполним форму, возвращаемую для URL вида `/Admin/Create`, данными из табл. 28.12.

Таблица 28.12. Значения для создания примера пользователя

Имя	Значение
Name	bob
Email	bob@invalid.com
Password	secret

Имя пользователя и пароль успешно проходят проверку, но адрес электронной почты не относится к корректному домену. В результате отправки формы отобразится сообщение об ошибке проверки (рис. 28.11).

Рис. 28.11. Выполнение специальной проверки пользователей

Процесс сочетания встроенной проверки, обеспечиваемой классом `UserValidator<T>`, и специальной проверки аналогичен такому же процессу для проверки паролей (листинг 28.26).

Листинг 28.26. Расширение встроенного класса проверки пользователей в файле `CustomUserValidator.cs` из папки `Infrastructure`

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;

namespace Users.Infrastructure {

  public class CustomUserValidator : UserValidator<AppUser> {

    public override async Task<IdentityResult> ValidateAsync(
        UserManager<AppUser> manager,
        AppUser user) {

      IdentityResult result = await base.ValidateAsync(manager, user);

      List<IdentityError> errors = result.Succeeded ?
        new List<IdentityError>() : result.Errors.ToList();
```

```
        if (!user.Email.ToLower().EndsWith("@example.com")) {
          errors.Add(new IdentityError {
            Code = "EmailDomainError",
            Description = "Only example.com email addresses are allowed"
          });
        }
        return errors.Count == 0 ? IdentityResult.Success
          : IdentityResult.Failed(errors.ToArray());
      }
    }
}
```

Завершение построения средств администрирования

Чтобы завершить построение инструмента администрирования, осталось только реализовать средства для редактирования и удаления пользователей.

В листинге 28.27 показаны изменения, внесенные в файл `Views/Admin/Index.cshtml` для нацеливания на действия `Edit` и `Delete` контроллера `Admin`.

Листинг 28.27. Добавление кнопок редактирования и удаления в файле Index.cshtml из папки Views/Admin

```
@model IEnumerable<AppUser>

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

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

<table class="table table-sm table-bordered">
  <tr><th>ID</th><th>Name</th><th>Email</th></tr>
  @if (Model.Count() == 0) {
    <tr><td colspan="3" class="text-center">No User Accounts</td></tr>
  } else {
    foreach (AppUser user in Model) {
      <tr>
        <td>@user.Id</td><td>@user.UserName</td><td>@user.Email</td>
        <td>
          <form asp-action="Delete" asp-route-id="@user.Id" method="post">
            <a class="btn btn-sm btn-primary" asp-action="Edit"
               asp-route-id="@user.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>
```

Кнопка Delete (Удалить) отправляет форму действию `Delete` контроллера `Admin`, что важно, поскольку при изменении состояния приложения требуется запрос `POST`. Кнопка Edit (Редактировать) является якорным элементом, который будет отправлять запрос `GET`, т.к. первый шаг в процессе редактирования предусматривает отображение текущих данных. Кнопка Edit содержится в элементе `form`, поэтому CSS-стили Bootstrap не укладывают ее вертикально. Кроме того, в представление добавлена сводка по проверке достоверности модели, так что можно легко отображать сообщения об ошибках, которые поступают из остальных средств администрирования.

Реализация возможности удаления

В классе `UserManager<T>` определен метод `DeleteAsync()`, который принимает экземпляр класса пользователя и удаляет его из базы данных. В листинге 28.28 метод `DeleteAsync()` используется для реализации средства удаления в контроллере `Admin`.

Листинг 28.28. Удаление пользователей в файле `AdminController.cs` из папки `Controllers`

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

namespace Users.Controllers {
  public class AdminController : Controller {
    private UserManager<AppUser> userManager;
    public AdminController(UserManager<AppUser> usrMgr) {
      userManager = usrMgr;
    }

    // ...для краткости другие действия не показаны...

    [HttpPost]
    public async Task<IActionResult> Delete(string id) {
      AppUser user = await userManager.FindByIdAsync(id);
      if (user != null) {
        IdentityResult result = await userManager.DeleteAsync(user);
        if (result.Succeeded) {
          return RedirectToAction("Index");
        } else {
          AddErrorsFromResult(result);
        }
      } else {
        ModelState.AddModelError("", "User Not Found");
      }
      return View("Index", userManager.Users);
    }
    private void AddErrorsFromResult(IdentityResult result) {
      foreach (IdentityError error in result.Errors) {
        ModelState.AddModelError("", error.Description);
      }
    }
  }
}
```

Метод действия `Delete()` получает в своем аргументе уникальный идентификатор пользователя и применяет метод `FindByIdAsync()` для нахождения соответствующего объекта пользователя, который можно передать методу `DeleteAsync()`. В качестве результата метод `DeleteAsync()` возвращает объект `IdentityResult`, который обрабатывается таким же образом, как в предшествующих примерах, чтобы обеспечить отображение пользователю сообщений о любых ошибках. Для тестирования функциональности удаления создадим нового пользователя и затем щелкнем на кнопке Delete, расположенной рядом с отображаемыми деталями об этом пользователе в представлении `Index` (рис. 28.12).

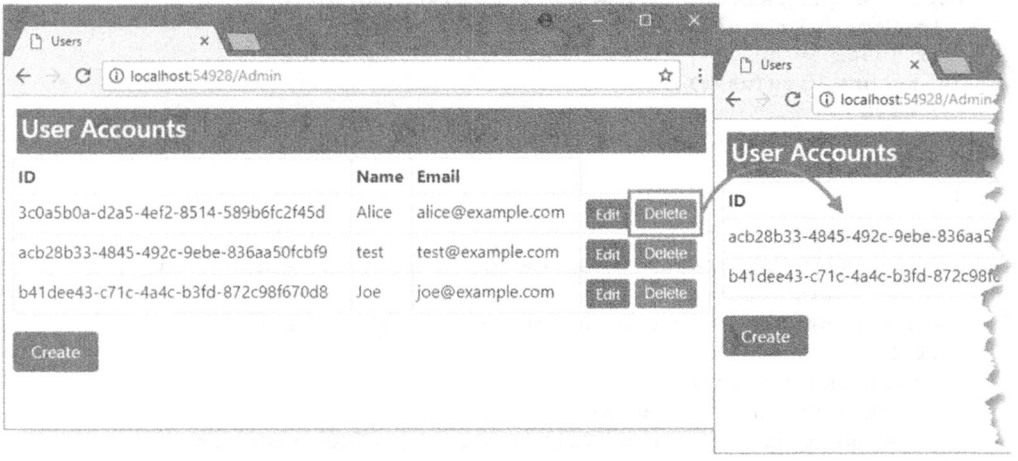

Рис. 28.12. Удаление пользовательской учетной записи

Реализация возможности редактирования

Для завершения инструмента администрирования необходимо добавить поддержку редактирования адреса электронной почты и пароля, которые связаны с пользовательской учетной записью. В настоящий момент имеются только свойства, определяемые пользователями, но в главе 30 будет показано, как расширить схему специальными свойствами. В листинге 28.29 приведен код методов действий `Edit()`, добавленных в контроллер `Admin`.

Листинг 28.29. Добавление действий `Edit` в файле `AdminController.cs` из папки `Controllers`

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

namespace Users.Controllers {

  public class AdminController : Controller {
    private UserManager<AppUser> userManager;
    private IUserValidator<AppUser> userValidator;
    private IPasswordValidator<AppUser> passwordValidator;
    private IPasswordHasher<AppUser> passwordHasher;
```

```
public AdminController(UserManager<AppUser> usrMgr,
       IUserValidator<AppUser> userValid,
       IPasswordValidator<AppUser> passValid,
       IPasswordHasher<AppUser> passwordHash) {
  userManager = usrMgr;
  userValidator = userValid;
  passwordValidator = passValid;
  passwordHasher = passwordHash;
}

// ...для краткости другие действия не показаны...

public async Task<IActionResult> Edit(string id) {
  AppUser user = await userManager.FindByIdAsync(id);
  if (user != null) {
    return View(user);
  } else {
    return RedirectToAction("Index");
  }
}

[HttpPost]
public async Task<IActionResult> Edit(string id, string email,
       string password) {
  AppUser user = await userManager.FindByIdAsync(id);
  if (user != null) {
    user.Email = email;
    IdentityResult validEmail
      = await userValidator.ValidateAsync(userManager, user);
    if (!validEmail.Succeeded) {
      AddErrorsFromResult(validEmail);
    }
    IdentityResult validPass = null;
    if (!string.IsNullOrEmpty(password)) {
      validPass = await passwordValidator.ValidateAsync(userManager,
        user, password);
      if (validPass.Succeeded) {
        user.PasswordHash = passwordHasher.HashPassword(user,
          password);
      } else {
        AddErrorsFromResult(validPass);
      }
    }
    if ((validEmail.Succeeded && validPass == null)
        || (validEmail.Succeeded
        && password != string.Empty && validPass.Succeeded)) {
      IdentityResult result = await userManager.UpdateAsync(user);
      if (result.Succeeded) {
        return RedirectToAction("Index");
      } else {
        AddErrorsFromResult(result);
      }
    }
  } else {
```

```
            ModelState.AddModelError("", "User Not Found");
        }
        return View(user);
    }

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

Действие Edit, на которое нацелены запросы GET, использует строку идентификатора, встроенную в представление Index, чтобы вызвать метод FindByIdAsync() для получения объекта AppUser, представляющего пользователя.

Более сложная реализация действия Edit получает запрос POST и имеет аргументы для идентификатора пользователя, нового адреса электронной почты и пароля. Завершение операции редактирования требует выполнения нескольких задач.

Первая задача связана с проверкой полученных значений. В настоящий момент работа ведется с простым объектом пользователя (хотя в главе 30 объясняется, как настраивать данные, сохраняемые для пользователей), но даже в таком случае нужно проверить пользовательские данные, чтобы обеспечить соблюдение специальных политик, которые были определены в разделах "Проверка паролей" и "Проверка деталей, связанных с пользователем" ранее в главе. Сначала проверяется адрес электронной почты:

```
...
user.Email = email;
IdentityResult validEmail = await userValidator.
ValidateAsync(userManager, user);
if (!validEmail.Succeeded) {
  AddErrorsFromResult(validEmail);
}
...
```

К конструктору контроллера добавлена зависимость от IUserValidator<AppUser>, чтобы можно было проверять новый адрес электронной почты. Обратите внимание, что перед выполнением проверки значение свойства Email должно быть изменено, т.к. метод ValidateAsync() принимает только экземпляры класса пользователя.

Вторая задача связана с изменением пароля, если он был предоставлен. Система ASP.NET Core Identity сохраняет хеши паролей, а не сами пароли. Целью является препятствование похищению паролей. Необходимо получить проверенный пароль и сгенерировать хеш-код, который будет сохранен в базе данных, чтобы пользователь мог быть аутентифицирован, как демонстрируется в главе 29.

Пароли преобразуются в хеши через реализацию интерфейса IPasswordHasher <AppUser>, которая получается за счет объявления зависимости конструктора, распознаваемой посредством внедрения зависимостей. В интерфейсе IPasswordHasher определен метод HashPassword(), который принимает строковый аргумент и возвращает его хешированное значение:

```
...
if (!string.IsNullOrEmpty(password)) {
  validPass = await passwordValidator.ValidateAsync(userManager, user,
    password);
```

```
    if (validPass.Succeeded) {
      user.PasswordHash = passwordHasher.HashPassword(user, password);
    } else {
      AddErrorsFromResult(validPass);
    }
  }
  ...
```

Изменения, внесенные в класс пользователя, не сохраняются в базе данных до тех пор, пока не будет вызван метод `UpdateAsync()`:

```
  ...
  if ((validEmail.Succeeded && validPass == null) || (validEmail.Succeeded
      && password != string.Empty && validPass.Succeeded)) {
    IdentityResult result = await userManager.UpdateAsync(user);
    if (result.Succeeded) {
      return RedirectToAction("Index");
    } else {
      AddErrorsFromResult(result);
    }
  }
  ...
```

Создание представления

Финальным компонентом является представление, которое будет отображать текущие значения для пользователя и позволит отправлять контроллеру новые значения. Добавим в папку `Views/Admin` файл по имени `Edit.cshtml` с содержимым, приведенным в листинге 28.30.

Листинг 28.30. Содержимое файла `Edit.cshtml` из папки `Views/Admin`

```
@model AppUser

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

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

<form asp-action="Edit" method="post">
  <div class="form-group">
    <label asp-for="Id"></label>
    <input asp-for="Id" class="form-control" disabled />
  </div>
  <div class="form-group">
    <label asp-for="Email"></label>
    <input asp-for="Email" class="form-control" />
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input name="password" class="form-control" />
  </div>
  <button type="submit" class="btn btn-primary">Save</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

Представление отображает идентификатор пользователя, который не может быть изменен, как статический текст и предлагает форму редактирования адреса элект-

ронной почты и пароля (рис. 28.13). Обратите внимание, что для элементов пароля вспомогательная функция дескриптора не применяется, т.к. класс пользователя не содержит информации о пароле, поскольку в базе данных хранятся только хешированные значения.

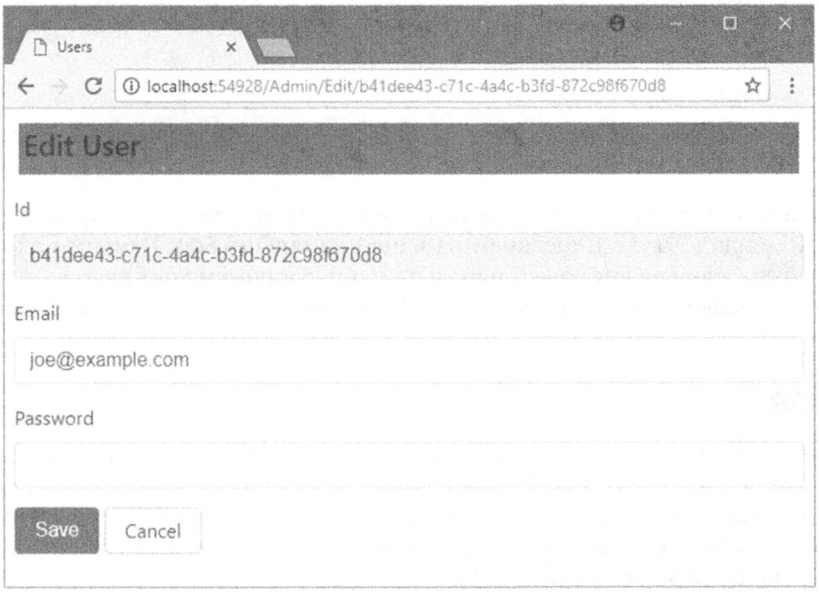

Рис. 28.13. Редактирование пользовательской учетной записи

Осталось закомментировать настройки проверки пользователей в классе `Startup`, чтобы для имен пользователей использовались стандартные символы (листинг 28.31). Ввиду того, что некоторые учетные записи в базе данных были созданы до изменения настроек проверки, отредактировать их не удастся, т.к. имена пользователей не пройдут проверку. А поскольку проверка применяется ко всему объекту пользователя, когда проверяется адрес электронной почты, результатом оказывается пользовательская учетная запись, которую невозможно изменить.

Листинг 28.31. Отключение настроек проверки пользователей в файле `Startup.cs` из папки `Users`

```
...
public void ConfigureServices(IServiceCollection services) {

  services.AddTransient<IPasswordValidator<AppUser>,
    CustomPasswordValidator>();
  services.AddTransient<IUserValidator<AppUser>,
    CustomUserValidator>();

  services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
      Configuration["Data:SportStoreIdentity:ConnectionString"]));

  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();

  services.AddMvc();
}
...
```

Чтобы протестировать возможности редактирования, запустим приложение, запросим URL вида `/Admin` и щелкнем на одной из кнопок Edit. Изменим адрес электронной почты или введем новый пароль (либо предпримем то и другое), после чего щелкнем на кнопке Save (Сохранить) для обновления базы данных и возвращения к URL вида `/Admin`.

Резюме

В главе было показано, как создавать конфигурацию и классы, требующиеся для использования системы ASP.NET Core Identity, а также продемонстрировано, каким образом их можно применять для создания инструмента администрирования пользователей.