ASP.NET Identity 教程

——微软用于实现用户管理、认证与授权的最新技术

作者：Adam Freeman

翻译：林逸

盐城工学院

前言

本系列教程详细、完整、深入地介绍了微软的ASP.NET Identity技术，描述了如何运用ASP.NET Identity实现应用程序的*用户管理*，以及实现应用程序的*认证与授权*等相关技术，译者希望本系列教程能成为掌握ASP.NET Identity技术的一份完整而有价值的资料。读者若是能够按照文章的描述，一边阅读、一边实践、一边理解，定能有意想不到的巨大收获！本教程译自Adam Freeman所著的《Pro ASP.NET MVC 5 Platform》一书中的第13-15章，译文之中如有不当之处敬请读者原谅。

本文也发表于博客园译者的博客上（https://www.cnblogs.com/r01cn），在那里可以隐藏英文进行阅读。本文译者：盐城工学院 r01cn

目录

CHAPTER 13

13 Getting Started with Identity

13 Identity初步

Identity is a new API from Microsoft to manage users in ASP.NET applications. The mainstay for user management in recent years has been ASP.NET Membership, which has suffered from design choices that were reasonable when it was introduced in 2005 but that have aged badly. The biggest limitation is that the schema used to store the data worked only with SQL Server and was difficult to extend without re-implementing a lot of provider classes. The schema itself was overly complex, which made it harder to implement changes than it should have been.
Identity是微软在ASP.NET应用程序中管理用户的一个新的API。近年来用户管理的基石一直是ASP.NET的Membership。Membership在2005年推出时还算是一个合理的选择，但目前看来已经严重过时了。它最大的限制是用以存储数据的架构（Database Schema）只能使用SQL Server，而且难以扩展，除非重新实现大量的提供器类。其数据架构本身也过于复杂，使之比理论上还要难以实现修改。

Microsoft made a couple of attempts to improve Membership prior to releasing Identity. The first was known as *simple membership*, which reduced the complexity of the schema and made it easier to customize user data but still needed a relational storage model. The second attempt was the ASP.NET *universal providers*, which I used in Chapter 10 when I set up SQL Server storage for session data. The advantage of the universal providers is that they use the Entity Framework Code First feature to automatically create the database schema, which made it possible to create databases where access to schema management tools wasn't possible, such as the Azure cloud service. But even with the improvements, the fundamental issues of depending on relational data and difficult customizations remained.
在发布Identity之前，微软曾做过两次改善Membership的尝试。第一个尝试称为*Simple Membership*（简单成员），它降低了数据库架构的复杂性，并使之易于定制用户数据，但仍然需要关系型存储模型。第二个尝试是ASP.NET的*Universal Providers*（通用提供器），第10章在为会话数据建立SQL Server存储库时曾用过它。Universal Providers的好处是，这些提供器使用了Entity Framework的Code First特性，能够自动地创建数据库架构，使之能够在架构管理工具无法访问的情况下，例如Azure云服务，也能够创建数据库。但即使有了改善，其依赖于关系型数据以及难以定制等根本问题仍然存在。

To address both problems and to provide a more modern user management platform, Microsoft has replaced Membership with Identity. As you'll learn in this chapter and Chapters 14 and 15, ASP.NET Identity is flexible and extensible, but it is immature, and features that you might take for granted in a more mature system can require a surprising amount of work.
为了解决这两个问题并提供一个更现代的用户管理平台，微软用Identity取代了Membership。正如将在本章以及第14、15章所了解到的，ASP.NET Identity灵活且可扩展，但它仍不成熟，你在一些更成熟的系统中能够获得的特性，可能需要超常的工作量。

Microsoft has over-compensated for the inflexibility of Membership and made Identity so open and so adaptable that it can be used in just about any way—just as long as you have the time and energy to implement what you require.
微软已经完全弥补了Membership的不灵活性，使Identity十分开放和广泛适应，几乎能够以任何方式进行使用——只要你有时间有能力做出你所需要的实现即可。

In this chapter, I demonstrate the process of setting up ASP.NET Identity and creating a simple user administration tool that manages individual user accounts that are stored in a database.
在本章中，我会演示建立ASP.NET Identity的过程，并创建一个简单的用户管理工具，用以管理存储在数据库中的个别用户账号。

ASP.NET Identity supports other kinds of user accounts, such as those stored using Active Directory, but I don't describe them since they are not used that often outside corporations (where Active Directive implementations tend to be so convoluted that it would be difficult for me to provide useful general examples).
ASP.NET Identity还支持其他类型的用户账号，例如用Active Directory（活动目录）存储的账号，但我不会对其进行描述，因为这种账号通常不会用于公司的外部（这种场合的Active Directory实现往往很复杂，我难以提供有用的通用示例）。

In Chapter 14, I show you how to perform authentication and authorization using those user accounts, and in Chapter 15, I show you how to move beyond the basics and apply some advanced techniques. Table 13-1 summarizes this chapter.
在第14章中，我将演示如何用这些用户账号进行认证与授权，第15章将演示如何进入高级论题，运用一些高级技术。表13-1是本章概要。

Table 13-1. *Chapter Summary*
表13-1. *本章概要*

Problem **问题**	**Solution** **解决方案**	**Listing** **清单号**
Install ASP.NET Identity. 安装ASP.NET Identity	Add the NuGet packages and define a connection string and an OWIN start class in the `Web.config` file. 添加NuGet包，并在Web.config文件中定义一个链接字符串和一个OWIN启动类	1–4
Prepare to use ASP.NET Identity. 使用ASP.NET Identity的准备	Create classes that represent the user, the user manager, the database context, and the OWIN start class. 创建表示用户、用户管理器、数据库上下文的类，以及OWIN类	5–8
Enumerate user accounts. 枚举用户账号	Use the `Users` property defined by the user manager class. 使用由用户管理器类定义的`Users`属性	9, 10
Create user accounts. 创建用户账号	Use the `CreateAsync` method defined by the user manager class. 使用由用户管理器类定义的`CreateAsync`方法	11–13
Enforce a password policy. 强制口令策略	Set the `PasswordValidator` property defined by the user manager class, either using the built-in `PasswordValidator` class or using a custom derivation. 设置由用户管理器类定义的`PasswordValidator`属性，既可以使用内建的`PasswordValidator`类，也可以使用自定义的派生类。	14–16

Validate new user accounts. 验证新的用户账号	Set the `UserValidator` property defined by the user manager class, either using the built-in `UserValidator` class or using a custom derivation. 设置由用户管理器类定义的`UserValidator`属性，既可以使用内建的`UserValidator`类，也可以使用自定义的派生类。	17–19
Delete user accounts. 删除用户账号	Use the `DeleteAsync` method defined by the user manager class. 使用由用户管理器定义的`DeleteAsync`方法	20–22
Modify user accounts. 修改用户账号	Use the `UpdateAsync` method defined by the user manager class. 使用由用户管理器类定义的`UpdateAsync`方法	23–24

13.1 Preparing the Example Project

13.1 准备示例项目

I created a project called `Users` for this chapter, following the same steps I have used throughout this book. I selected the `Empty` template and checked the option to add the folders and references required for an MVC application. I will be using Bootstrap to style the views in this chapter, so enter the following command into the Visual Studio Package Manager Console and press Enter to download and install the NuGet package:

本章根据本书一直采用的同样步骤创建了一个名称为`Users`的项目。在创建过程中选择了“Empty（空）”模板，并在“Add the folders and references（添加文件夹和引用）”中选中了“MVC”复选框。本章将使用Bootstrap来设置视图的样式，因此在Visual Studio的“Package Manager Console（包管理器控制台）”中输入以下命令，并按回车，下载并安装这个NuGet包。

```
Install-Package -version 3.0.3 bootstrap
```

I created a `Home` controller to act as the focal point for the examples in this chapter. The definition of the controller is shown in Listing 13-1. I'll be using this controller to describe details of user accounts and data, and the `Index` action method passes a dictionary of values to the default view via the `View` method.

我创建了`Home`控制器，以作为本章示例的焦点。该控制器的定义如清单13-1所示。此控制器将用来描述用户账号的细节和数据，`Index`动作方法通过`View`方法给默认视图传递了一个字典值。

Listing 13-1. *The Contents of the HomeController.cs File*

清单13-1. *HomeController.cs文件的内容*

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

namespace Users.Controllers {

    public class HomeController : Controller {

        public ActionResult Index() {
            Dictionary<string, object> data
```

```
                = new Dictionary<string, object>();
            data.Add("Placeholder", "Placeholder");
            return View(data);
        }
    }
}
```

I created a view by right-clicking the `Index` action method and selecting Add View from the pop-up menu. I set View Name to Index and set Template to Empty (without model). Unlike the examples in previous chapters, I want to use a common layout for this chapter, so I checked the Use a Layout Page option. When I clicked the Add button, Visual Studio created the `Views/Shared/_Layout.cshtml` and `Views/Home/Index.cshtml` files. Listing 13-2 shows the contents of the `_Layout.cshtml` file.
通过右击`Index`动作方法，并从弹出菜单选择“Add View（添加视图）”，我创建了一个视图。将“View Name（视图名称）”设置为“Index”，并将“Template（模板）”设置为“空（无模型）”。与前面几章的示例不同，本章希望使用一个通用的布局，于是选中了“Use a Layout Page（使用布局页面）”复选框。点击“Add（添加）”按钮后，Visual Studio创建了`Views/Shared/_Layout.cshtml`和`Views/Home/Index.cshtml`文件。清单13-2显示了`_Layout.cshtml`文件的内容。

Listing 13-2. *The Contents of the _Layout.cshtml File*
清单13-2. *_Layout.cshtml文件的内容*

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.min.css" rel="stylesheet" />
    <style>
        .container { padding-top: 10px; }
        .validation-summary-errors { color: #f00; }
    </style>
</head>
<body class="container">
    <div class="container">
        @RenderBody()
    </div>
</body>
</html>
```

Listing 13-3 shows the contents of the `Index.cshtml` file.
清单13-3显示了`Index.cshtml`文件的内容。

Listing 13-3. *The Contents of the Index.cshtml File*

***清单13-3.** Index.cshtml文件的内容*

```
@{
    ViewBag.Title = "Index";
}

<div class="panel panel-primary">
    <div class="panel-heading">User Details</div>
    <table class="table table-striped">
        @foreach (string key in Model.Keys) {
            <tr>
                <th>@key</th>
                <td>@Model[key]</td>
            </tr>
        }
    </table>
</div>
```

To test that the example application is working, select Start Debugging from the Visual Studio Debug menu and navigate to the `/Home/Index` URL. You should see the result illustrated by Figure 13-1.
为了测试该应用程序示例能够工作，从Visual Studio的“Debug（调试）”菜单中选择“Start Debugging（启动调试）”，并导航到`/Home/Index`网址，便可以看到如图13-1所示的结果。

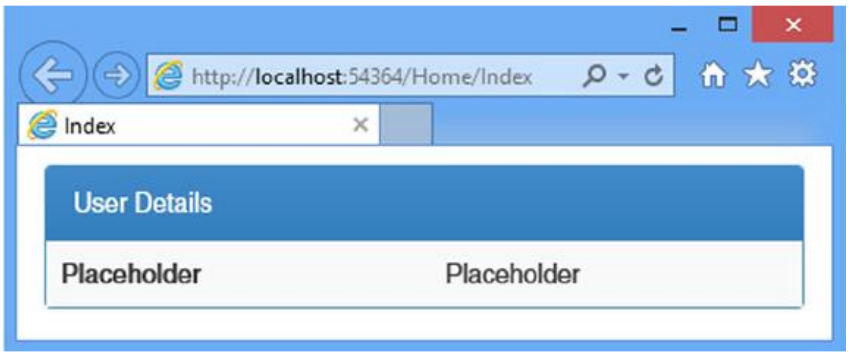

***Figure 13-1.** Testing the example application*
***图13-1.** 测试示例应用程序*

13.2 Setting Up ASP.NET Identity

13.2 建立 ASP.NET Identity

For most ASP.NET developers, Identity will be the first exposure to the Open Web Interface for .NET (OWIN). OWIN is an abstraction layer that isolates a web application from the environment that hosts it. The idea is that the abstraction will allow for greater innovation in the ASP.NET technology stack, more flexibility in the environments that can host ASP.NET applications, and a lighter-weight server infrastructure.
对于大多数ASP.NET开发者而言，Identity将是第一个暴露给OWIN（Open Web Interface for .NET——.NET

开放Web接口）的组件。OWIN是一个将Web应用程序从托管它的环境中独立出来的抽象层。其思想是这个独立出来的抽象层能够使ASP.NET技术堆栈有更大的创新，使托管ASP.NET应用程序的环境有更多的灵活性，并可以是轻量级的服务器架构。

OWIN is an open standard (which you can read at http://owin.org/spec/owin-1.0.0.html). Microsoft has created Project Katana, its implementation of the OWIN standard and a set of components that provide the functionality that web applications require. The attraction to Microsoft is that OWIN/Katana isolates the ASP.NET technology stack from the rest of the .NET Framework, which allows a greater rate of change.
OWIN是一个开放标准（参阅http://owin.org/spec/owin-1.0.html）。微软已经创建了Katana项目，该项目是OWIN标准的实现，并提供了一组Web应用程序所需功能的组件。让微软感兴趣的是OWIN/Katana将ASP.NET技术堆栈从.NET框架的其余部分独立了出来，这带来了更大的修改速率（译者对OWIN还没太了解，这可能是指若按照OWIN的方式开发应用程序，可以使应用程序更易于修改——译者注）。

OWIN developers select the services that they require for their application, rather than consuming an entire platform as happens now with ASP.NET. Individual services—known as *middleware* in the OWIN terminology—can be developed at different rates, and developers will be able to choose between providers for different services, rather than being tied to a Microsoft implementation.
OWIN开发人员为他们的应用程序选择所需的服务，而不是像现在这样单纯地使用ASP.NET平台。个别服务——在OWIN术语中称为“*Middleware*（中间件）”——可能会有不同的开发速率，而开发人员将能够在不同的服务提供商之间进行选择，而不是被绑定在微软实现上。

There is a lot to like about the direction that OWIN and Katana are heading in, but it is in the early days, and it will be some time before it becomes a complete platform for ASP.NET applications. As I write this, it is possible to build Web API and SignalR applications without needing the `System.Web` namespace or IIS to process requests, but that's about all. The MVC framework requires the standard ASP.NET platform and will continue to do so for some time to come.
OWIN和Katana有很多喜欢发展的方向，但它仍处于早期时期，还需要一段时间才能成为ASP.NET应用程序的完整平台。在我编写本书的时候，它已可以在不需要`System.Web`命名空间或者IIS处理请求的情况下，建立Web API和SignalR应用程序了。MVC框架需要标准的ASP.NET平台，因此在一段时间内仍将沿用现在的做法。

The ASP.NET platform and IIS are not going away. Microsoft has been clear that it sees one of the most attractive aspects of OWIN as allowing developers more flexibility in which middleware components are hosted by IIS, and Project Katana already has support for the `System.Web` namespaces. OWIN and Katana are not the end of ASP.NET—rather, they represent an evolution where Microsoft allows developers more flexibility in how ASP.NET applications are assembled and executed.
ASP.NET平台以及IIS不会消失。微软已经清楚地看到，OWIN最有吸引力的一个方面是开发者具有了更大的灵活性，中间组件可以由IIS来托管，而Katana项目已经实现了对`System.Web`命名空间的支持。OWIN和Katana不是ASP.NET的终结——而是预示着一种变革，微软让开发人员能够在ASP.NET应用程序的编译和执行方面更加灵活。

Tip Identity is the first major ASP.NET component to be delivered as OWIN middleware, but it won't be the last. Microsoft has made sure that the latest versions of Web API and SignalR don't depend on the `System.Web` namespaces, and that means that any component intended for use across the ASP.NET family of technologies has to be delivered via OWIN. I get into more detail about OWIN in my *Expert ASP.NET Web API 2 for MVC Developers* book, which will be published by Apress in 2014.
提示：Identity是作为OWIN中间件而交付的第一个主要的ASP.NET组件，但这不是最后一个。微软已经保

证，Web API和SignalR的最新版本不会依赖于System.Web命名空间，这意味着，打算交叉使用ASP.NET家族技术的任何组件，都必须通过OWIN实行交付。关于OWIN，在我的*Expert ASP.NET Web API 2 for MVC Developers*一书中有更详细的论述，该书已由Apress于2014年出版。

OWIN and Katana won't have a major impact on MVC framework developers for some time, but changes are already taking effect—and one of these is that ASP.NET Identity is implemented as an OWIN middleware component. This isn't ideal because it means that MVC framework applications have to mix OWIN and traditional ASP.NET platform techniques to use Identity, but it isn't too burdensome once you understand the basics and know how to get OWIN set up, which I demonstrate in the sections that follow.
OWIN和Katana在一段时间内还不会对MVC框架开发人员发生重大冲击，但变化正在产生影响——其中之一便是ASP.NET Identity要作为OWIN的中间件来实现。这种情况不太理想，因为这意味着，MVC框架的应用程序为了使用Identity，必须将OWIN和传统的ASP.NET平台混搭在一起，这太烦琐了，在你理解了基本概念并且知道如何建立OWIN（以下几小节演示）之后，便会明白。

13.2.1 Creating the ASP.NET Identity Database
13.2.1 创建 ASP.NET Identity 数据库

ASP.NET Identity isn't tied to a SQL Server schema in the same way that Membership was, but relational storage is still the default—and simplest—option, and it is the one that I will be using in this chapter. Although the NoSQL movement has gained momentum in recent years, relational databases are still the mainstream storage choice and are well-understood in most development teams.
ASP.NET Identity并未像Membership那样，被绑定到SQL Server架构，但关系型存储仍是默认的，而且是最简单的选择，这也是本章中将使用的形式。虽然近年来出现了NoSQL运动势头，但关系型数据库仍然是主要的存储选择，而且大多数开发团队都有良好理解。

ASP.NET Identity uses the Entity Framework Code First feature to automatically create its schema, but I still need to create the database into which that schema—and the user data—will be placed, just as I did in Chapter 10 when I created the database for session state data (the universal provider that I used to manage the database uses the same Code First feature).
ASP.NET Identity使用Entity Framework的Code First特性自动地创建它的数据架构（Schema），但我仍然需要创建一个用来放置此数据架构以及用户数据的数据库，就像第10章所做的那样，当时为会话状态数据创建了数据库（用来管理数据库的通用提供器同样使用了Code First特性）。

Tip You don't need to understand how Entity Framework or the Code First feature works to use ASP.NET Identity.
提示：为了使用ASP.NET Identity，不一定要理解Entity Framework或Code First特性的工作机制。

As in Chapter 10, I will be using the `localdb` feature to create my database. As a reminder, `localdb` is included in Visual Studio and is a cut-down version of SQL Server that allows developers to easily create and work with databases.
正如第10章那样，我将使用`localdb`特性来创建数据库。要记住的是，`localdb`是包含在Visual Studio之中的，而且是一个简化版的SQL Server，能够让开发者方便地创建和使用数据库。

Select SQL Server Object Explorer from the Visual Studio View menu and right-click the SQL Server object in the window that appears. Select Add SQL Server from the pop-up menu, as shown in Figure 13-2.
从Visual Studio的“View（视图）”菜单中选择“SQL Server Object Explorer（SQL Server对象资源管理器）”，并在所出现的窗口中右击“SQL Server”对象。从弹出菜单选择“Add SQL Server（添加SQL Server）”，

如图13-2所示。

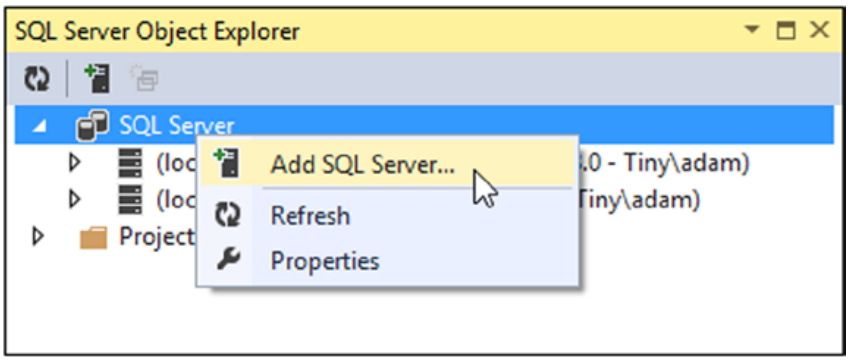

Figure 13-2. Creating a new database connection

***图13-2.** 创建一个新的数据库连接*

Visual Studio will display the Connect to Server dialog. Set the server name to (localdb)\v11.0, select the Windows Authentication option, and click the Connect button. A connection to the database will be established and shown in the SQL Server Object Explorer window. Expand the new item, right-click Databases, and select Add New Database from the pop-up window, as shown in Figure 13-3.

Visual Studio将显示“Connect to Server（连接到服务器）”对话框，将服务器名称设置为(localdb)\v11.0，选择“Windows Authentication（Windows认证）”选项，点击“Connect（连接）”按钮。这将建立一个数据库连接，并显示在“SQL Server对象资源管理器”窗口中。展开这个新项，右击“Databases（数据库）”，并从弹出菜单选择“Add New Database（添加新数据库）”，如图13-3所示。

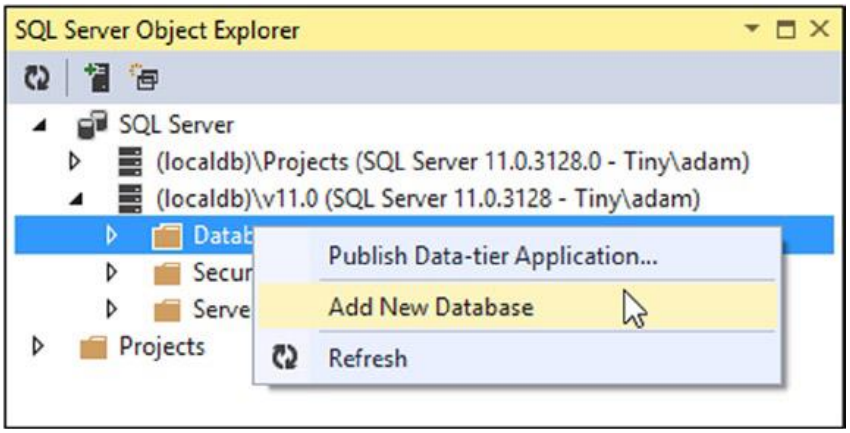

***Figure 13-3.** Adding a new database*

图13-3. 添加新数据库

Set the Database Name option to IdentityDb, leave the Database Location value unchanged, and click the OK button to create the database. The new database will be shown in the Databases section of the SQL connection in the SQL Server Object Explorer.

将“Database Name（数据库名称）”选项设置为“IdentityDb”，不用改变“Database Location（数据库位置）”的值，点击OK按钮创建此数据库。这个新数据库将出现在“SQL Server对象资源管理器”中“SQL连接”的“数据库”小节中。

13.2.2 Adding the Identity Packages
13.2.2 添加 Identity 包

Identity is published as a set of NuGet packages, which makes it easy to install them into any project. Enter the following commands into the Package Manager Console:
Identity是作为一组NuGet包发布的，因此易于将其安装到任何项目。请在“Package Manager Console（包管理器控制台）”中输入以下命令：

```
Install-Package Microsoft.AspNet.Identity.EntityFramework –Version 2.0.0(2.2.1)
Install-Package Microsoft.AspNet.Identity.OWIN -Version 2.0.0(2.2.1)
Install-Package Microsoft.Owin.Host.SystemWeb -Version 2.1.0(3.0.1)
```

Visual Studio can create projects that are configured with a generic user account management configuration, using the Identity API. You can add the templates and code to a project by selecting the MVC template when creating the project and setting the Authentication option to Individual User Accounts. I don't use the templates because I find them too general and too verbose and because I like to have direct control over the contents and configuration of my projects. I recommend you do the same, not least because you will gain a better understanding of how important features work, but it can be interesting to look at the templates to see how common tasks are performed.
Visual Studio能够创建一些使用Identity API的项目，这种项目以“泛型用户账号管理配置”进行配置。你可以给项目添加一些模板和代码，只需在创建项目时选择MVC模板，并将认证选项设置为Individual User Accounts（个别用户账号）。我没有使用这种模板，因为我发现它们太普通，也太混乱，而且我喜欢对我项目中的内容和配置有直接的控制。我建议你也这么做，这不仅让你能够对重要特性的工作机制获得更好的理解，而且，考察模板如何执行常规任务也是有趣的。

13.2.3 Updating the Web.config File
13.2.3 更新 Web.config 文件

Two changes are required to the `Web.config` file to prepare a project for ASP.NET Identity. The first is a connection string that describes the database I created in the previous section. The second change is to define an application setting that names the class that initializes OWIN middleware and that is used to configure Identity. Listing 13-4 shows the changes I made to the `Web.config` file. (I explained how connection strings and application settings work in Chapter 9.)
为了做好项目使用ASP.NET Identity的准备，需要在Web.config文件中做两处修改。第一处是连接字符串，它描述了我在上一小节中创建的数据库。第二处修改是定义一个应用程序设置，它命名对OWIN中间件进行初始化的类，并将它用于配置Identity。清单显示了对Web.config文件的修改（第9章已经解释过连接字符串和应用程序设置）。

Listing 13-4. *Preparing the Web.config File for ASP.NET Identity*
清单13-4. 为ASP.NET Identity准备Web.config文件

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSections>
        <section name="entityFramework"
```

```
            type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
            EntityFramework, Version=6.0.0.0, Culture=neutral,
            PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    </configSections>

    <connectionStrings>
        <add name="IdentityDb" providerName="System.Data.SqlClient"
            connectionString="Data Source=(localdb)\v11.0;
                Initial Catalog=IdentityDb;
                Integrated Security=True;
                Connect  Timeout=15;
                Encrypt=False;TrustServerCertificate=False;
                MultipleActiveResultSets=True"/>
    </connectionStrings>

    <appSettings>
        <add key="webpages:Version" value="3.0.0.0" />
        <add key="webpages:Enabled" value="false" />
        <add key="ClientValidationEnabled" value="true" />
        <add key="UnobtrusiveJavaScriptEnabled" value="true" />
        <add key="owin:AppStartup" value="Users.IdentityConfig" />
    </appSettings>
    <system.web>
        <compilation debug="true" targetFramework="4.5.1" />
        <httpRuntime targetFramework="4.5.1" />
    </system.web>
    <entityFramework>
        <defaultConnectionFactory
            type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
            EntityFramework">
        <parameters>
            <parameter value="v11.0" />
        </parameters>
    </defaultConnectionFactory>
    <providers>
        <provider invariantName="System.Data.SqlClient"
            type="System.Data.Entity.SqlServer.SqlProviderServices,
            EntityFramework.SqlServer" />
        </providers>
    </entityFramework>
</configuration>
```

Caution Make sure you put the `connectionString` value on a single line. I had to break it over several lines to make the listing fit on the page, but ASP.NET expects a single, unbroken string. If in doubt,

download the source code that accompanies this book, which is freely available from www.apress.com.
警告：要确保将connectionString的值放在一行中，这里把它给断开了，是为了让清单适应页面，但ASP.NET要求是单一无断行的字符串。如果有疑问，请下载本书的伴随代码，下载地址：www.apress.com。

OWIN defines its own application startup model, which is separate from the global application class that I described in Chapter 3. The application setting, called owin:AppStartup, specifies a class that OWIN will instantiate when the application starts in order to receive its configuration.
OWIN定义了它自己的应用程序启动模型，与第3章所描述的全局应用程序类是分开的。上述的应用程序设置，名称为owin:AppStartup，指定了一个应用程序启动时OWIN会进行实例化的类，目的是接受它的配置。

Tip Notice that I have set the MultipleActiveResultSets property to true in the connection string. This allows the results from multiple queries to be read simultaneously, which I rely on in Chapter 14 when I show you how to authorize access to action methods based on role membership.
提示：注意，我已经在连接字符串中将MultipleActiveResultSets属性（多活动结果集）设置为true，这样可以从同时读取多个查询形成结果，第14章依赖于这种方式，那时会演示如何根据角色成员授权访问动作方法。

13.2.4 Creating the Entity Framework Classes
13.2.4 创建 Entity Framework 类

If you have used Membership in projects, you may be surprised by just how much initial preparation is required for ASP.NET Identity. The extensibility that Membership lacked is readily available in ASP.NET Identity, but it comes with a price of having to create a set of implementation classes that the Entity Framework uses to manage the database. In the sections that follow, I'll show you how to create the classes needed to get Entity Framework to act as the storage system for ASP.NET Identity.
如果你曾在项目使用过Membership，可能会感觉奇怪，ASP.NET Identity需要的初始化准备怎么这么多。在ASP.NET Identity中具备了Membership所缺乏的可扩展性，但其代价是需要创建一组实现类，由Entity Framework用于管理数据库。在以下小节中，我将演示如何创建所需要的这些类，以便让Entity Framework把它们用于ASP.NET Identity的存储系统。

1. Creating the User Class
1. 创建用户类

The first class to define is the one that represents a user, which I will refer to as the *user class*. The user class is derived from IdentityUser, which is defined in the Microsoft.AspNet.Identity.EntityFramework namespace. IdentityUser provides the basic user representation, which can be extended by adding properties to the derived class, which I describe in Chapter 15. Table 13-2 shows the built-in properties that IdentityUser defines, which are the ones I will be using in this chapter.
第一个要定义的类是表现一个用户的类，我将它称为“*User Class*（用户类）”。这个用户类派生于IdentityUser，它是在Microsoft.AspNet.Identity.EntityFramework命名空间中定义的。IdentityUser提供了基本的用户表示，可以通过在它派生的类中添加属性的办法，对这个类进行扩展，我会在第15章中对此进行描述。表13-2列出了IdentityUser所定义的内建属性（现在流行将“内建”说成“内置”，其实这两者在含义上有很大差别，“内建”完全是自己创建的，而“内置”有可能是别人做的东西拿来放入其中的——译者注），本章将使用这些属性。

Table 13-2. *The Properties Defined by the IdentityUser Class*

表13-2. *IdentityUser类定义的属性*

Name 名称	Description 描述
`Claims`	Returns the collection of claims for the user, which I describe in Chapter 15 返回用户的声明集合，关于声明（Claims）将在第15章描述
`Email`	Returns the user's e-mail address 返回用户的E-mail地址
`Id`	Returns the unique ID for the user 返回用户的唯一ID
`Logins`	Returns a collection of logins for the user, which I use in Chapter 15 返回用户的登录集合，将在第15章中使用
`PasswordHash`	Returns a hashed form of the user password, which I use in the “Implementing the Edit Feature” section 返回哈希格式的用户口令，在“实现Edit特性”中会用到它
`Roles`	Returns the collection of roles that the user belongs to, which I describe in Chapter 14 返回用户所属的角色集合，将在第14章描述
`PhoneNumber`	Returns the user's phone number 返回用户的电话号码
`SecurityStamp`	Returns a value that is changed when the user identity is altered, such as by a password change 返回变更用户标识时被修改的值，例如被口令修改的值
`UserName`	Returns the username

Tip The classes in the `Microsoft.AspNet.Identity.EntityFramework` namespace are the Entity Framework–specific concrete implementations of interfaces defined in the `Microsoft.AspNet.Identity` namespace. `IdentityUser`, for example, is the implementation of the `IUser` interface. I am working with the concrete classes because I am relying on the Entity Framework to store my user data in a database, but as ASP.NET Identity matures, you can expect to see alternative implementations of the interfaces that use different storage mechanisms (although most projects will still use the Entity Framework since it comes from Microsoft).

提示：`Microsoft.AspNet.Identity.EntityFramework`命名空间中的类是，`Microsoft.AspNet.Identity`命名空间中所定义接口的Entity Framework专用的具体实现。例如，`IdentityUser`便是`IUser`接口的实现。我会使用这些具体类，因为我要依靠Entity Framework在数据库中存储我的用户数据，等到ASP.NET Identity变得成熟时，你可能会期望看到这些接口的其他实现，它们使用了不同的存储机制（当然，大多数项目仍然会使用Entity Framework，因为它来自于微软）。

What is important at the moment is that the `IdentityUser` class provides access only to the basic information about a user: the use's name, e-mail, phone, password hash, role memberships, and so on. If I want to store any additional information about the user, I have to add properties to the class that I derive from `IdentityUser` and that will be used to represent users in my application. I demonstrate how to do this in Chapter 15.

目前最重要的是这个`IdentityUser`类只提供了对用户基本信息的访问：用户名、E-mail、电话、哈希口令、角色成员等等。如果希望存储用户的各种附加信息，就需要在`IdentityUser`派生的类上添加属性，并将它用于表示应用程序中的用户，第15章中将演示其做法。

To create the user class for my application, I created a class file called `AppUserModels.cs` in the

Models folder and used it to create the AppUser class, which is shown in Listing 13-5.
为了创建应用程序中的用户类，我在Models文件夹中创建了一个类文件，名称为AppUserModels.cs（注意，这个文件名称错了，应当是AppUser.cs），并用它创建了AppUser类，这个类如清单13-5所示。

Listing 13-5. *The Contents of the AppUser.cs File*
清单13-5. *AppUser.cs文件的内容*

```
using System;
using Microsoft.AspNet.Identity.EntityFramework;

namespace Users.Models {

    public class AppUser : IdentityUser {
        // additional properties will go here
        // 这里将放置附加属性
    }
}
```

That's all I have to do at the moment, although I'll return to this class in Chapter 15, when I show you how to add application-specific user data properties.
以上便是此刻要做的全部工作，第15章会再次讨论这个类，那时会演示如何添加应用程序专用的用户数据属性。

2. Creating the Database Context Class
2. 创建数据库上下文类

The next step is to create an Entity Framework database context that operates on the AppUser class. This will allow the Code First feature to create and manage the schema for the database and provide access to the data it stores. The context class is derived from IdentityDbContext<T>, where T is the user class (AppUser in the example). I created a folder called Infrastructure in the project and added to it a class file called AppIdentityDbContext.cs, the contents of which are shown in Listing 13-6.
下一个步骤是创建Entity Framework数据库的上下文，用于对AppUser类进行操作。这可以用Code First特性来创建和管理数据架构，并对数据库所存储的数据进行访问。这个上下文类派生于IdentityDbContext<T>，其中的T是用户类（即此例中的AppUser）。我在项目中创建了一个文件夹，名称为Infrastructure，并在其中添加了一个类文件，名称为AppIdentityDbContext.cs，其内容如清单13-6所示。

Listing 13-6. *The Contents of the AppIdentityDbContext.cs File*
清单13-6. *AppIdentityDbContext.cs文件的内容*

```
using System.Data.Entity;
using Microsoft.AspNet.Identity.EntityFramework;
using Users.Models;
```

```
namespace Users.Infrastructure {
    public class AppIdentityDbContext : IdentityDbContext<AppUser> {

        public AppIdentityDbContext() : base("IdentityDb") { }

        static AppIdentityDbContext() {
            Database.SetInitializer<AppIdentityDbContext>(new IdentityDbInit());
        }

        public static AppIdentityDbContext Create() {
            return new AppIdentityDbContext();
        }
    }

    public class IdentityDbInit
            : DropCreateDatabaseIfModelChanges<AppIdentityDbContext> {

        protected override void Seed(AppIdentityDbContext context) {
            PerformInitialSetup(context);
            base.Seed(context);
        }

        public void PerformInitialSetup(AppIdentityDbContext context) {
            // initial configuration will go here
            // 初始化配置将放在这儿
        }
    }
}
```

The constructor for the `AppIdentityDbContext` class calls its base with the name of the connection string that will be used to connect to the database, which is `IdentityDb` in this example. This is how I associate the connection string I defined in Listing 13-4 with ASP.NET Identity.
这个`AppIdentityDbContext`类的构造器调用了它的基类，其参数是连接字符串的名称，`IdentityDb`，用于与数据库连接。这是让清单13-4定义的连接字符串与ASP.NET Identity联结起来的方法。

The `AppIdentityDbContext` class also defines a static constructor, which uses the `Database.SetInitializer` method to specify a class that will seed the database when the schema is first created through the Entity Framework Code First feature. My seed class is called `IdentityDbInit`, and I have provided just enough of a class to create a placeholder so that I can return to seeding the database later by adding statements to the `PerformInitialSetup` method. I show you how to seed the database in Chapter 14.
这个`AppIdentityDbContext`类还定义了一个静态的构造器，它使用`Database.SetInitializer`方法指定了一个种植数据库的类（种植数据库的含义是往数据库中植入数据的意思，说穿了，就是用一些数据对数据库进行初始化——译者注），当通过Entity Framework的Code First特性第一次创建数据库架构时，会用到这个类。这个种植类叫做`IdentityDbInit`，而且我已经提供了一个类，创建了一个占位符，

后面可以回过头来在PerformInitialSetup方法中添加语句，就可以种植数据库了。第14章将演示如何种植数据库。

Finally, the AppIdentityDbContext class defines a Create method. This is how instances of the class will be created when needed by the OWIN, using the class I describe in the "Creating the Start Class" section.
最后，AppIdentityDbContext类定义了一个Create方法。这是由OWIN在必要时创建类实例的办法，这个由OWIN使用的类将在"创建启动类"中进行描述。

Note Don't worry if the role of these classes doesn't make sense. If you are unfamiliar with the Entity Framework, then I suggest you treat it as something of a black box. Once the basic building blocks are in place—and you can copy the ones into your chapter to get things working—then you will rarely need to edit them.
注：如果对这些类的意义无法理解，不用担心。如果不熟悉Entity Framework，我建议你将它视为是某种黑箱事物。一旦基础构建块就绪——而且对本章的这些代码进行拷贝——那么就几乎不需要编辑它们了。

---

3. Creating the User Manager Class
3. 创建用户管理器类

One of the most important Identity classes is the *user manager*, which manages instances of the user class. The user manager class must be derived from UserManager<T>, where T is the user class. The UserManager<T> class isn't specific to the Entity Framework and provides more general features for creating and operating on user data. Table 13-3 shows the basic methods and properties defined by the UserManager<T> class for managing users. There are others, but rather than list them all here, I'll describe them in context when I describe the different ways in which user data can be managed.
最重要的Identity类之一是"*User Manager*（用户管理器）"，用来管理用户类实例。用户管理器类必须派生于UserManager<T>，其中T是用户类。这个UserManager<T>类并非是专用于Entity Framework的，而且它提供了很多通用特性，用以创建用户并对用户数据进行操作。表13-3列出了UserManager<T>类为管理用户而定义的基本方法和操作。还有一些其他方法，这里并未全部列出来，我会在适当的情形下对它们进行描述，那时会介绍管理用户数据的不同方式。

Table 13-3. *The Basic Methods and Properties Defined by the UserManager<T> Class*
表13-3. *UserManager<T>类所定义的基本方法和操作*

Name 名称	Description 描述
ChangePasswordAsync(id, old, new)	Changes the password for the specified user. 为指定用户修改口令
CreateAsync(user)	Creates a new user without a password. See Chapter 15 for an example. 创建一个不带口令的新用户，参见第15章示例
CreateAsync(user, pass)	Creates a new user with the specified password. See the "Creating Users" section. 创建一个带有指定口令的新用户，参见"创建用户"
DeleteAsync(user)	Deletes the specified user. See the "Implementing the Delete Feature" section. 删除指定用户，参见"实现Delete特性"
FindAsync(user, pass)	Finds the object that represents the user and authenticates their password. See

	Chapter 14 for details of authentication. 查找代表该用户的对象，并认证其口令，详见第14章的认证细节
`FindByIdAsync(id)`	Finds the user object associated with the specified ID. See the “Implementing the Delete Feature” section. 查找与指定ID相关联的用户对象，参见“实现Delete特性”
`FindByNameAsync(name)`	Finds the user object associated with the specified name. I use this method in the “Seeding the Database” section of Chapter 14. 查找与指定名称相关联的用户对象，第14章“种植数据库”时会用到这个方法
`UpdateAsync(user)`	Pushes changes to a user object back into the database. See the “Implementing the Edit Feature” section. 将用户对象的修改送入数据库，参见“实现Edit特性”
`Users`	Returns an enumeration of the users. See the “Enumerating User Accounts” section. 返回用户枚举，参见“枚举用户账号”

Tip Notice that the names of all of these methods end with `Async`. This is because ASP.NET Identity is implemented almost entirely using C# asynchronous programming features, which means that operations will be performed concurrently and not block other activities. You will see how this works once I start demonstrating how to create and manage user data. There are also synchronous extension methods for each `Async` method. I stick to the asynchronous methods for most examples, but the synchronous equivalents are useful if you need to perform multiple related operations in sequence. I have included an example of this in the “Seeding the Database” section of Chapter 14. The synchronous methods are also useful when you want to call Identity methods from within property getters or setters, which I do in Chapter 15.

提示：请注意方法名以`Async`结尾的那些方法。因为ASP.NET Identity几乎完全是用C#的异步编程特性实现的，这意味着会并发地执行各种操作，而不会阻塞其他活动。在我开始演示如何创建和管理用户数据时，你便会看到这种情况。对于每一个`Async`方法也有相应的同步扩展方法。对于大多数示例，我会坚持这种异步方法。但是，如果你需要按顺序执行一些相关的操作，同步方法是有用的。在第14章的“种植数据库”中就包含了一个这样的例子。当你希望从属性内部的getter或setter代码块中调用Identity方法时，同步方法也是有用的，在第15章中，我就是这么做的。

I added a class file called `AppUserManager.cs` to the `Infrastructure` folder and used it to define the user manager class, which I have called `AppUserManager`, as shown in Listing 13-7.

我在`Infrastructure`文件夹中添加了一个类文件，名称为`AppUserManager.cs`，用它定义了用户管理器类，名称为`AppUserManager`，如清单13-7所示。

Listing 13-7. *The Contents of the AppUserManager.cs File*

清单13-7. *AppUserManager.cs文件的内容*

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Users.Models;
```

```
namespace Users.Infrastructure {
    public class AppUserManager : UserManager<AppUser> {

        public AppUserManager(IUserStore<AppUser> store)
                : base(store) {
        }

        public static AppUserManager Create(
                IdentityFactoryOptions<AppUserManager> options,
                IOwinContext context) {
            AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
            AppUserManager manager = new AppUserManager(new UserStore<AppUser>(db));
            return manager;
        }
    }
}
```

The static `Create` method will be called when Identity needs an instance of the `AppUserManager`, which will happen when I perform operations on user data—something that I will demonstrate once I have finished performing the setup.
在Identity需要一个`AppUserManager`的实例时，将会调用静态的`Create`方法，这种情况将在对用户数据执行操作时发生——也是在完成设置之后要演示的事情。

To create an instance of the `AppUserManager` class, I need an instance of `UserStore<AppUser>`. The `UserStore<T>` class is the Entity Framework implementation of the `IUserStore<T>` interface, which provides the storage-specific implementation of the methods defined by the `UserManager` class. To create the `UserStore<AppUser>`, I need an instance of the `AppIdentityDbContext` class, which I get through OWIN as follows:
为了创建`AppUserManager`类的实例，我需要一个`UserStore<AppUser>`实例。这个`UserStore<T>`类是`IUserStore<T>`接口的Entity Framework实现，它提供了`UserManager`类所定义的存储方法的实现。为了创建`UserStore<AppUser>`，又需要`AppIdentityDbContext`类的一个实例，这是通过OWIN按如下办法得到的：

```
...
AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
...
```

The `IOwinContext` implementation passed as an argument to the `Create` method defines a generically typed `Get` method that returns instances of objects that have been registered in the OWIN start class, which I describe in the following section.
被作为参数传递给`Create`方法的`IOwinContext`实现定义了一个泛型的`Get`方法，它会返回已经在OWIN启动类中注册的对象实例，启动类在以下小节描述。

4. Creating the Start Class
4. 创建启动类

The final piece I need to get ASP.NET Identity up and running is a *start class*. In Listing 13-4, I defined an application setting that specified a configuration class for OWIN, like this:
为了使ASP.NET Identity就绪并能运行，要做的最后一个片段是*Start Class*（启动类）。在清单13-4中，我定义了一个应用程序设置，它为OWIN指定配置类，如下所示：

```
...
<add key="owin:AppStartup" value="Users.IdentityConfig" />
...
```

OWIN emerged independently of ASP.NET and has its own conventions. One of them is that there is a class that is instantiated to load and configure middleware and perform any other configuration work that is required. By default, this class is called `Start`, and it is defined in the global namespace. This class contains a method called `Configuration`, which is called by the OWIN infrastructure and passed an implementation of the `Owin.IAppBuilder` interface, which supports setting up the middleware that an application requires. The `Start` class is usually defined as a partial class, with its other class files dedicated to each kind of middleware that is being used.
OWIN是独立出现在ASP.NET中的，并且有它自己的约定。其中之一就是，为了加载和配置中间件，并执行所需的其他配置工作，需要有一个被实例化的类。默认情况下，这个类叫做`Start`，而且是在全局命名空间中定义的。这个类含有一个名称为`Configuration`的方法，该方法由OWIN基础架构进行调用，并为该方法传递一个`Owin.IAppBuilder`接口的实现，由它支持应用程序所需中间件的设置。`Start`类通常被定义成分部类，还有一些其他的类文件，它们分别专用于要用的每一种中间件。

I freely ignore this convention, given that the only OWIN middleware that I use in MVC framework applications is Identity. I prefer to use the application setting in the `Web.config` file to define a single class in the top-level namespace of the application. To this end, I added a class file called `IdentityConfig.cs` to the `App_Start` folder and used it to define the class shown in Listing 13-8, which is the class that I specified in the `Web.config` folder.
我随意地忽略了这一约定，因为我在MVC框架应用程序中使用的唯一OWIN中间件就是Identity。为了在应用程序的顶级命名空间定义一个类，我喜欢在`Web.config`文件中使用应用程序设置。于是我在`App_Start`文件夹中添加了一个类文件，名称为`IdentityConfig.cs`，并用它定义了如清单13-8所示的类，这是我在`Web.config`文件中指定的一个类。

Listing 13-8. *The Contents of the IdentityConfig.cs File*
清单13-8. *IdentityConfig.cs文件的内容*

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;
using Users.Infrastructure;

namespace Users {
```

```
    public class IdentityConfig {
        public void Configuration(IAppBuilder app) {

            app.CreatePerOwinContext<AppIdentityDbContext>(AppIdentityDbContext.Create);
            app.CreatePerOwinContext<AppUserManager>(AppUserManager.Create);

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

The `IAppBuilder` interface is supplemented by a number of extension methods defined in classes in the `Owin` namespace. The `CreatePerOwinContext` method creates a new instance of the `AppUserManager` and `AppIdentityDbContext` classes for each request. This ensures that each request has clean access to the ASP.NET Identity data and that I don't have to worry about synchronization or poorly cached database data.
`IAppBuilder`接口是由一些扩展方法提供的，这些扩展方法的定义在`Owin`命名空间的一些类中。`CreatePerOwinContext`用于创建`AppUserManager`的新实例，`AppIdentityDbContext`类用于每一个请求。这样保证每一个请求对ASP.NET Identity数据有清晰的访问，我不必为同步时的情况或匮乏的数据缓存而操心。

The `UseCookieAuthentication` method tells ASP.NET Identity how to use a cookie to identity authenticated users, where the options are specified through the `CookieAuthenticationOptions` class. The important part here is the `LoginPath` property, which specifies a URL that clients should be redirected to when they request content without authentication. I have specified `/Account/Login`, and I will create the controller that handles these redirections in Chapter 14.
`UseCookieAuthentication`方法告诉ASP.NET Identity如何用cookie去标识已认证的用户，以及通过`CookieAuthenticationOptions`类指定的选项在哪儿。这里重要的部分是`LoginPath`属性，它指定了一个URL，这是未认证客户端请求内容时要重定向的地址。我在其中指定了`/Account/Login`，将在第14章创建这个控制器来处理这些重定向。

13.3 Using ASP.NET Identity

13.3 使用 ASP.NET Identity

Now that the basic setup is out of the way, I can start to use ASP.NET Identity to add support for managing users to the example application. In the sections that follow, I will demonstrate how the Identity API can be used to create administration tools that allow for centralized management of users. Table 13-4 puts ASP.NET Identity into context.
现在，已经完成了基本设置，可以开始使用ASP.NET Identity在示例应用程序中添加对用户管理的支持了。在以下几小节中，我将演示如何将Identity API用于创建管理工具，这样能够集中化地管理用户。表13-4

说明了ASP.NET Identity的情形。

Table 13-4. *Putting Content Caching by Attribute in Context*

表13-4. *通过注解属性设置内容缓存的情形（这个标题作者写错了——译者注）*

Question **问题**	**Answer** **回答**
What is it? 什么ASP.NET Identity?	ASP.NET Identity is the API used to manage user data and perform authentication and authorization. ASP.NET Identity是用来管理用户数据并执行认证和授权的API
Why should I care? 为何要关心它？	Most applications require users to create accounts and provide credentials to access content and features. ASP.NET Identity provides the facilities for performing these operations. 大多数应用程序都需要用户创建账号，并提供凭据去访问内容和功能。ASP.NET Identity提供了执行这些操作的工具
How is it used by the MVC framework? 在MVC框架中如何使用它？	ASP.NET Identity isn't used directly by the MVC framework but integrates through the standard MVC authorization features. ASP.NET Identity不是由MVC框架直接使用的，但它集成了标准的MVC授权特性

13.3.1 Enumerating User Accounts
13.3.1 枚举用户账号

Centralized user administration tools are useful in just about all applications, even those that allow users to create and manage their own accounts. There will always be some customers who require bulk account creation, for example, and support issues that require inspection and adjustment of user data. From the perspective of this chapter, administration tools are useful because they consolidate a lot of basic user management functions into a small number of classes, making them useful examples to demonstrate the fundamental features of ASP.NET Identity.

集中化的用户管理工具在几乎所有应用程序中都是有用的，即使是那些允许用户创建并管理自己账号的情况也是如此。例如，总会有这样一些客户，他们需要大量创建账号，并支持需要检查和调整用户数据的问题。根据本章的观点，管理工具是有用的，因为它们将许多基本的用户管理功能整合到了少量的几个类之中，这对于演示ASP.NET Identity的基本特性是很有用的。

I started by adding a controller called Admin to the project, which is shown in Listing 13-9, and which I will use to define my user administration functionality.

首先，在项目中添加一个控制器，名称为Admin，如清单13-9所示，我将用它定义我的用户管理功能。

Listing 13-9. *The Contents of the AdminController.cs File*

清单13-9. AdminController.cs文件的内容

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

namespace Users.Controllers {
    public class AdminController : Controller {
```

```
        public ActionResult Index() {
            return View(UserManager.Users);
        }

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

The Index action method enumerates the users managed by the Identity system; of course, there aren't any users at the moment, but there will be soon. The important part of this listing is the way that I obtain an instance of the AppUserManager class, through which I manage user information. I will be using the AppUserManager class repeatedly as I implement the different administration functions, and I defined the UserManager property in the Admin controller to simplify my code.
动作方法枚举由Identity系统管理的用户，当然，此刻还没有任何用户，但马上就会有了。该清单重要的部分是我获得类实例的方式，通过它，我可以管理用户信息。在我实现不同的管理功能时，我会反复使用AppUserManager类，而且，我在Admin控制器中定义了UserManager属性，以便简化代码。

The Microsoft.Owin.Host.SystemWeb assembly adds extension methods for the HttpContext class, one of which is GetOwinContext. This provides a per-request context object into the OWIN API through an IOwinContext object. The IOwinContext isn't that interesting in an MVC framework application, but there is another extension method called GetUserManager<T> that is used to get instances of the user manager class.
Microsoft.Owin.Host.SystemWeb程序集为HttpContext类添加了一些扩展方法，其中之一便是GetOwinContext。它通过一个IOwinContext对象，将基于每请求的上下文对象提供给WOIN API。MVC框架应用程序对IOwinContext不感兴趣，但是有另外一个扩展方法，叫做GetUserManager<T>，可以用来得到用户管理器类的实例。

Tip As you may have gathered, there are lots of extension methods in ASP.NET Identity; overall, the API is something of a muddle as it tries to mix OWIN, abstract Identity functionality, and the concrete Entity Framework storage implementation.
提示：正如你所推断的一样，ASP.NET Identity中有许多扩展方法，总体而言，这个API是一种混合体，它试图将WOIN、抽象Identity功能以及具体的Entity Framework存储实现混合在一起。

I called the GetUserManager with a generic type parameter to specify the AppUserManager class that I created earlier in the chapter, like this:
我用一个泛型参数调用了GetUserManager，用以指定本章前面创建的AppUserManager类，如下所示：

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

Once I have an instance of the AppUserManager class, I can start to query the data store. The AppUserManager.Users property returns an enumeration of user objects—instances of the AppUser class in my application—which can be queried and manipulated using LINQ.
一旦有了AppUserManager类的实例，便可以开始查询数据存储了。AppUserManager.Users属性返回了一个用户对象的枚举——应用程序中AppUser类的实例——于是可以用LINQ对这个枚举进行查询和维护。

In the Index action method, I pass the value of the Users property to the View method so that I can list details of the users in the view. Listing 13-10 shows the contents of the Views/Admin/Index.cshtml file that I created by right-clicking the Index action method and selecting Add View from the pop-up menu.
在Index动作方法中，给View方法传递了Users属性的值，以便能够在视图列出用户的细节。清单13-10显示了Views/Admin/Index.cshtml文件的内容，这是通过右击Index动作方法，并从弹出菜单选择“Add View（添加视图）”而创建的。

Listing 13-10. *The Contents of the Index.cshtml File in the /Views/Admin Folder*
清单13-10. */Views/Admin文件夹中Index.cshtml文件的内容*

```
@using Users.Models
@model IEnumerable<AppUser>
@{
    ViewBag.Title = "Index";
}

<div class="panel panel-primary">
    <div class="panel-heading">
        User Accounts
    </div>
    <table class="table table-striped">
        <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>
</div>
@Html.ActionLink("Create", "Create", null, new { @class = "btn btn-primary" })
```

This view contains a table that has rows for each user, with columns for the unique ID, username, and

e-mail address. If there are no users in the database, then a message is displayed, as shown in Figure 13-4.
这个视图含有一个表格，每个用户为一行，带有唯一ID、用户名以及E-mail地址的表格列。如果数据库中没有用户，那么会显示一条消息（图中的“No User Accounts”消息——译者注），如图13-4所示。

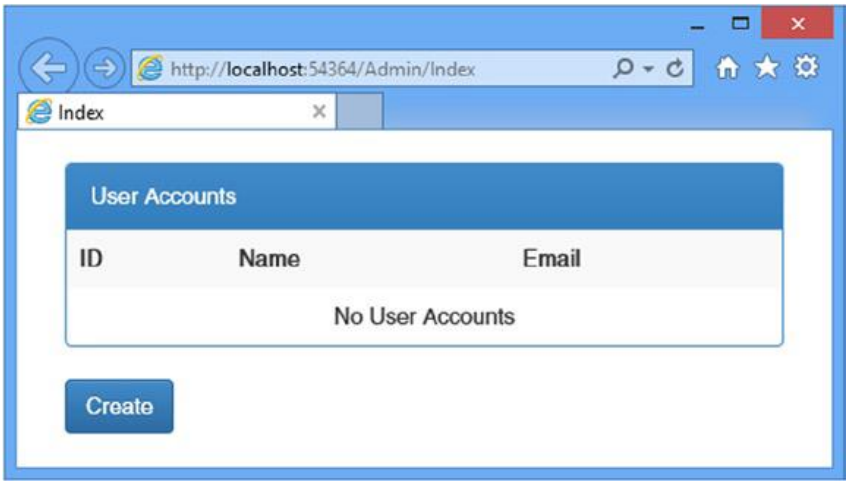

Figure 13-4. *Display the (empty) list of users*
图13-4. *显示用户列表（空）*

I included a `Create` link in the view (which I styled as a button using Bootstrap) that targets the `Create` action on the `Admin` controller. I'll implement this action shortly to support adding users.
在视图中有一个`Create`的链接（其样式用Bootstrap形成了一个按钮），它的目标是`Admin`控制器中的`Create`动作。为了支持添加用户，我会很快实现这个动作。

RESETTING THE DATABASE

重置数据库

When you start the application and navigate to the `/Admin/Index` URL, it will take a few moments before the contents rendered from the view are displayed. This is because the Entity Framework has connected to the database and determined that there is no schema defined. The Code First feature uses the classes I defined earlier in the chapter (and some which are contained in the Identity assemblies) to create the schema so that it is ready to be queried and to store data.
当启动应用程序并导航到`/Admin/Index` URL时，在视图渲染的内容被显示出来之前，会花一些时间。这是因为Entity Framework已经链接到数据库，并发现此时尚未定义数据库架构。Code First特性使用本章前面定义的类（以及包含在Identity程序集中的类）创建该架构，以便做好查询和存储数据的准备。

You can see the effect by opening the Visual Studio SQL Server Object Explorer window and expanding entry for the `IdentityDB` database schema, which will include tables with names such as `AspNetUsers` and `AspNetRoles`.
通过打开Visual Studio的“SQL Server Object Explorer（SQL Server对象资源管理器）”窗口，并展开`IdentityDB`数据库架构，便可以看到其中包含了一些数据表，如`AspNetUsers`和`AspNetRoles`等等 。

To delete the database, right-click the `IdentityDb` item and select `Delete` from the pop-up menu. Check both of the options in the Delete Database dialog and click the OK button to delete the database.

要删除该数据库，右击IdentityDb条目，并从弹出菜单选择“Delete（删除）”。选中“Delete Database（删除数据库）”对话框中的复选框，并点击OK，以删除该数据库。

Right-click the Databases item, select Add New Database (as shown in Figure 13-3), and enter IdentityDb in the Database Name field. Click OK to create the empty database. The next time you start the application and navigate to the Admin/Index URL, the Entity Framework will detect that there is no schema and re-create it.
右击“Databases（数据库）”条目，选择“Add New Database（添加新数据库）”（如图13-3所示），并在“Database Name（数据库名）”字段中输入IdentityDb，点击OK，便可以创建空数据库。下一次启动应用程序，并导航到Admin/Index URL时，Entity Framework又将侦测到没有数据库架构，又会重新创建此数据库。

13.3.2 Creating Users
13.3.2 创建用户

I am going to use MVC framework model validation for the input my application receives, and the easiest way to do this is to create simple view models for each of the operations that my controller supports. I added a class file called UserViewModels.cs to the Models folder and used it to define the class shown in Listing 13-11. I'll add further classes to this file as I define models for additional features.
对于应用程序接收的输入，我打算使用MVC架构的模型验证，最容易的做法是为控制器所支持的每一种操作创建一个简单的视图模型。我在Models文件夹中添加了一个类文件，名称为UserViewModels.cs，并用它定义了如清单13-11所示的类。随着我为其他特性定义模型，会进一步地在这个文件中添加一些类。

Listing 13-11. *The Contents of the UserViewModels.cs File*
清单13-11. *UserViewModels.cs文件的内容*

```
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; }
    }
}
```

The initial model I have defined is called CreateModel, and it defines the basic properties that I require to create a user account: a username, an e-mail address, and a password. I have used the Required attribute from the System.ComponentModel.DataAnnotations namespace to denote that values are required for all three properties defined in the model.

我所定义的第一个模型叫做CreateModel，它定义了创建用户账号所需要的基本属性：用户名、E-mail地址以及口令。这个模型中所定义的所有属性都使用了System.ComponentModel.DataAnnotations命名空间中的Required注释属性，用以说明这些值是必须的。

I added a pair of Create action methods to the Admin controller, which are targeted by the link in the Index view from the previous section and which uses the standard controller pattern to present a view to the user for a GET request and process form data for a POST request. You can see the new action methods in Listing 13-12.
我在Admin控制器中添加了一对Create动作方法，它们是上一小节Index视图中的那个链接所指向的目标，而且，对于GET请求以及处理表单数据的POST请求，使用的是标准的控制器模式将视图表现给用户。可以从清单13-12中看到这两个新的动作方法。

Listing 13-12. *Defining the Create Action Methods in the AdminController.cs File*
清单13-12. *在AdminController.cs文件中定义Create动作方法*

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

namespace Users.Controllers {
    public class AdminController : Controller {

        public ActionResult Index() {
            return View(UserManager.Users);
        }

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

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

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

```
            }
            return View(model);
        }

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

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

The important part of this listing is the `Create` method that takes a `CreateModel` argument and that will be invoked when the administrator submits their form data. I use the `ModelState.IsValid` property to check that the data I am receiving contains the values I require, and if it does, I create a new instance of the `AppUser` class and pass it to the `UserManager.CreateAsync` method, like this:
该清单重要的部分是以`CreateModel`为参数并在管理员递交其表单数据时调用的那个`Create`动作方法（`POST`版的`Create`动作方法——译者注）。我用`ModelState.IsValid`属性检查所接收到的数据是否包含了我所需要的数据，如果是，便创建一个`AppUser`类的新实例，并将它传递给`UserManager.CreateAsync`方法，如下所示：

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

The result from the `CreateAsync` method is an implementation of the `IdentityResult` interface, which describes the outcome of the operation through the properties listed in Table 13-5.
`CreateAsync`方法的结果是一个`IdentityResult`接口的实现，它通过表13-5中的属性描述操作的输出。

Table 13-5. *The Properties Defined by the IdentityResult Interface*
表13-5. *IdentityResult接口所定义的属性*

Name 名称	Description 描述
`Errors`	Returns a string enumeration that lists the errors encountered while attempting the operation 返回一个字符串枚举，其中列出了尝试操作期间所遇到的错误
`Succeeded`	Returns true if the operation succeeded 在操作成功时返回true

USING THE ASP.NET IDENTITY ASYNCHRONOUS METHODS

使用 ASP.NET Identity 异步方法

You will notice that all of the operations for manipulating user data, such as the `UserManager.CreateAsync` method I used in Listing 13-12, are available as asynchronous methods. Such methods are easily consumed with the `async` and `await` keywords. Using asynchronous Identity methods allows your action methods to be executed asynchronously, which can improve the overall throughput of your application.

你会注意到，维护用户数据的所有操作，如清单13-12中所用到的`UserManager.CreateAsync`方法，都可以作为异步方法来使用。这种方法很容易与`async`和`await`关键字一起使用。使用异步的Identity方法让你的动作方法能够异步执行，这可以从整体上改善应用程序。

However, you can also use synchronous extension methods provided by the Identity API. All of the commonly used asynchronous methods have a synchronous wrapper so that the functionality of the `UserManager.CreateAsync` method can be called through the synchronous `UserManager.Create` method. I use the asynchronous methods for preference, and I recommend you follow the same approach in your projects. The synchronous methods can be useful for creating simpler code when you need to perform multiple dependent operations, so I used them in the “Seeding the Database” section of Chapter 14 as a demonstration.

然而，你也可以使用那些由Identity API提供的同步扩展方法。所有常用的异步方法都有一个同步的封装程序，因此`UserManager.CreateAsync`方法的功能可以通过同步的`UserManager.Create`方法进行调用。我更喜欢使用异步方法，而且我建议在你的项目中遵循同样的方式。当需要执行多个有依赖的操作时，为了形成简单的代码，同步方法可能是有用的，因此，作为一个演示，我在第14章的“种植数据库”小节中使用了它们。

I inspect the `Succeeded` property in the `Create` action method to determine whether I have been able to create a new user record in the database. If the `Succeeded` property is `true`, then I redirect the browser to the `Index` action so that list of users is displayed:

在`Create`动作方法中，我检测了`Succeeded`属性，以确定是否能够在数据库创建一条新的用户记录。如果`Succeeded`属性为`true`，那么便将浏览器重定向到`Index`动作，以便显示用户列表：

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

If the `Succeeded` property is `false`, then I call the `AddErrorsFromResult` method, which enumerates the messages from the `Errors` property and adds them to the set of model state errors, taking advantage of the MVC framework model validation feature. I defined the `AddErrorsFromResult` method because I will have to process errors from other operations as I build the functionality of my administration

controller. The last step is to create the view that will allow the administrator to create new accounts. Listing 13-13 shows the contents of the `Views/Admin/Create.cshtml` file.
如果`Succeeded`属性为`false`，那么便调用`AddErrorsFromResult`方法，该方法枚举了`Errors`属性中的消息，并将它们添加到模型状态的错误消息集合中，此过程利用了MVC框架的模型验证特性。我定义`AddErrorsFromResult`方法，是因为随着进一步地构建这个管理控制器的功能，还必须处理来自其他操作的错误消息。最后一步是创建视图，让管理员创建新账号。清单13-13显示了文件的`Views/Admin/Create.cshtml`内容。

Listing 13-13. *The Contents of the Create.cshtml File*
清单13-13. *Create.cshtml文件的内容*

```
@model Users.Models.CreateModel
@{ ViewBag.Title = "Create User";}

<h2>Create User</h2>
@Html.ValidationSummary(false)
@using (Html.BeginForm()) {
    <div class="form-group">
        <label>Name</label>
        @Html.TextBoxFor(x => x.Name, new { @class = "form-control"})
    </div>
    <div class="form-group">
        <label>Email</label>
        @Html.TextBoxFor(x => x.Email, new { @class = "form-control" })
    </div>
    <div class="form-group">
        <label>Password</label>
        @Html.PasswordFor(x => x.Password, new { @class = "form-control" })
    </div>
    <button type="submit" class="btn btn-primary">Create</button>
    @Html.ActionLink("Cancel", "Index", null, new { @class = "btn btn-default"})
}
```

There is nothing special about this view—it is a simple form that gathers values that the MVC framework will bind to the properties of the model class that is passed to the `Create` action method.
该视图没有什么特别的地方——只是一个简单的表单，用来收集一些值，MVC框架会将它们绑定到模型类的属性上，然后传递给`Create`动作方法。

1. Testing the Create Functionality
1. 测试 Create 功能

To test the ability to create a new user account, start the application, navigate to the `/Admin/Index` URL, and click the `Create` button. Fill in the form with the values shown in Table 13-6.
为了测试创建新用户账号的能力，启动应用程序，导航到`/Admin/Index`网址，并点击`Create`按钮。用一些值填充表单，如表13-6所示。

Table 13-6. *The Values for Creating an Example User*

表13-6. *创建示例用户的值*

Name **名称**	**Value** **值**
Name	Joe
Email	joe@example.com
Password	Secret

Tip Although not widely known by developers, there are domains that are reserved for testing, including example.com. You can see a complete list at https://tools.ietf.org/html/rfc2606.

提示：虽然不是很多开发人员都知道，有一些域名是保留用于测试的，包括example.com。可以在https://tools.ietf.org/html/rfc2606网站看到完整的列表。

Once you have entered the values, click the `Create` button. ASP.NET Identity will create the user account, which will be displayed when your browser is redirected to the `Index` action method, as shown in Figure 13-5. You will see a different ID value because IDs are randomly generated for each user account.

一旦输入了这些值，点击“Create”按钮。ASP.NET Identity将创建此用户账号，当浏览器被重定向到`Index`动作方法，会显示出该用户的信息，如图13-5所示。从图中可以看到不同的ID值，这是因为对于每个用户账号，ID是随机生成的。

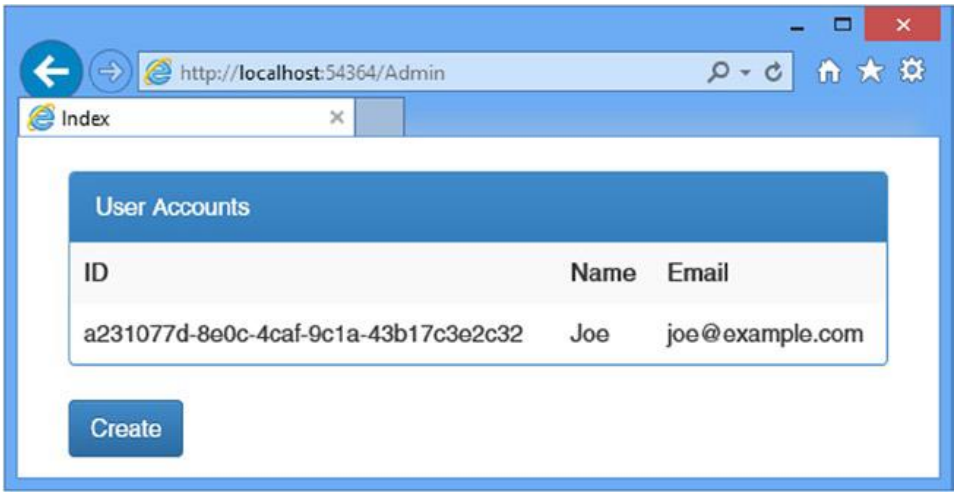

Figure 13-5. *The effect of adding a new user account*

图13-5. *添加新用户账号的效果*

Click the Create button again and enter the same details into the form, using the values in Table 13-6. This time when you submit the form, you will see an error reported through the model validation summary, as shown in Figure 13-6.

再次点击“Create”按钮，并在表章中输入同样细节，即，使用表13-6中的值。当这一次递交表单时，会看到一条通过模型验证摘要报告的错误，如图13-6所示。

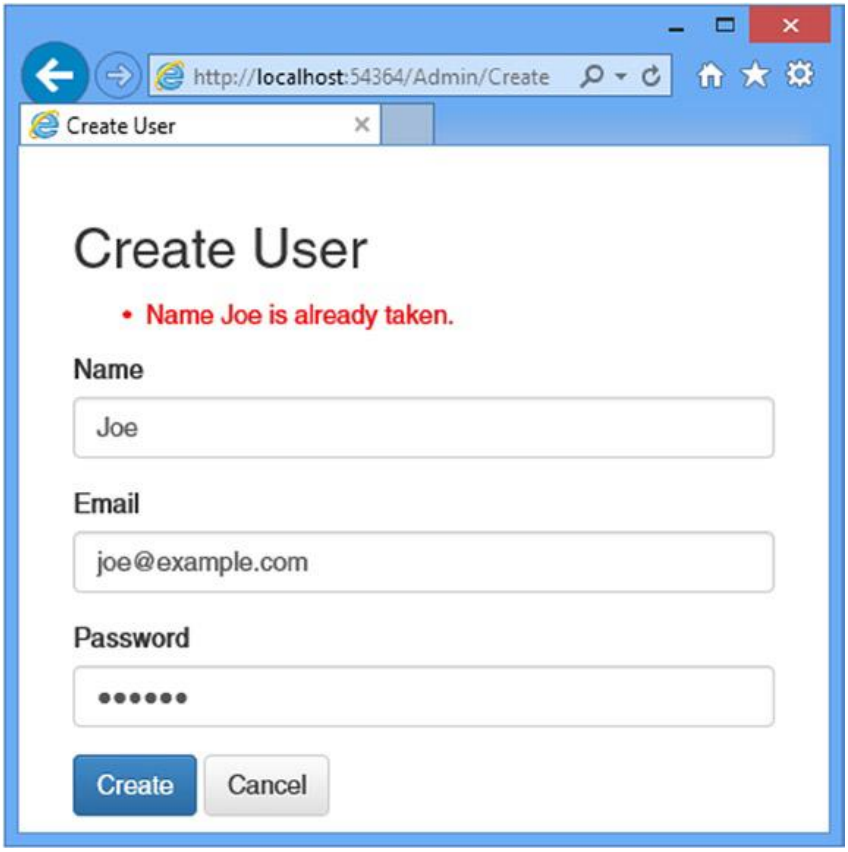

Figure 13-6. *An error trying to create a new user*

图13-6. *试图创建新用户时的错误*

13.3.3 Validating Passwords
13.3.3 验证口令

One of the most common requirements, especially for corporate applications, is to enforce a password policy. ASP.NET Identity provides the `PasswordValidator` class, which can be used to configure a password policy using the properties described in Table 13-7.

一个最常用的需求，特别是对于公司的应用程序，是强制口令策略。ASP.NET Identity提供了一个`PasswordValidator`类，可以用表13-7所描述的属性来配置口令策略。

Table 13-7. *The Properties Defined by the PasswordValidator Class*

表13-7. *PasswordValidator类定义的属性*

Name 名称	Description 描述
`RequiredLength`	Specifies the minimum length of a valid passwords. 指定合法口令的最小长度
`RequireNonLetterOrDigit`	When set to `true`, valid passwords must contain a character that is neither a letter nor a digit. 当设置为`true`时，合法口令必须含有非字母和数字的字符
`RequireDigit`	When set to `true`, valid passwords must contain a digit. 当设置为`true`时，合法口令必须含有数字
`RequireLowercase`	When set to true, valid passwords must contain a lowercase character.

	当设置为true时，合法口令必须含有小写字母
RequireUppercase	When set to true, valid passwords must contain an uppercase character. 当设置为true时，合法口令必须含有大写字母

A password policy is defined by creating an instance of the PasswordValidator class, setting the property values, and using the object as the value for the PasswordValidator property in the Create method that OWIN uses to instantiate the AppUserManager class, as shown in Listing 13-14.
定义口令策略的办法是，创建一个PasswordValidator类实例、设置其属性的值，并在OWIN用来实例化AppUserManager类的Create方法中将该对象作为PasswordValidator属性的值，如清单13-14所示。

Listing 13-14. *Setting a Password Policy in the AppUserManager.cs File*
清单13-14. *在AppUserManager.cs文件中设置口令策略*

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Users.Models;

namespace Users.Infrastructure {
    public class AppUserManager : UserManager<AppUser> {

        public AppUserManager(IUserStore<AppUser> store) : base(store) {
        }

        public static AppUserManager Create(IdentityFactoryOptions<AppUserManager>
                options, IOwinContext context) {

            AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
            AppUserManager manager = new AppUserManager(
                new UserStore<AppUser>(db));

            manager.PasswordValidator = new PasswordValidator {
                RequiredLength = 6,
                RequireNonLetterOrDigit = false,
                RequireDigit = false,
                RequireLowercase = true,
                RequireUppercase = true
            };

            return manager;
        }
    }
}
```

I used the `PasswordValidator` class to specify a policy that requires at least six characters and a mix of uppercase and lowercase characters. You can see how the policy is applied by starting the application, navigating to the `/Admin/Index` URL, clicking the `Create` button, and trying to create an account that has the password secret. The password doesn't meet the new password policy, and an error is added to the model state, as shown in Figure 13-7.

我用`PasswordValidator`类指定了一个策略，它要求至少6个字符，并混用大小字符。通过启动应用程序，便可以看到该口令策略的运用情况，导航到`/Admin/Index`网址，点击`Create`按钮，并尝试创建一个有口令加密的账号。若口令不满足这一新策略，便会在模型状态中添加一条错误消息，如图13-7所示。

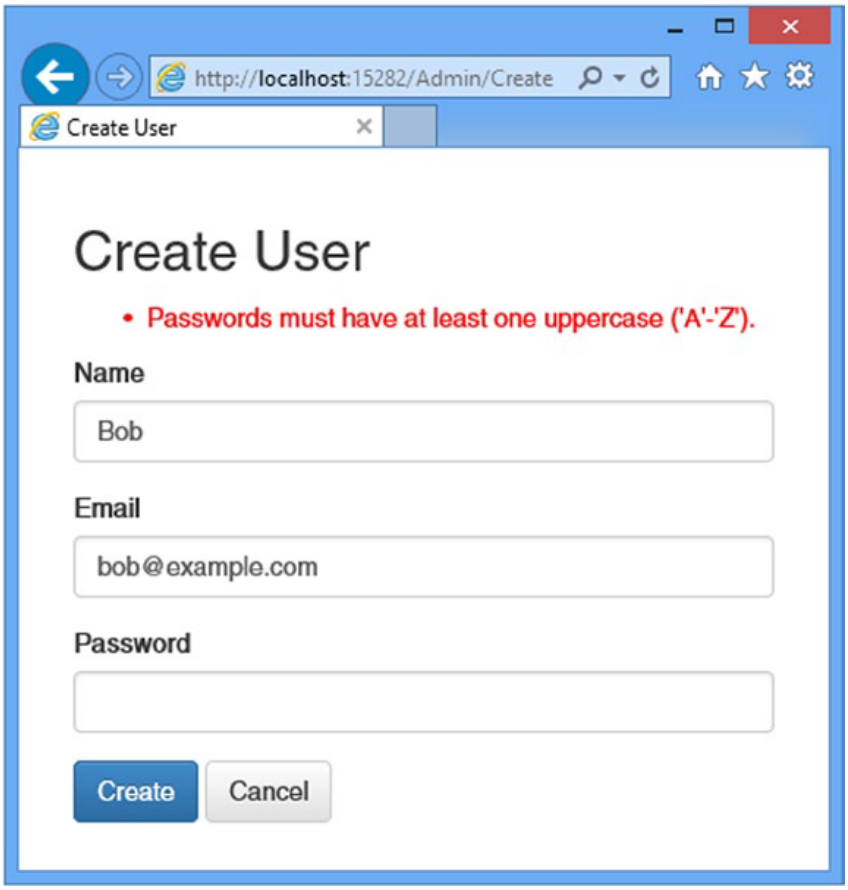

Figure 13-7. *Reporting an error when validating a password*

图13-7. 验证口令时报告的错误

Implementing a Custom Password Validator
实现自定义口令验证器

The built-in password validation is sufficient for most applications, but you may need to implement a custom policy, especially if you are implementing a corporate line-of-business application where complex password policies are common. Extending the built-in functionality is done by deriving a new class from `PasswordValidatator` and overriding the `ValidateAsync` method. As a demonstration, I added a class file called `CustomPasswordValidator.cs` in the `Infrastructure` folder and used it to define the class shown in Listing 13-15.

内建的口令验证对于大多数应用程序足够了，但你可能需要实现自定义的策略，尤其是在你实现一个公司的在线业务应用程序时，往往需要复杂的口令策略。这种对内建功能进行扩展的做法是，从

PasswordValidatator派生一个新类，并重写ValidateAsync方法。作为一个演示，我在Infrastructure文件夹中添加了一个类文件，名称为CustomPasswordValidator.cs，如清单13-15所示。

Listing 13-15. *The Contents of the CustomPasswordValidator.cs File*

清单13-15. *CustomPasswordValidator.cs文件的内容*

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNet.Identity;

namespace Users.Infrastructure {
    public class CustomPasswordValidator : PasswordValidator {

        public override async Task<IdentityResult> ValidateAsync(string pass) {
            IdentityResult result = await base.ValidateAsync(pass);
            if (pass.Contains("12345")) {
                var errors = result.Errors.ToList();
                errors.Add("Passwords cannot contain numeric sequences");
                result = new IdentityResult(errors);
            }
            return result;
        }
    }
}
```

I have overridden the ValidateAsync method and call the base implementation so I can benefit from the built-in validation checks. The ValidateAsync method is passed the candidate password, and I perform my own check to ensure that the password does not contain the sequence 12345. The properties of the IdentityResult class are read-only, which means that if I want to report a validation error, I have to create a new instance, concatenate my error with any errors from the base implementation, and pass the combined list as the constructor argument. I used LINQ to concatenate the base errors with my custom one.
这里重写了ValidateAsync方法，并调用了基实现，因此能够受益于内建的验证检查。给ValidateAsync方法传递了申请的口令，然后执行了自己的检查，确保口令中不含数据序列12345。IdentityResult类的属性是只读的，这意味着，如果想报告验证错误，则必须创建一个新实例，把这些错误与基实现的错误联系在一起，并将这种组合而成的列表作为构造器参数进行传递。为了将基实现的错误与自定义错误联系起来，我使用了LINQ。

Listing 13-16 shows the application of my custom password validator in the AppUserManager class.
清单13-6显示了在AppUserManager类中使用自定义口令验证器的应用程序。

Listing 13-16. *Applying a Custom Password Validator in the AppUserManager.cs File*

清单13-16. *在AppUserManager.cs文件中运用自定义口令验证器*

```
using Microsoft.AspNet.Identity;
```

```
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Users.Models;

namespace Users.Infrastructure {
    public class AppUserManager : UserManager<AppUser> {

        public AppUserManager(IUserStore<AppUser> store) : base(store) {
        }

        public static AppUserManager Create(IdentityFactoryOptions<AppUserManager>
                    options, IOwinContext context) {

            AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
            AppUserManager manager = new AppUserManager(
                new UserStore<AppUser>(db));

            manager.PasswordValidator = new CustomPasswordValidator {
                RequiredLength = 6,
                RequireNonLetterOrDigit = false,
                RequireDigit = false,
                RequireLowercase = true,
                RequireUppercase = true
            };

            return manager;
        }
    }
}
```

To test the custom password validation, try to create a new user account with the password secret12345. This will break two of the validation rules—one from the built-in validator and one from my custom implementation. Error messages for both problems are added to the model state and displayed when the Create button is clicked, as shown in Figure 13-8.

为了测试自定义口令验证器，尝试创建一个以口令密码为12345的新用户账号。这会打破两条规则——一条来自于内建验证器，一条来自于自定义实现。两个错误的错误消息被添加到了模型状态，并在点击Create按钮时被显示出来，如图13-8所示。

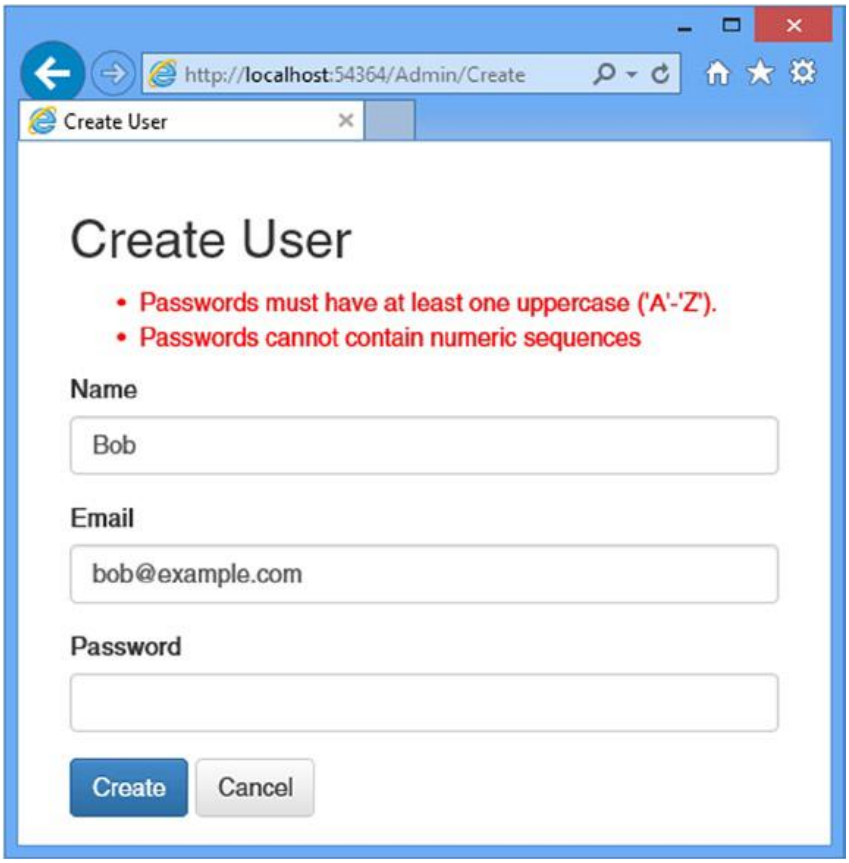

Figure 13-8. *The effect of a custom password validation policy*

图13-8. *自定义口令验证策略的效果*

13.3.4 Validating User Details
13.3.4 验证用户细节

More general validation can be performed by creating an instance of the `UserValidator` class and using the properties it defines to restrict other user property values. Table 13-8 describes the `UserValidator` properties.

还可以执行更一般的验证，办法是创建`UserValidator`类的实例，并使用它所定义的属性，以限制用户其他属性的值。表13-8描述了`UserValidator`的属性。

Table 13-8. *The Properties Defined by the UserValidator Class*

表13-8. *UserValidator类所定义的属性*

Name 名称	Description 描述
`AllowOnlyAlphanumericUserNames`	When `true`, usernames can contain only alphanumeric characters. 当为`true`时，用户名只能含有字母数字字符
`RequireUniqueEmail`	When true, e-mail addresses must be unique. 当为`true`时，邮件地址必须唯一

Performing validation on user details is done by creating an instance of the `UserValidator` class and assigning it to the `UserValidator` property of the user manager class within the `Create` method that OWIN uses to create instances. Listing 13-17 shows an example of using the built-in validator class.

对用户细节执行验证的做法是创建UserValidator类实例，并在OWIN用来创建实例的Create方法中，将它赋给用户管理器类的UserValidator属性。清单13-17演示了使用内建验证器类的一个示例。

Listing 13-17. *Using the Built-in user Validator Class in the AppUserManager.cs File*

清单13-17. *在AppUserManager.cs文件中使用内建的验证器类*

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Users.Models;

namespace Users.Infrastructure {
    public class AppUserManager : UserManager<AppUser> {

        public AppUserManager(IUserStore<AppUser> store) : base(store) {
        }

        public static AppUserManager Create(IdentityFactoryOptions<AppUserManager>
                    options, IOwinContext context) {

            AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
            AppUserManager manager = new AppUserManager(
                new UserStore<AppUser>(db));

            manager.PasswordValidator = new CustomPasswordValidator {
                RequiredLength = 6,
                RequireNonLetterOrDigit = false,
                RequireDigit = false,
                RequireLowercase = true,
                RequireUppercase = true
            };

            manager.UserValidator = new UserValidator<AppUser>(manager) {
                AllowOnlyAlphanumericUserNames = true,
                RequireUniqueEmail = true
            };

            return manager;
        }
    }
}
```

The UserValidator class takes a generic type parameter that specifies the type of the user class,

which is AppUser in this case. Its constructor argument is the user manager class, which is an instance of the user manager class (which is AppUserManager for my application).
UserValidator类有一个泛型的类型参数，它指定了用户类的类型，即本示例中的AppUser。它的构造器参数是用户管理器类，这是用户管理器类（此应用程序中的AppUserManager）的一个实例。

The built-in validation support is rather basic, but you can create a custom validation policy by creating a class that is derived from UserValidator. As a demonstration, I added a class file called CustomUserValidator.cs to the Infrastructure folder and used it to create the class shown in Listing 13-18.
内建的验证支持是相当基本的，但通过创建UserValidator的派生类，可以创建自定义验证策略。作为一个演示，我在Infrastructure文件夹中添加了一个名称为CustomUserValidator.cs的类文件，并用它创建了如清单13-19所示的类。

Listing 13-18. *The Contents of the CustomUserValidator.cs File*
清单13-18. *CustomUserValidator.cs文件的内容*

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNet.Identity;
using Users.Models;

namespace Users.Infrastructure {
    public class CustomUserValidator : UserValidator<AppUser> {

        public CustomUserValidator(AppUserManager mgr) : base(mgr) {
        }

        public override async Task<IdentityResult> ValidateAsync(AppUser user) {
            IdentityResult result = await base.ValidateAsync(user);
            if (!user.Email.ToLower().EndsWith("@example.com")) {
                var errors = result.Errors.ToList();
                errors.Add("Only example.com email addresses are allowed");
                result = new IdentityResult(errors);
            }
            return result;
        }
    }
}
```

The constructor of the derived class must take an instance of the user manager class and call the base implementation so that the built-in validation checks can be performed. Custom validation is implemented by overriding the ValidateAsync method, which takes an instance of the user class and returns an IdentityResult object. My custom policy restricts users to e-mail addresses in the example.com domain and performs the same LINQ manipulation I used for password validation to concatenate my error message with those produced by the base class. Listing 13-19 shows how I applied my custom validation class in the

Create method of the AppUserManager class, replacing the default implementation.
这个派生类的构造器必须以用户管理器类实例为参数，并调用基实现，才能够执行内建的验证检查。自定义验证是通过重写ValidateAsync方法而实现的，该方法以用户类实例为参数，并返回一个IdentityResult对象。上述自定义验证策略将用户的E-mail地址限制在example.com主域内，并执行了与口令验证同样的LINQ操作，将这里的错误消息与基类产生的错误消息关联在一起。清单13-19演示了如何在AppUserManager类的Create方法中运用自定义验证类，用以替换默认的实现。

Listing 13-19. *Using a Custom User Validation Class in the AppUserManager.cs File*
清单13-19. *在AppUserManager.cs文件中使用自定义用户验证类*

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Users.Models;

namespace Users.Infrastructure {
    public class AppUserManager : UserManager<AppUser> {

        public AppUserManager(IUserStore<AppUser> store) : base(store) {
        }

        public static AppUserManager Create(IdentityFactoryOptions<AppUserManager>
                options, IOwinContext context) {

            AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
            AppUserManager manager = new AppUserManager(
                new UserStore<AppUser>(db));

            manager.PasswordValidator = new CustomPasswordValidator {
                RequiredLength = 6,
                RequireNonLetterOrDigit = false,
                RequireDigit = false,
                RequireLowercase = true,
                RequireUppercase = true
            };

            manager.UserValidator = new CustomUserValidator(manager) {
                AllowOnlyAlphanumericUserNames = true,
                RequireUniqueEmail = true
            };

            return manager;
        }
```

```
    }
}
```

You can see the result if you try to create an account with an e-mail address such as bob@otherdomain.com, as shown in Figure 13-9.
如果试图创建一个E-mail地址为bob@otherdomain.com的账号，便会看到如图13-9所示的结果。

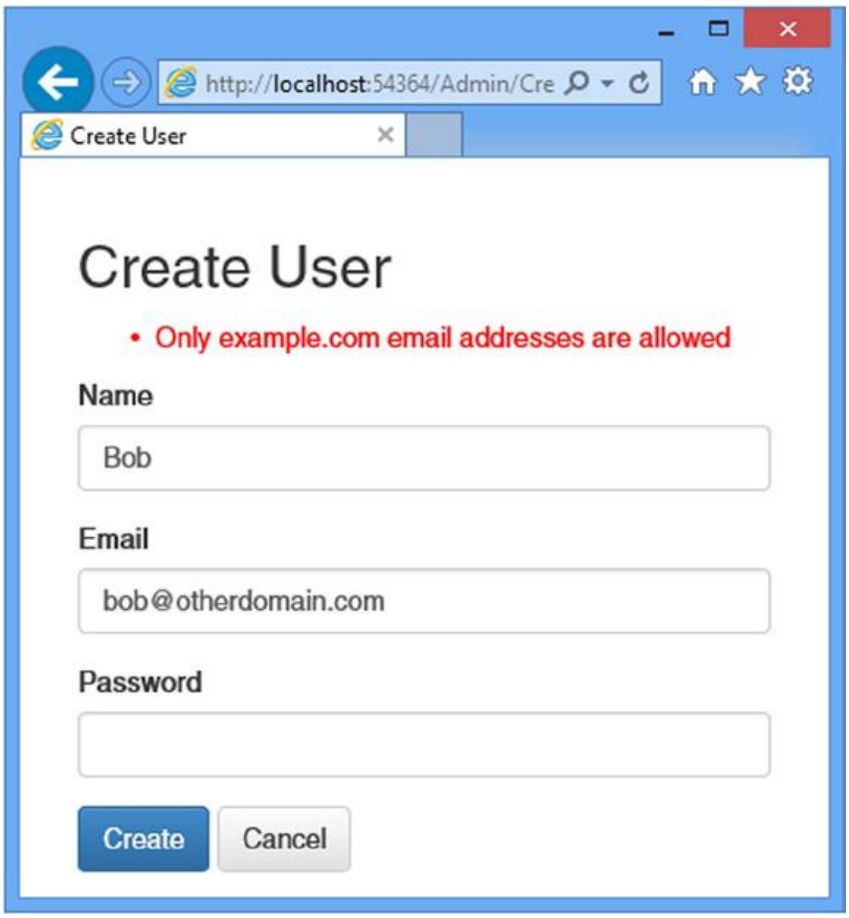

Figure 13-9. *An error message shown by a custom user validation policy*
图13-9. *由自定义用户验证显示的错误消息*

13.4 Completing the Administration Features

13.4 完成管理特性

I only have to implement the features for editing and deleting users to complete my administration tool. In Listing 13-20, you can see the changes I made to the `Views/Admin/Index.cshtml` file to target `Edit` and `Delete` actions in the `Admin` controller.
为了完成本例的管理工具，我只需实现编辑和删除用户的特性。在清单13-20中可以看到我对`Views/Admin/Index.cshtml`文件所做修改，它们的目标是`Admin`控制器中的`Edit`和`Delete`方法。

Listing 13-20. *Adding Edit and Delete Buttons to the Index.cshtml File*

清单13-20. 在Index.cshtml文件中添加“Edit”和“Delete”按钮

```
@using Users.Models
@model IEnumerable<AppUser>
@{ ViewBag.Title = "Index"; }
<div class="panel panel-primary">
    <div class="panel-heading">
        User Accounts
    </div>
    <table class="table table-striped">
        <tr><th>ID</th><th>Name</th><th>Email</th><th></th></tr>
        @if (Model.Count() == 0) {
            <tr><td colspan="4" 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>
                        @using (Html.BeginForm("Delete", "Admin",
                                    new { id = user.Id })) {
                            @Html.ActionLink("Edit", "Edit", new { id = user.Id },
                                    new { @class = "btn btn-primary btn-xs" })
                            <button class="btn btn-danger btn-xs" type="submit">
                                Delete
                            </button>
                        }
                    </td>
                </tr>
            }
        }
    </table>
</div>
@Html.ActionLink("Create", "Create", null, new { @class = "btn btn-primary" })
```

Tip You will notice that I have put the `Html.ActionLink` call that targets the `Edit` action method inside the scope of the `Html.Begin` helper. I did this solely so that the Bootstrap styles will style both elements as buttons displayed on a single line.

提示：你可能会注意到，我在`Html.Begin`辅助器的范围内放入了一个`Html.ActionLink`调用，其目标为`Edit`动作方法。这么做纯粹是为了让Bootstrap将两个元素的样式作为按钮显示成一行。

13.4.1 Implementing the Delete Feature
13.4.1 实现 Delete 特性

The user manager class defines a `DeleteAsync` method that takes an instance of the user class and removes it from the database. In Listing 13-21, you can see how I have used the `DeleteAsync` method to implement the delete feature of the `Admin` controller.
用户管理器类定义了一个`DeleteAsync`方法，它以用户类实例为参数，并将其从数据库删除。在清单13-21中，可以看到如何用`DeleteAsync`方法来实现`Admin`控制器的删除特性。

Listing 13-21. *Deleting Users in the AdminController.cs File*
清单13-21. *AdminController.cs文件中的删除用户*

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

namespace Users.Controllers {
    public class AdminController : Controller {

        // ...other action methods omitted for brevity...
        // ...出于简化，这里忽略了其他方法...

        [HttpPost]
        public async Task<ActionResult> 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 {
                    return View("Error", result.Errors);
                }
            } else {
                return View("Error", new string[] { "User Not Found" });
            }
        }

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

```
        }

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

My action method receives the unique ID for the user as an argument, and I use the `FindByIdAsync` method to locate the corresponding user object so that I can pass it to `DeleteAsync` method. The result of the `DeleteAsync` method is an `IdentityResult`, which I process in the same way I did in earlier examples to ensure that any errors are displayed to the user. You can test the delete functionality by creating a new user and then clicking the Delete button that appears alongside it in the `Index` view.
该动作方法以用户的唯一ID作为参数，并且使用`FindByIdAsync`方法定位了相应的用户对象，以便将该对象传递给`DeleteAsync`方法。`DeleteAsync`方法的结果是一个`IdentityResult`，对它的处理方式与之前示例的方式相同，以确保将任何错误都显示给用户。为了测试这个删除功能，可以创建一个新的用户，然后在`Index`视图中点击出现在该用户旁边的“Delete”按钮。

There is no view associated with the `Delete` action, so to display any errors I created a view file called `Error.cshtml` in the `Views/Shared` folder, the contents of which are shown in Listing 13-22.
`Delete`动作没有相关联的视图，因此，为了将错误显示出来，我在`Views/Shared`文件夹中创建了一个视图文件，名称为`Error.cshtml`，其内容如清单13-22所示。

Listing 13-22. *The Contents of the Error.cshtml File*
清单13-22. *Error.cshtml文件的内容*

```
@model IEnumerable<string>
@{ ViewBag.Title = "Error";}
<div class="alert alert-danger">
    @switch (Model.Count()) {
        case 0:
            @: Something went wrong. Please try again
            break;
        case 1:
            @Model.First();
            break;
        default:
            @: The following errors were encountered:
                <ul>
                    @foreach (string error in Model) {
                        <li>@error</li>
                    }
                </ul>
```

```
            break;
    }
</div>
@Html.ActionLink("OK", "Index", null, new { @class = "btn btn-default" })
```

Tip I put this view in the `Views/Shared` folder so that it can be used by other controllers, including the one I create to manage roles and role membership in Chapter 14.

提示：我将此视图放在`Views/Shared`文件夹中，是为了其他控制器也能够使用它，包括第14章为了管理角色及其成员要创建的控制器。

13.4.2 Implementing the Edit Feature
13.4.2 实现 Edit 特性

To complete the administration tool, I need to add support for editing the e-mail address and password for a user account. These are the only properties defined by users at the moment, but I'll show you how to extend the schema with custom properties in Chapter 15. Listing 13-23 shows the `Edit` action methods that I added to the `Admin` controller.

为了完成用户管理工具，我需要为编辑用户账号的E-mail地址和口令的支持。此时这些是用户定义仅有的属性，不过到第15章时，将演示如何扩展数据库架构，使用户带有自定义属性。清单13-23显示了添加到`Admin`控制器中的`Edit`动作方法。

Listing 13-23. *Adding the Edit Actions in the AdminController.cs File*

清单13-23. *在AdminController.cs文件中添加Edit动作*

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

namespace Users.Controllers {
    public class AdminController : Controller {

        // ...other action methods omitted for brevity...
        // ...出于简化，这里忽略了其他动作方法...

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

```
}

[HttpPost]
public async Task<ActionResult> Edit(string id, string email, string password) {
    AppUser user = await UserManager.FindByIdAsync(id);
    if (user != null) {
        user.Email = email;
        IdentityResult validEmail
            = await UserManager.UserValidator.ValidateAsync(user);
        if (!validEmail.Succeeded) {
            AddErrorsFromResult(validEmail);
        }
        IdentityResult validPass = null;
        if (password != string.Empty) {
            validPass
                = await UserManager.PasswordValidator.ValidateAsync(password);
            if (validPass.Succeeded) {
                user.PasswordHash =
                    UserManager.PasswordHasher.HashPassword(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 (string error in result.Errors) {
        ModelState.AddModelError("", error);
    }
}
```

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

The `Edit` action targeted by `GET` requests uses the ID string embedded in the `Index` view to call the `FindByIdAsync` method in order to get an `AppUser` object that represents the user.
由GET请求作为目标的`Edit`动作使用`Index`视图中的ID字符串调用`FindByIdAsync`方法，目的是获得表示该用户的`AppUser`对象。

The more complex implementation receives the `POST` request, with arguments for the user ID, the new e-mail address, and the password. I have to perform several tasks to complete the editing operation.
那个较复杂的实现接收POST请求，以用户ID、新E-mail地址以及口令为参数。我必须执行几个任务才能完成这种编辑操作。

The first task is to validate the values I have received. I am working with a simple user object at the moment—although I'll show you how to customize the data stored for users in Chapter 15—but even so, I need to validate the user data to ensure that I don't violate the custom policies defined in the “Validating User Details” and “Validating Passwords” sections. I start by validating the e-mail address, which I do like this:
第一个任务是验证接收到的值。此时使用的是一个简单的用户对象——尽管第15章将演示如何自定义存储用户的数据——但即使如此，还是需要验证用户数据，以确保不会与“验证用户细节”和“验证口令”小节中定义的自定义策略相冲突。首先验证E-mail地址，做法如下：

```
...
user.Email = email;
IdentityResult validEmail = await UserManager.UserValidator.ValidateAsync(user);
if (!validEmail.Succeeded) {
    AddErrorsFromResult(validEmail);
}
...
```

Tip Notice that I have to change the value of the `Email` property before I perform the validation because the `ValidateAsync` method only accepts instances of the user class.
提示：注意，在执行验证之前，必须修改`Email`属性的值，因为`ValidateAsync`方法只接收用户类的实例。

The next step is to change the password, if one has been supplied. ASP.NET Identity stores hashes of passwords, rather than the passwords themselves—this is intended to prevent passwords from being stolen. My next step is to take the validated password and generate the hash code that will be stored in the database so that the user can be authenticated (which I demonstrate in Chapter 14).
下一个步骤是在已提供口令时，修改用户口令。ASP.NET Identity存储的是口令的哈希值，而不是口令本身——目的是防止口令被窃取。我的下一个步骤是取得这个已验证的口令，并生成将被存储到数据库中的哈希码，以便让用户能够被认证，这将在第14章演示。

Passwords are converted to hashes through an implementation of the IPasswordHasher interface, which is obtained through the AppUserManager.PasswordHasher property. The IPasswordHasher interface defines the HashPassword method, which takes a string argument and returns its hashed value, like this:
将口令转换成哈希值是通过IPasswordHasher接口的实现来做的，该接口实现是通过AppUserManager.PasswordHasher属性获得的。IPasswordHasher接口定义了HashPassword方法，它以一个字符串为参数，返回该字符串的哈希值，如下所示：

```
...
if (password != string.Empty) {
    validPass = await UserManager.PasswordValidator.ValidateAsync(password);
    if (validPass.Succeeded) {
        user.PasswordHash = UserManager.PasswordHasher.HashPassword(password);
    } else {
        AddErrorsFromResult(validPass);
    }
}
...
```

Changes to the user class are not stored in the database until the UpdateAsync method is called, like this:
对用户类的修改直到调用UpdateAsync方法时，才会存储到数据库中，如下所示：

```
...
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);
    }
}
...
```

1. Creating the View
1. 创建视图

The final component is the view that will render the current values for a user and allow new values to be submitted to the controller. Listing 13-24 shows the contents of the Views/Admin/Edit.cshtml file.
最后一个组件是渲染当前用户值并将新值递交给控制器的视图。清单13-24显示了Views/Admin/Edit.cshtml文件的内容。

Listing 13-24. *The Contents of the Edit.cshtml File*

清单13-24. Edit.cshtml文件的内容

```
@model Users.Models.AppUser
@{ ViewBag.Title = "Edit"; }
@Html.ValidationSummary(false)
<h2>Edit User</h2>

<div class="form-group">
    <label>Name</label>
    <p class="form-control-static">@Model.Id</p>
</div>
@using (Html.BeginForm()) {
    @Html.HiddenFor(x => x.Id)
    <div class="form-group">
        <label>Email</label>
        @Html.TextBoxFor(x => x.Email, new { @class = "form-control" })
    </div>
    <div class="form-group">
        <label>Password</label>
        <input name="password" type="password" class="form-control" />
    </div>
    <button type="submit" class="btn btn-primary">Save</button>
    @Html.ActionLink("Cancel", "Index", null, new { @class = "btn btn-default" })
}
```

There is nothing special about the view. It displays the user ID, which cannot be changed, as static text and provides a form for editing the e-mail address and password, as shown in Figure 13-10. Validation problems are displayed in the validation summary section of the view, and successfully editing a user account will return to the list of accounts in the system.

该视图没什么特别的。它将用户ID显示成静态文本，这是不能修改的，并且提供了一个对E-mail地址和口令进行编辑的表单，如图13-10所示。验证问题会显示在视图的验证摘要处，当成功编辑一个用户账号时，会返回到系统的账号列表。

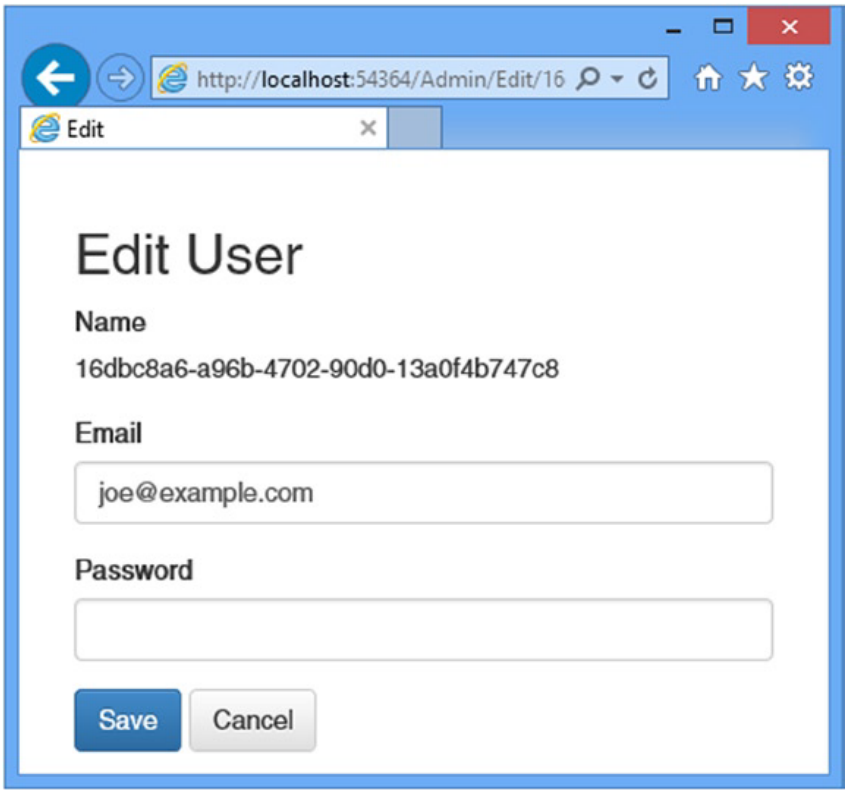

Figure 13-10. *Editing a user account*

图 13-10. *编辑用户账号*

13.5 Summary

13.5 小结

In this chapter, I showed you how to create the configuration and classes required to use ASP.NET Identity and demonstrated how they can be applied to create a user administration tool. In the next chapter, I show you how to perform authentication and authorization with ASP.NET Identity.

在本章中，我演示了如何创建使用ASP.NET Identity所需的配置和类，并且演示了如何运用它们来创建用户管理工具。下一章将演示如何用ASP.NET Identity执行认证与授权。

CHAPTER 14

14 Applying ASP.NET Identity

14 运用ASP.NET Identity

In this chapter, I show you how to apply ASP.NET Identity to authenticate and authorize the user accounts created in the previous chapter. I explain how the ASP.NET platform provides a foundation for authenticating requests and how ASP.NET Identity fits into that foundation to authenticate users and enforce authorization through roles. Table 14-1 summarizes this chapter.

本章将演示如何将ASP.NET Identity用于对上一章中创建的用户账号进行认证与授权。我将解释ASP.NET平台对请求进行认证的基础，并解释ASP.NET Identity如何融入这种基础对用户进行认证，以及通过角色增强授权功能。表14-1描述了本章概要。

Table 14-1. *Chapter Summary*

表14-1. *本章概要*

Problem **问题**	**Solution** **解决方案**	**Listing** **清单号**
Prepare an application for user authentication. 准备用户认证的应用程序	Apply the `Authorize` attribute to restrict access to action methods and define a controller to which users will be redirected to provide credentials. 运用`Authorize`注解属性来限制对动作方法的访问，并定义一个对用户重定向的控制器，以便让用户提供凭据	1–4
Authenticate a user. 认证用户	Check the name and password using the `FindAsync` method defined by the user manager class and create an implementation of the `IIdentity` interface using the `CreateIdentityMethod`. Set an authentication cookie for subsequent requests by calling the `SignIn` method defined by the authentication manager class. 使用由用户管理器类定义的`FindAsync`方法检查用户名和口令，并使用`CreateIdentityMethod`创建一个`IIdentity`接口的实现。通过调用由认证管理器类定义的`SignIn`方法，设置后继请求的认证Cookie。	5
Prepare an application for role-based authorization. 准备基于角色授权的应用程序	Create a role manager class and register it for instantiation in the OWIN startup class. 创建一个角色管理器类，将其注册为OWIN启动类中的实例化	6–8
Create and delete roles. 创建和删除角色	Use the `CreateAsync` and `DeleteAsync` methods defined by the role manager class. 使用由角色管理器类定义的`CreateAsync`和`DeleteAsync`方法。	9–12

Manage role membership. 管理角色成员	Use the `AddToRoleAsync` and `RemoveFromRoleAsync` methods defined by the user manager class. 使用由用户管理器类定义的`AddToRoleAsync`和`RemoveFromRoleAsync`方法	13–15
Use roles for authorization. 使用角色进行授权	Set the `Roles` property of the `Authorize` attribute. 设置`Authorize`注解属性的`Roles`属性	16–19
Seed the database with initial content. 将初始化内容植入数据库	Use the database context initialization class. 使用数据库上下文的初始化类	20, 21

14.1 Preparing the Example Project

14.1 准备示例项目

In this chapter, I am going to continue working on the `Users` project I created in Chapter 13. No changes to the application components are required.
在本章中，我打算继续沿用第13章所创建的`Users`项目，不需要修改该应用程序的组件。

14.2 Authenticating Users

14.2 认证用户

The most fundamental activity for ASP.NET Identity is to authenticate users, and in this section, I explain and demonstrate how this is done. Table 14-2 puts authentication into context.
ASP.NET Identity最基本的活动就是认证用户，在本小节中，我将解释并演示其做法。表14-2描述了认证的情形。

Table 14-2. *Putting Authentication in Context*
表14-2. *认证情形*

Question **问题**	**Answer** **回答**
What is it? 什么是认证？	Authentication validates credentials provided by users. Once the user is authenticated, requests that originate from the browser contain a cookie that represents the user identity. 认证是验证用户提供的凭据。一旦用户已被认证，源自该浏览器的请求便会含有表示该用户标识的Cookie。
Why should I care? 为何要关心它？	Authentication is how you check the identity of your users and is the first step toward restricting access to sensitive parts of the application. 认证是你检查用户标识的办法，也是限制对应用程序敏感部分进行访问的第一步。
How is it used by the MVC framework? 如何在MVC框架中使用它？	Authentication features are accessed through the `Authorize` attribute, which is applied to controllers and action methods in order to restrict access to authenticated users. 认证特性是通过`Authorize`注解属性进行访问的，将该注解属性运用于控制器和动作方法，目的是将访问限制到已认证用户。

Tip I use names and passwords stored in the ASP.NET Identity database in this chapter. In Chapter 15, I demonstrate how ASP.NET Identity can be used to authenticate users with a service from Google (Identity also supports authentication for Microsoft, Facebook, and Twitter accounts).
提示：本章会使用存储在ASP.NET Identity数据库中的用户名和口令。在第15章中将演示如何将ASP.NET Identity用于认证享有Google服务的用户（Identity还支持对Microsoft、Facebook以及Twitter账号的认证）。

14.2.1 Understanding the Authentication/Authorization Process
14.2.1 理解认证/授权过程

The ASP.NET Identity system integrates into the ASP.NET platform, which means you use the standard MVC framework techniques to control access to action methods, such as the `Authorize` attribute. In this section, I am going to apply basic restrictions to the `Index` action method in the `Home` controller and then implement the features that allow users to identify themselves so they can gain access to it. Listing 14-1 shows how I have applied the `Authorize` attribute to the `Home` controller.
ASP.NET Identity系统集成到了ASP.NET平台，这意味着你可以使用标准的MVC框架技术来控制对动作方法的访问，例如使用`Authorize`注解属性。在本小节中，我打算在`Home`控制中的`Index`动作方法上运用基本的限制，然后实现让用户对自己进行标识，以使他们能够访问。清单14-1演示了如何将`Authorize`注解属性运用于`Home`控制器。

Listing 14-1. *Securing the Home Controller*
清单14-1. *实施Home控制器的安全*

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

namespace Users.Controllers {

    public class HomeController : Controller {

        [Authorize]
        public ActionResult Index() {
            Dictionary<string, object> data
                = new Dictionary<string, object>();
            data.Add("Placeholder", "Placeholder");
            return View(data);
        }
    }
}
```

Using the `Authorize` attribute in this way is the most general form of authorization and restricts access to the `Index` action methods to requests that are made by users who have been authenticated by the application.
这种方式使用`Authorize`注解属性是授权的最一般形式，它限制了对`Index`动作方法的访问，由用户发送给该动作方法的请求必须是应用程序已认证的用户。

If you start the application and request a URL that targets the Index action on the Home controller (/Home/Index, /Home, or just /), you will see the error shown by Figure 14-1.
如果启动应用程序，并请求以Home控制器中Index动作为目标的URL（/Home/Index、/Home或/），将会看到如图14-1所示的错误。

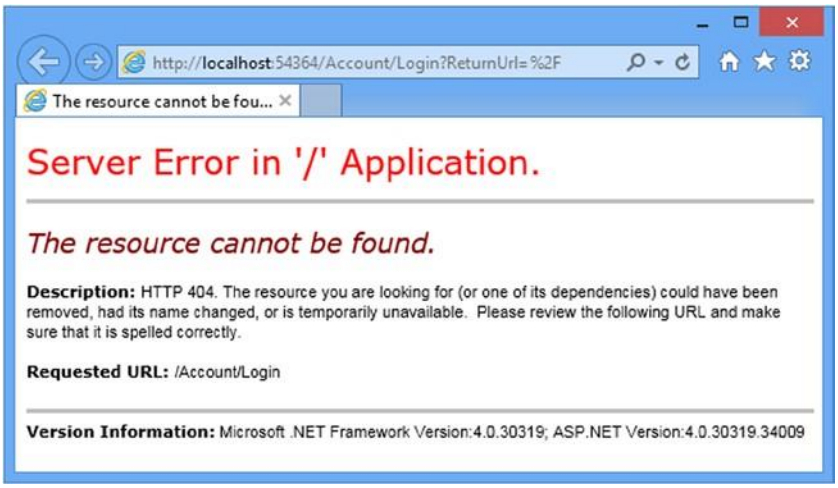

Figure 14-1. *Requesting a protected URL*
图 14-1. *请求一个受保护的 URL*

The ASP.NET platform provides some useful information about the user through the HttpContext object, which is used by the Authorize attribute to check the status of the current request and see whether the user has been authenticated. The HttpContext.User property returns an implementation of the IPrincipal interface, which is defined in the System.Security.Principal namespace. The IPrincipal interface defines the property and method shown in Table 14-3.
ASP.NET平台通过HttpContext对象提供一些关于用户的有用信息，该对象由Authorize注解属性使用的，以检查当前请求的状态，考察用户是否已被认证。HttpContext.User属性返回的是IPrincipal接口的实现，该接口是在System.Security.Principal命名空间中定义的。IPrincipal接口定义了如表14-3所示的属性和方法。

Table 14-3. *The Members Defined by the IPrincipal Interface*
表14-3. *IPrincipal接口所定义的成员*

Name 名称	Description 描述
Identity	Returns an implementation of the IIdentity interface that describes the user associated with the request. 返回IIdentity接口的实现，它描述了与请求相关联的用户
IsInRole(role)	Returns true if the user is a member of the specified role. See the “Authorizing Users with Roles” section for details of managing authorizations with roles. 如果用户是指定角色的成员，则返回true。参见“以角色授权用户”小节，其中描述了以角色进行授权管理的细节

The implementation of IIdentity interface returned by the IPrincipal.Identity property provides some basic, but useful, information about the current user through the properties I have described in Table 14-4.

由IPrincipal.Identity属性返回的IIdentity接口实现通过一些属性提供了有关当前用户的一些基本却有用的信息，表14-4描述了这些属性。

Table 14-4. *The Properties Defined by the IIdentity Interface*

表14-4. *IIdentity接口定义的属性*

Name 名称	Description 描述
AuthenticationType	Returns a string that describes the mechanism used to authenticate the user 返回一个字符串，描述了用于认证用户的机制
IsAuthenticated	Returns true if the user has been authenticated 如果用户已被认证，返回true。
Name	Returns the name of the current user 返回当前用户的用户名

Tip In Chapter 15 I describe the implementation class that ASP.NET Identity uses for the IIdentity interface, which is called ClaimsIdentity.

提示：第15章会描述ASP.NET Identity用于IIdentity接口的实现类，其名称为ClaimsIdentity。

ASP.NET Identity contains a module that handles the AuthenticateRequest life-cycle event, which I described in Chapter 3, and uses the cookies sent by the browser to establish whether the user has been authenticated. I'll show you how these cookies are created shortly. If the user is authenticated, the ASP.NET framework module sets the value of the IIdentity.IsAuthenticated property to true and otherwise sets it to false. (I have yet to implement the feature that will allow users to authenticate, which means that the value of the IsAuthenticated property is always false in the example application.)

ASP.NET Identity含有一个处理AuthenticateRequest生命周期事件（第3章曾做过描述）的模块，并使用浏览器发送过来的Cookie确认用户是否已被认证。我很快会演示如何创建这些Cookie。如果用户已被认证，此ASP.NET框架模块便会将IIdentity.IsAuthenticated属性的值设置为true，否则设置为false。（此刻尚未实现让用户进行认证的特性，这意味着在本示例应用程序中，IsAuthenticated属性的值总是false。）

The Authorize module checks the value of the IsAuthenticated property and, finding that the user isn't authenticated, sets the result status code to 401 and terminates the request. At this point, the ASP.NET Identity module intercepts the request and redirects the user to the /Account/Login URL. This is the URL that I defined in the IdentityConfig class, which I specified in Chapter 13 as the OWIN startup class, like this:

Authorize模块检查IsAuthenticated属性的值，会发现该用户是未认证的，于是将结果状态码设置为401（未授权），并终止该请求。但在这一点处（这里是ASP.NET Identity在请求生命周期中的切入点——译者注），ASP.NET Identity模块会拦截该请求，并将用户重定向到/Account/Login URL。我在IdentityConfig类中已定义了此URL，IdentityConfig是第13章所指定的OWIN启动类，如下所示：

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;
using Users.Infrastructure;
```

```
namespace Users {
    public class IdentityConfig {
        public void Configuration(IAppBuilder app) {

            app.CreatePerOwinContext<AppUserManager>(AppUserManager.Create);

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

The browser requests the `/Account/Login` URL, but since it doesn't correspond to any controller or action in the example project, the server returns a 404 – Not Found response, leading to the error message shown in Figure 14-1.

浏览器请求`/Account/Login`时，但因为示例项目中没有相应的控制器或动作，于服务器返回了“404 – 未找到”响应，从而导致了如图14-1所示的错误消息。

14.2.2 Preparing to Implement Authentication

14.2.2 实现认证的准备

Even though the request ends in an error message, the request in the previous section illustrates how the ASP.NET Identity system fits into the standard ASP.NET request life cycle. The next step is to implement a controller that will receive requests for the `/Account/Login` URL and authenticate the user. I started by adding a new model class to the `UserViewModels.cs` file, as shown in Listing 14-2.

虽然请求终止于一条错误消息，但上一小节的请求已勾画出ASP.NET Identity系统是如何切入标准的ASP.NET请求生命周期的。下一个步骤是实现一个控制器，用它来接收对`/Account/Login` URL的请求，并认证用户。我首先在`UserViewModels.cs`文件中添加了一个模型类，如清单14-2所示。

Listing 14-2. *Adding a New Model Class to the UserViewModels.cs File*

清单14-2. *在UserViewModels.cs文件中添加一个新的模型类*

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

The new model has `Name` and `Password` properties, both of which are decorated with the `Required` attribute so that I can use model validation to check that the user has provided values.
新模型具有`Name`和`Password`属性，两者都用`Required`注解属性进行了注释，以使我能够使用模型验证来检查用户是否提供了这些属性的值。

Tip In a real project, I would use client-side validation to check that the user has provided name and password values before submitting the form to the server, but I am going to keep things focused on identity and the server-side functionality in this chapter. See *Pro ASP.NET MVC 5* for details of client-side form validation.
提示：在一个实际的项目中，我会在用户将表单递交到服务器之前，使用客户端验证来检查用户已经提供了`Name`和`Password`的值，但在本章中，我打算把注意力集中在标识和服务器端的功能方面。客户端表单验证的详情可参阅*Pro ASP.NET MVC 5*一书。

I added an `Account` controller to the project, as shown in Listing 14-3, with `Login` action methods to collect and process the user's credentials. I have not implemented the authentication logic in the listing because I am going to define the view and then walk through the process of validating user credentials and signing users into the application.
我在项目中添加了一个`Account`控制器，如清单14-3所示，其中带有`Login`动作方法，用以收集和处理用户的凭据。该清单尚未实现认证逻辑，因为我打算先定义视图，然后再实现验证用户凭据的过程，并让用户签入应用程序。

Listing 14-3. *The Contents of the AccountController.cs File*
清单14-3. *AccountController.cs文件的内容*

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

namespace Users.Controllers {

    [Authorize]
    public class AccountController : Controller {

        [AllowAnonymous]
```

```
        public ActionResult Login(string returnUrl) {
            if (ModelState.IsValid) {
            }
            ViewBag.returnUrl = returnUrl;
            return View();
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> Login(LoginModel details, string returnUrl) {
            return View(details);
        }
    }
}
```

Even though it doesn't authenticate users yet, the `Account` controller contains some useful infrastructure that I want to explain separately from the ASP.NET Identity code that I'll add to the `Login` action method shortly.
尽管它此刻尚未认证用户，但`Account`控制器已包含了一些有用的基础结构，我想通过ASP.NET Identity代码对这些结构分别加以解释，很快就会在`Login`动作方法中添加这些代码。

First, notice that both versions of the `Login` action method take an argument called `returnUrl`. When a user requests a restricted URL, they are redirected to the `/Account/Login` URL with a query string that specifies the URL that the user should be sent back to once they have been authenticated. You can see this if you start the application and request the `/Home/Index` URL. Your browser will be redirected, like this:
首先要注意`Login`动作方法有两个版本，它们都有一个名称为`returnUrl`的参数。当用户请求一个受限的URL时，他们被重定向到`/Account/Login` URL上，并带有查询字符串，该字符串指定了一旦用户得到认证后将用户返回的URL，如下所示：

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

The value of the `ReturnUrl` query string parameter allows me to redirect the user so that navigating between open and secured parts of the application is a smooth and seamless process.
`ReturnUrl`查询字符串参数的值可让我能够对用户进行重定向，使应用程序公开和保密部分之间的导航成为一个平滑无缝的过程。

Next, notice the attributes that I have applied to the `Account` controller. Controllers that manage user accounts contain functionality that should be available only to authenticated users, such as password reset, for example. To that end, I have applied the `Authorize` attribute to the controller class and then used the `AllowAnonymous` attribute on the individual action methods. This restricts action methods to authenticated users by default but allows unauthenticated users to log in to the application.
下一个要注意的是运用于`Account`控制器的注解属性。管理用户账号的控制器含有只能由已认证用户才能使用的功能，例如口令重置。为此，我在控制器类上运用了`Authorize`注解属性，然后又在个别动作方法上运用了`AllowAnonymous`注解属性。这会将这些动作方法默认限制到已认证用户，但又能允许未

认证用户登录到应用程序。

Finally, I have applied the `ValidateAntiForgeryToken` attribute, which works in conjunction with the `Html.AntiForgeryToken` helper method in the view and guards against cross-site request forgery. Cross-site forgery exploits the trust that your user has for your application and it is especially important to use the helper and attribute for authentication requests.
最后要注意的是，我运用了`ValidateAntiForgeryToken`注解属性，该属性与视图中的`Html.AntiForgeryToken`辅助器方法联合工作，防止Cross-Site Request Forgery（CSRF，跨网站请求伪造）的攻击。CSRF会利用应用程序对用户的信任，因此使用这个辅助器和注解属性对于认证请求是特别重要的。

Tip you can learn more about cross-site request forgery at http://en.wikipedia.org/wiki/Cross-site_request_forgery.
提示：更多关于CSRF的信息，请参阅http://en.wikipedia.org/wiki/Cross-site_request_forgery。

My last preparatory step is to create the view that will be rendered to gather credentials from the user. Listing 14-4 shows the contents of the `Views/Account/Login.cshtml` file, which I created by right-clicking the `Index` action method and selecting `Add View` from the pop-up menu.
最后一项准备步骤是创建一个视图，用以收集来自于用户的凭据。清单14-4显示了`Views/Account/Login.cshtml`文件的内容，这是通过右击`Index`动作方法，然后从弹出菜单选择“`Add View`（添加视图）”而创建的。

Listing 14-4. *The Contents of the Login.cshtml File*
清单14-4. *Login.cshtml文件的内容*

```
@model Users.Models.LoginModel
@{ ViewBag.Title = "Login";}
<h2>Log In</h2>

@Html.ValidationSummary()

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

The only notable aspects of this view are using the `Html.AntiForgeryToken` helper and creating a

hidden `input` element to preserve the `returnUrl` argument. In all other respects, this is a standard Razor view, but it completes the preparations for authentication and demonstrates the way that unauthenticated requests are intercepted and redirected. To test the new controller, start the application and request the `/Home/Index` URL. You will be redirected to the `/Account/Login` URL, as shown in Figure 14-2.
该视图唯一要注意的方面是使用了`Html.AntiForgeryToken`辅助器，并创建了一个隐藏的`input`元素，以保护`returnUrl`参数。在其他方面，这是一个标准的Razor视图，但它实现了认证的准备工作，并能演示被拦截且被重定向的未认证请求的情况。为了测试这个新的控制器，启动应用程序，并请求`/Home/Index`。你将被重定向到`/Account/Login`，如图14-2所示。

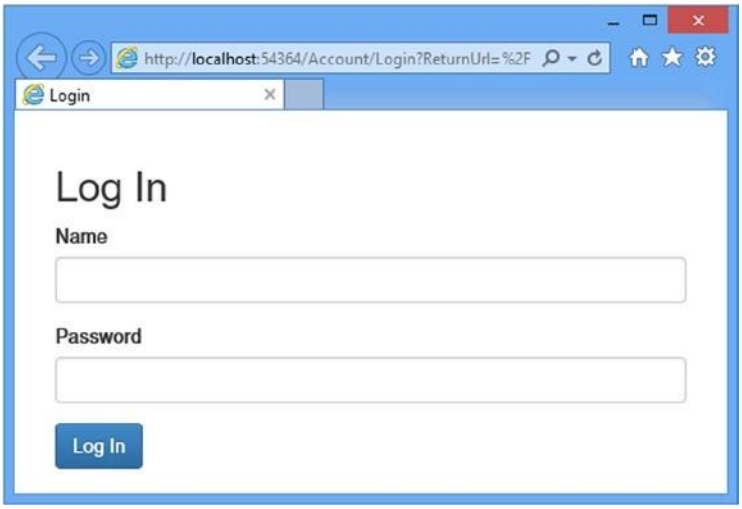

Figure 14-2. *Prompting the user for authentication credentials*
图14-2. *提示用户输入认证凭据*

14.2.3 Adding User Authentication
14.2.3 添加用户认证

Requests for protected URLs are being correctly redirected to the `Account` controller, but the credentials provided by the user are not yet used for authentication. In Listing 14-5, you can see how I have completed the implementation of the `Login` action.
发送给受保护URL的请求会被正确地重定向到`Account`控制器，但由用户提供的凭据尚未被用于认证。从清单14-5可以看出如何完成`Login`动作的实现。

Listing 14-5. *Adding Authentication to the AccountController.cs File*
清单14-5. *在AccountController.cs文件中添加认证*

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

```
using System.Web;

namespace Users.Controllers {

    [Authorize]
    public class AccountController : Controller {

        [AllowAnonymous]
        public ActionResult Login(string returnUrl) {
            ViewBag.returnUrl = returnUrl;
            return View();
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> Login(LoginModel details, string returnUrl) {
            if (ModelState.IsValid) {
                AppUser user = await UserManager.FindAsync(details.Name,
                           details.Password);
                if (user == null) {
                    ModelState.AddModelError("", "Invalid name or password.");
                } else {
                    ClaimsIdentity ident = await UserManager.CreateIdentityAsync(user,
                              DefaultAuthenticationTypes.ApplicationCookie);
                    AuthManager.SignOut();
                    AuthManager.SignIn(new AuthenticationProperties {
                        IsPersistent = false}, ident);
                     return Redirect(returnUrl);
                }
            }
            ViewBag.returnUrl = returnUrl;
            return View(details);
        }

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

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

```
            }
        }
    }
}
```

The simplest part is checking the credentials, which I do through the `FindAsync` method of the `AppUserManager` class, which you will remember as the user manager class from Chapter 13:
最简单的部分是检查凭据，这是通过`AppUserManager`类的`FindAsync`方法来做的，你可能还记得，`AppUserManager`是第13章的用户管理器类。

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

I will be using the `AppUserManager` class repeatedly in the `Account` controller, so I defined a property called `UserManager` that returns the instance of the class using the `GetOwinContext` extension method for the `HttpContext` class, just as I did for the Admin controller in Chapter 13.
我会在`Account`控制器中反复使用`AppUserManager`类，因此定义了一个名称为的`UserManager`属性，它使用`HttpContext`类的`GetOwinContext`扩展方法来返回`AppUserManager`类的实例。

The `FindAsync` method takes the account name and password supplied by the user and returns an instance of the user class (`AppUser` in the example application) if the user account exists *and* if the password is correct. If there is no such account or the password doesn't match the one stored in the database, then the `FindAsync` method returns null, in which case I add an error to the model state that tells the user that something went wrong.
`FindAsync`方法以用户提供的账号名和口令为参数，并在该用户账号存在*且*口令正确时，返回一个用户类（此例中的`AppUser`）的实例。如果无此账号，或者与数据库存储的不匹配，那么`FindAsync`方法返回空（`null`），出现这种情况时，我给模型状态添加了一条错误消息，告诉用户可能出错了。

If the `FindAsync` method does return an `AppUser` object, then I need to create the cookie that the browser will send in subsequent requests to show they are authenticated. Here are the relevant statements:
如果`FindAsync`方法确实返回了`AppUser`对象，那么则需要创建Cookie，浏览器会在后继的请求中发送这个Cookie，表明他们是已认证的。以下是有关语句：

```
...
ClaimsIdentity ident = await UserManager.CreateIdentityAsync(user,
       DefaultAuthenticationTypes.ApplicationCookie);
AuthManager.SignOut();
AuthManager.SignIn(new AuthenticationProperties {IsPersistent = false}, ident);
return Redirect(returnUrl);
...
```

The first step is to create a `ClaimsIdentity` object that identifies the user. The `ClaimsIdentity` class is the ASP.NET Identity implementation of the `IIdentity` interface that I described in Table 14-4 and that you can see used in the "Using Roles for Authorization" section later in this chapter.

第一步是创建一个标识该用户的ClaimsIdentity对象。ClaimsIdentity类是表14-4所描述的IIdentity接口的ASP.NET Identity实现，可以在本章稍后的“使用角色授权”小节中看到它的使用。

Tip Don’t worry about why the class is called ClaimsIdentity at the moment. I explain what claims are and how they can be used in Chapter 15.

提示：此刻不必关心这个类为什么要调用ClaimsIdentity，第15章会解释什么是声明（Claims），并介绍如何使用它们。

Instances of ClaimsIdentity are created by calling the user manager CreateIdentityAsync method, passing in a user object and a value from the DefaultAuthenticationTypes enumeration. The ApplicationCookie value is used when working with individual user accounts.

ClaimsIdentity的实例是调用用户管理器的CreateIdentityAsync方法而创建的，在其中传递了一个用户对象和DefaultAuthenticationTypes枚举中的一个值。在使用个别用户账号进行工作时，会用到ApplicationCookie值。

The next step is to invalidate any existing authentication cookies and create the new one. I defined the AuthManager property in the controller because I’ll need access to the object it provides repeatedly as I build the functionality in this chapter. The property returns an implementation of the IAuthenticationManager interface that is responsible for performing common authentication options. I have described the most useful methods provided by the IAuthenticationManager interface in Table 14-5.

下一个步骤是让已认证的Cookie失效，并创建一个新的Cookie。我在该控制器中定义了AuthManager属性，因为在建立本章功能过程中，需要反复访问它所提供的对象。该属性返回的是IAuthenticationManager接口的实现，它负责执行常规的认证选项。表14-5中描述了IAuthenticationManager接口所提供的最有用的方法。

Table 14-5. *The Most Useful Methods Defined by the IAuthenticationManager Interface*

表14-5. *IAuthenticationManager接口定义的最有用的方法*

Name **名称**	**Description** **描述**
SignIn(options, identity)	Signs the user in, which generally means creating the cookie that identifies authenticated requests 签入用户，这通常意味着要创建用来标识已认证请求的Cookie
SignOut()	Signs the user out, which generally means invalidating the cookie that identifies authenticated requests 签出用户，这通常意味着使标识已认证用户的Cookie失效

The arguments to the SignIn method are an AuthenticationProperties object that configures the authentication process and the ClaimsIdentity object. I set the IsPersistent property defined by the AuthenticationProperties object to true to make the authentication cookie persistent at the browser, meaning that the user doesn’t have to authenticate again when starting a new session. (There are other properties defined by the AuthenticationProperties class, but the IsPersistent property is the only one that is widely used at the moment.)

SignIn方法的参数是一个AuthenticationProperties对象，用以配置认证过程以及ClaimsIdentity对象。我将AuthenticationProperties对象定义的IsPersistent属性设置为true，以使认证Cookie在浏览器中是持久化的，意即用户在开始新会话时，不必再次进行认证。

（AuthenticationProperties类还定义了一些其他属性，但IsPersistent属性是此刻唯一要广泛使用的一个属性。）

The final step is to redirect the user to the URL they requested before the authentication process started, which I do by calling the Redirect method.
最后一步是将用户重定向到他们在认证过程开始之前所请求的URL，这是通过调用Redirect方法实现的。

CONSIDERING TWO-FACTOR AUTHENTICATION

考虑双因子认证

I have performed single-factor authentication in this chapter, which is where the user is able to authenticate using a single piece of information known to them in advance: the password.
在本章中，我实行的是单因子认证，在这种场合中，用户只需使用一个他们预知的单一信息片段：口令，便能够进行认证。

ASP.NET Identity also supports two-factor authentication, where the user needs something extra, usually something that is given to the user at the moment they want to authenticate. The most common examples are a value from a SecureID token or an authentication code that is sent as an e-mail or text message (strictly speaking, the two factors can be anything, including fingerprints, iris scans, and voice recognition, although these are options that are rarely required for most web applications).
ASP.NET Identity还支持双因子认证，在这种情况下，用户需要一些附加信息，通常是在他们需要认证时才发给他们的某种信息。最常用的例子是SecureID令牌的值，或者是通过E-mail发送的认证码或文本消息（严格地讲，第二因子可以是任何东西，包括指纹、眼瞳扫描、声音识别等，尽管这些是在大多数Web应用程序中很少需要用到的选项。）

Security is increased because an attacker needs to know the user's password *and* have access to whatever provides the second factor, such an e-mail account or cell phone.
这样增加了安全性，因为攻击者需要知道用户的口令，*并且*能够对提供第二因子的客户端进行访问，如E-mail账号或移动电话等。

I don't show two-factor authentication in the book for two reasons. The first is that it requires a lot of preparatory work, such as setting up the infrastructure that distributes the second-factor e-mails and texts and implementing the validation logic, all of which is beyond the scope of this book.
本章不演示双因子认证有两个原因。第一是它需要许多准备工作，例如要建立分发第二因子的邮件和文本的基础架构，并实现验证逻辑，这些都超出了本书的范围。

The second reason is that two-factor authentication forces the user to remember to jump through an additional hoop to authenticate, such as remembering their phone or keeping a security token nearby, something that isn't always appropriate for web applications. I carried a SecureID token of one sort or another for more than a decade in various jobs, and I lost count of the number of times that I couldn't log in to an employer's system because I left the token at home.
第二个原因是双因子认证强制用户要记住一个额外的认证令牌，例如，要记住他们的电话，或者将安全令牌带在身边，这对Web应用程序而言，并非总是合适的。我十几年在各种工作中都带着这种或那种令牌，而且我有数不清的次数无法登录雇员系统，因为我将令牌丢在了家里。

If you are interested in two-factor security, then I recommend relying on a third-party provider such as Google for authentication, which allows the user to choose whether they want the additional security (and inconvenience) that two-factor authentication provides. I demonstrate third-party authentication in Chapter 15.

如果对双因子安全性有兴趣，那么我建议你依靠第三方提供器，例如Google认证，它允许用户选择是否希望使用双因子提供的附加安全性（而且是不方便的）。第15章将演示第三方认证。

14.2.4 Testing Authentication
14.2.4 测试认证

To test user authentication, start the application and request the `/Home/Index` URL. When redirected to the `/Account/Login` URL, enter the details of one of the users I listed at the start of the chapter (for instance, the name `joe` and the password `MySecret`). Click the Log In button, and your browser will be redirected back to the `/Home/Index` URL, but this time it will submit the authentication cookie that grants it access to the action method, as shown in Figure 14-3.

为了测试用户认证，启动应用程序，并请求`/Home/Index` URL。当被重定向到`/Account/Login` URL时，输入本章开始时列出的一个用户的细节（例如，姓名为`joe`，口令为`MySecret`）。点击“Log In（登录）”按钮，你的浏览器将被重定向，回到`/Home/Index` URL，但这次它将递交认证Cookie，被准予访问该动作方法，如图14-3所示。

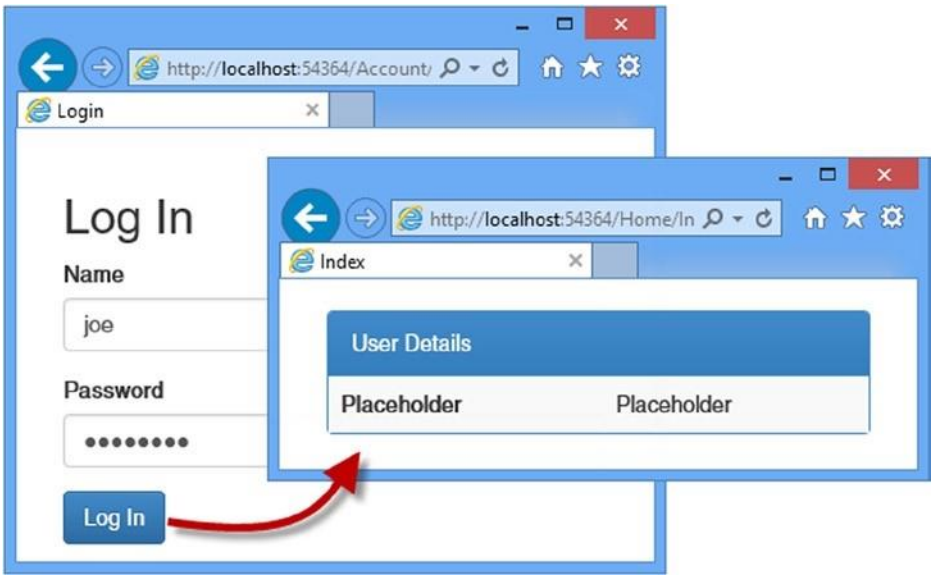

Figure 14-3. *Authenticating a user*

图 14-3. *认证用户*

Tip You can use the browser F12 tools to see the cookies that are used to identify authenticated requests.

提示：可以用浏览器的F12工具，看到用来标识已认证请求的Cookie。

14.3 Authorizing Users with Roles

14.3 以角色授权用户

In the previous section, I applied the `Authorize` attribute in its most basic form, which allows any authenticated user to execute the action method. In this section, I will show you how to refine authorization to give finer-grained control over which users can perform which actions. Table 14-6 puts authorization in context.

上一小节以最基本的形式运用了`Authorize`注解属性，这允许任何已认证用户执行动作方法。在本小节中，将展示如何精炼授权，以便在用户能够执行的动作上有更细粒度的控制。表14-6描述了授权的情形。

Table 14-6. *Putting Authorization in Context*

表16-4. *授权情形*

Question **问题**	**Answer** **答案**
What is it? 什么是授权？	Authorization is the process of granting access to controllers and action methods to certain users, generally based on role membership. 授权是将控制器和动作的准许访问限制到特定用户，通常是基于角色的成员
Why should I care? 为何要关注它？	Without roles, you can differentiate only between users who are authenticated and those who are not. Most applications will have different types of users, such as customers and administrators. 没有角色，你只能在已认证用户和未认证用户之间加以区分。大多数应用程序均有不同类型的用户，例如客户和管理员等
How is it used by the MVC framework? 在MVC框架中如何使用它？	Roles are used to enforce authorization through the `Authorize` attribute, which is applied to controllers and action methods. 角色通过`Authorize`注解属性可用于强制授权，`Authorize`可用于控制器和动作方法

Tip In Chapter 15, I show you a different approach to authorization using *claims*, which are an advanced ASP.NET Identity feature.

提示：第15章将使用*Claims*（声明）来演示不同的授权办法，Claims是一种高级的ASP.NET Identity特性。

14.3.1 Adding Support for Roles

14.3.1 添加角色支持

ASP.NET Identity provides a strongly typed base class for accessing and managing roles called `RoleManager<T>`, where `T` is the implementation of the `IRole` interface supported by the storage mechanism used to represent roles. The Entity Framework uses a class called `IdentityRole` to implement the `IRole` interface, which defines the properties shown in Table 14-7.

ASP.NET Identity为访问和管理角色提供了一个强类型的基类，叫做`RoleManager<T>`，其中`T`是`IRole`接口的实现，该实现得到了用来表示角色的存储机制的支持。Entity Framework实现了`IRole`接口，使用的是一个名称为`IdentityRole`的类，它定义了如表14-7所示的属性。

Table 14-7. *The Properties Defined by the IdentityRole Class*

表14-7. *IdentityRole类所定义的属性*

Name 名称	Description 描述
Id	Defines the unique identifier for the role 定义角色的唯一标识符
Name	Defines the name of the role 定义角色名称
Users	Returns a collection of IdentityUserRole objects that represents the members of the role 返回一个代表角色成员的IdentityUserRole对象集合

I don't want to leak references to the IdentityRole class throughout my application because it ties me to the Entity Framework for storing role data, so I start by creating an application-specific role class that is derived from IdentityRole. I added a class file called AppRole.cs to the Models folder and used it to define the class shown in Listing 14-6.

我不希望在整个应用程序中都暴露对IdentityRole类的引用，因为它为了存储角色数据，将我绑定到了Entity Framework。为此，我首先创建了一个应用程序专用的角色类，它派生于IdentityRole。我在Models文件夹中添加了一个类文件，名称为AppRole.cs，并用它定义了这个类，如清单14-6所示。

Listing 14-6*. The Contents of the AppRole.cs File*

清单14-6. *AppRole文件的内容*

```
using Microsoft.AspNet.Identity.EntityFramework;

namespace Users.Models {
    public class AppRole : IdentityRole {

        public AppRole() : base() {}

        public AppRole(string name) : base(name) { }
    }
}
```

The RoleManager<T> class operates on instances of the IRole implementation class through the methods and properties shown in Table 14-8.

RoleManager<T>类通过表14-8所示的方法和属性对IRole实现类的实例进行操作。

Table 14-8. *The Members Defined by the RoleManager<T> Class*

表14-8. *RoleManager<T>类定义的成员*

Name 名称	Description 描述
CreateAsync(role)	Creates a new role 创建一个新角色
DeleteAsync(role)	Deletes the specified role

	删除指定角色
`FindByIdAsync(id)`	Finds a role by its `ID` 找到指定`ID`的角色
`FindByNameAsync(name)`	Finds a role by its name 找到指定名称的角色
`RoleExistsAsync(name)`	Returns `true` if a role with the specified name exists 如果存在指定名称的角色，返回`true`
`UpdateAsync(role)`	Stores changes to the specified role 将修改存储到指定角色
`Roles`	Returns an enumeration of the roles that have been defined 返回已被定义的角色枚举

These methods follow the same basic pattern of the `UserManager<T>` class that I described in Chapter 13. Following the pattern I used for managing users, I added a class file called `AppRoleManager.cs` to the `Infrastructure` folder and used it to define the class shown in Listing 14-7.
这些方法与第13章描述的`UserManager<T>`类有同样的基本模式。按照对管理用户所采用的模式，我在`Infrastructure`文件夹中添加了一个类文件，名称为`AppRoleManager.cs`，用它定义了如清单14-7所示的类。

Listing 14-7. *The Contents of the AppRoleManager.cs File*
清单14-7. *AppRoleManager.cs文件的内容*

```
using System;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin; using Users.Models;

namespace Users.Infrastructure {

    public class AppRoleManager : RoleManager<AppRole>, IDisposable {

        public AppRoleManager(RoleStore<AppRole> store) : base(store) { }

        public static AppRoleManager Create(
                IdentityFactoryOptions<AppRoleManager> options,
                IOwinContext context) {
            return new AppRoleManager(new
                    RoleStore<AppRole>(context.Get<AppIdentityDbContext>()));
        }
    }
}
```

This class defines a `Create` method that will allow the OWIN start class to create instances for each

request where Identity data is accessed, which means I don't have to disseminate details of how role data is stored throughout the application. I can just obtain and operate on instances of the AppRoleManager class. You can see how I have registered the role manager class with the OWIN start class, IdentityConfig, in Listing 14-8. This ensures that instances of the AppRoleManager class are created using the same Entity Framework database context that is used for the AppUserManager class.
这个类定义了一个Create方法，它让OWIN启动类能够为每一个访问Identity数据的请求创建实例，这意味着在整个应用程序中，我不必散布如何存储角色数据的细节，却能获取AppRoleManager类的实例，并对其进行操作。在清单14-8中可以看到如何用OWIN启动类（IdentityConfig）来注册角色管理器类。这样能够确保，可以使用与AppUserManager类所用的同一个Entity Framework数据库上下文，来创建AppRoleManager类的实例。

Listing 14-8. *Creating Instances of the AppRoleManager Class in the IdentityConfig.cs File*
清单14-8. *在IdentityConfig.cs文件中创建AppRoleManager类的实例*

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;
using Users.Infrastructure;

namespace Users {
    public class IdentityConfig {
        public void Configuration(IAppBuilder app) {

            app.CreatePerOwinContext<AppIdentityDbContext>(AppIdentityDbContext.Create);
            app.CreatePerOwinContext<AppUserManager>(AppUserManager.Create);
            app.CreatePerOwinContext<AppRoleManager>(AppRoleManager.Create);

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

14.3.2 Creating and Deleting Roles
14.3.2 创建和删除角色

Having prepared the application for working with roles, I am going to create an administration tool for managing them. I will start the basics and define action methods and views that allow roles to be created and deleted. I added a controller called RoleAdmin to the project, which you can see in Listing 14-9.
现在已经做好了应用程序使用角色的准备，我打算创建一个管理工具来管理角色。首先从基本的开始，定义能够创建和删除角色的动作方法和视图。我在项目中添加了一个控制器，名称为RoleAdmin，如清单14-9所示。

Listing 14-9. *The Contents of the RoleAdminController.cs File*

清单14-9. *RoleAdminController.cs文件的内容*

```
using System.ComponentModel.DataAnnotations;
using System.Linq; using System.Threading.Tasks;
using System.Web; using System.Web.Mvc; using Microsoft.AspNet.Identity; using
Microsoft.AspNet.Identity.Owin;
using Users.Infrastructure; using Users.Models;

namespace Users.Controllers {
    public class RoleAdminController : Controller {

        public ActionResult Index() {
             return View(RoleManager.Roles);
        }

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

        [HttpPost]
        public async Task<ActionResult> Create([Required]string name) {
            if (ModelState.IsValid) {
                 IdentityResult result
                    = await RoleManager.CreateAsync(new AppRole(name));
                if (result.Succeeded) {
                    return RedirectToAction("Index");
                } else {
                    AddErrorsFromResult(result);
                }
            }
            return View(name);
        }

        [HttpPost]
        public async Task<ActionResult> Delete(string id) {
            AppRole role = await RoleManager.FindByIdAsync(id);
            if (role != null) {
                IdentityResult result = await RoleManager.DeleteAsync(role);
                if (result.Succeeded) {
                    return RedirectToAction("Index");
                } else {
                    return View("Error", result.Errors);
                }
```

```
            } else {
                return View("Error", new string[] { "Role Not Found" });
            }
        }

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

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

        private AppRoleManager RoleManager {
            get {
                return HttpContext.GetOwinContext().GetUserManager<AppRoleManager>();
            }
        }
    }
}
```

I have applied many of the same techniques that I used in the `Admin` controller in Chapter 13, including a `UserManager` property that obtains an instance of the `AppUserManager` class and an `AddErrorsFromResult` method that processes the errors reported in an `IdentityResult` object and adds them to the model state.
这里运用了许多第13章中`Admin`控制器所采用的同样技术，包括一个`UserManager`属性，用于获取`AppUserManager`类的实例；和一个`AddErrorsFromResult`方法，用来处理`IdentityResult`对象所报告的消息，并将消息添加到模型状态。

I have also defined a `RoleManager` property that obtains an instance of the `AppRoleManager` class, which I used in the action methods to obtain and manipulate the roles in the application. I am not going to describe the action methods in detail because they follow the same pattern I used in Chapter 13, using the `AppRoleManager` class in place of `AppUserManager` and calling the methods I described in Table 14-8.
我还定义了`RoleManager`属性，用来获取`AppRoleManager`类的实例，在动作方法中用该实例获取并维护应用程序的角色。我不打算详细描述这些动作方法，因为它们遵循着与第13章同样的模式，在使用`AppUserManager`的地方使用了`AppRoleManager`类，调用的是表14-8中的方法。

14.3.3 Creating the Views
14.3.3 创建视图

The views for the `RoleAdmin` controller are standard HTML and Razor markup, but I have included

them in this chapter so that you can re-create the example. I want to display the names of the users who are members of each role. The Entity Framework `IdentityRole` class defines a `Users` property that returns a collection of `IdentityUserRole` user objects representing the members of the role. Each `IdentityUserRole` object has a `UserId` property that returns the unique `ID` of a user, and I want to get the username for each ID. I added a class file called `IdentityHelpers.cs` to the `Infrastructure` folder and used it to define the class shown in Listing 14-10.
`RoleAdmin`控制器的视图是标准的HTML和Razor标记，但我还是将它们包含在本章之中，以便你能够重建本章的示例。我希望显示每个角色中成员的用户名。Entity Framework的`IdentityRole`类中定义了一个`Users`属性，它能够返回表示角色成员的`IdentityUserRole`用户对象集合。每一个`IdentityUserRole`对象都有一个`UserId`属性，它返回一个用户的唯一`ID`，不过，我希望得到的是每个`ID`所对应的用户名。我在`Infrastructure`文件夹中添加了一个类文件，名称为`IdentityHelpers.cs`，用它定义了如清单14-10所示的类。

Listing 14-10. *The Contents of the IdentityHelpers.cs File*
清单14-10. *IdentityHelpers.cs文件的内容*

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

namespace Users.Infrastructure {
    public static class IdentityHelpers {

       public static MvcHtmlString GetUserName(this HtmlHelper html, string id) {
           AppUserManager mgr
               =  HttpContext.Current.GetOwinContext().GetUserManager<AppUserManager>();
           return new MvcHtmlString(mgr.FindByIdAsync(id).Result.UserName);
       }
   }
}
```

Custom HTML helper methods are defined as extensions on the `HtmlHelper` class. My helper, which is called `GetUsername`, takes a string argument containing a user `ID`, obtains an instance of the `AppUserManager` through the `GetOwinContext.GetUserManager` method (where `GetOwinContext` is an extension method on the `HttpContext` class), and uses the `FindByIdAsync` method to locate the `AppUser` instance associated with the `ID` and to return the value of the `UserName` property.
这个自定义的HTML辅助器方法，是作为`HtmlHelper`类的扩展进行定义的。该辅助器的名称为`GetUsername`，以一个含有用户ID的字符串为参数，通过`GetOwinContext.GetUserManager`方法获取`AppUserManager`的一个实例（其中`GetOwinContext`是`HttpContext`类的扩展方法），并使用`FindByIdAsync`方法定位与`ID`相关联的`AppUser`实例，然后返回`UserName`属性的值。

Listing 14-11 shows the contents of the `Index.cshtml` file from the `Views/RoleAdmin` folder, which I created by right-clicking the `Index` action method in the code editor and selecting Add View from the pop-up menu.
清单14-11显示了`Views/RoleAdmin`文件夹中`Index.cshtml`文件的内容，这是通过在代码编辑器中右

击Index动作，并从弹出菜单中选择“Add View（添加视图）”来创建的。

Listing 14-11. *The Contents of the Index.cshtml File in the Views/RoleAdmin Folder*

清单14-11. *Views/RoleAdmin文件夹中Index.cshtml文件的内容*

```
@using Users.Models
@using Users.Infrastructure
@model IEnumerable<AppRole>
@{ ViewBag.Title = "Roles"; }

<div class="panel panel-primary">
    <div class="panel-heading">Roles</div>
    <table class="table table-striped">
        <tr><th>ID</th><th>Name</th><th>Users</th><th></th></tr>
        @if (Model.Count() == 0) {
            <tr><td colspan="4" class="text-center">No Roles</td></tr>
        } else {
            foreach (AppRole role in Model) {
                <tr>
                    <td>@role.Id</td>
                    <td>@role.Name</td>
                    <td>
                        @if (role.Users == null || role.Users.Count == 0) {
                            @: No Users in Role
                        } else {
                            <p>@string.Join(", ", role.Users.Select(x =>
                                Html.GetUserName(x.UserId)))</p>
                        }
                    </td>
                    <td>
                        @using (Html.BeginForm("Delete", "RoleAdmin",
                            new { id = role.Id })) {
                            @Html.ActionLink("Edit", "Edit", new { id = role.Id },
                                    new { @class = "btn btn-primary btn-xs" })
                            <button class="btn btn-danger btn-xs"
                                   type="submit">
                                Delete
                            </button>
                        }
                    </td>
                </tr>
            }
        }
    </table>
```

```
</div>
@Html.ActionLink("Create", "Create", null, new { @class = "btn btn-primary" })
```

This view displays a list of the roles defined by the application, along with the users who are members, and I use the `GetUserName` helper method to get the name for each user.
该视图显示了一个由应用程序定义的角色列表，且带有成员用户，我用`GetUserName`辅助器方法获取了每个用户的用户名。

Listing 14-12 shows the `Views/RoleAdmin/Create.cshtml` file, which I created to allow new roles to be created.
清单14-12显示了`Views/RoleAdmin/Create.cshtml`文件，这是用来创建新角色的视图。

Listing 14-12. *The Contents of the Create.cshtml File in the Views/RoleAdmin Folder*
清单14-12. *Views/RoleAdmin文件夹中Create.cshtml文件的内容*

```
@model string
@{ ViewBag.Title = "Create Role";}
<h2>Create Role</h2>
@Html.ValidationSummary(false)
@using (Html.BeginForm()) {
    <div class="form-group">
        <label>Name</label>
        <input name="name" value="@Model" class="form-control" />
    </div>
    <button type="submit" class="btn btn-primary">Create</button>
    @Html.ActionLink("Cancel", "Index", null, new { @class = "btn btn-default" })
}
```

The only information required to create a new view is a name, which I gather using a standard input element and submit the value to the `Create` action method.
创建该视图需要的唯一信息是角色名，我用标准的input元素进行采集，并将该值递交给`Create`动作方法。

14.3.4 Testing Creating and Deleting Roles
14.3.4 测试角色的创建和删除

To test the new controller, start the application and navigate to the `/RoleAdmin/Index` URL. To create a new role, click the `Create` button, enter a name in the input element, and click the second `Create` button. The new view will be saved to the database and displayed when the browser is redirected to the `Index` action, as shown in Figure 14-4. You can remove the role from the application by clicking the `Delete` button.
为了测试新的控制器，启动应用程序并导航到`/RoleAdmin/Index` URL。为了创建一个新的角色，点击“Create”按钮，在input元素中输入一个角色名，然后点击第二个“Create”按钮。新角色将被保存到数据库，并在浏览器被重定向到`Index`动作时显示出来，如图14-4所示。可以点击“Delete”按钮将该角色从应用程序中删除。

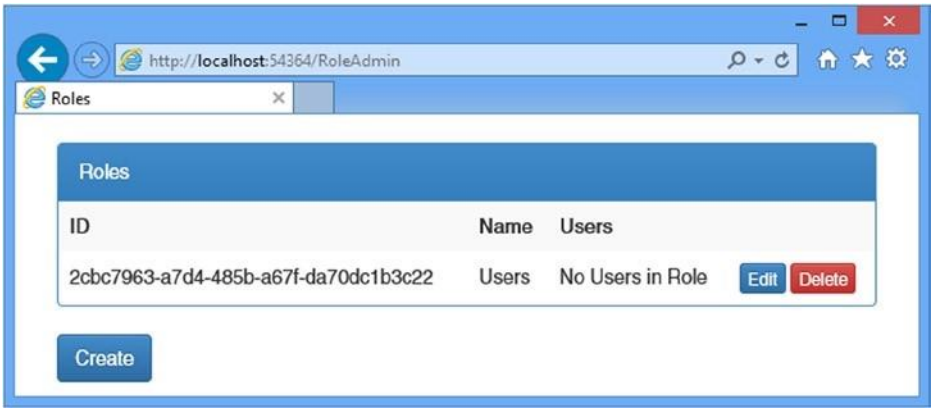

Figure 14-4. *Creating a new role*

图 14-4. *创建新角色*

14.3.5 Managing Role Memberships
14.3.5 管理角色成员

To authorize users, it isn't enough to just create and delete roles; I also have to be able to manage role memberships, assigning and removing users from the roles that the application defines. This isn't a complicated process, but it invokes taking the role data from the `AppRoleManager` class and then calling the methods defined by the `AppUserMangager` class that associate users with roles.
为了授权用户，仅仅创建和删除角色还不够。还必须能够管理角色成员，从应用程序定义的角色中指定和除去用户。这不是一个复杂的过程，但它要从`AppRoleManager`类获取角色数据，然后调用将用户与角色关联在一起的`AppUserMangager`类所定义的方法。

I started by defining view models that will let me represent the membership of a role and receive a new set of membership instructions from the user. Listing 14-13 shows the additions I made to the `UserViewModels.cs` file.
我首先定义了视图模型，这让我能够表示一个角色中的成员，并能够从用户那里接收一组新成员的指令。清单14-13显示了在`UserViewModels.cs`文件中所做的添加。

Listing 14-13. *Adding View Models to the UserViewModels.cs File*

清单14-13. *添加到UserViewModels.cs文件的视图模型*

```
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;

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

    public class RoleEditModel {
        public AppRole Role { get; set; }
        public IEnumerable<AppUser> Members { get; set; }
        public IEnumerable<AppUser> NonMembers { get; set; }
    }

     public class RoleModificationModel {
        [Required]
        public string RoleName { get; set; }
        public string[] IdsToAdd { get; set; }
        public string[] IdsToDelete { get; set; }
    }
}
```

The RoleEditModel class will let me pass details of a role and details of the users in the system, categorized by membership. I use AppUser objects in the view model so that I can extract the name and ID for each user in the view that will allow memberships to be edited. The RoleModificationModel class is the one that I will receive from the model binding system when the user submits their changes. It contains arrays of user IDs rather than AppUser objects, which is what I need to change role memberships.
RoleEditModel类使我能够在系统中传递角色细节和用户细节，按成员进行归类。我在视图模型中使用了AppUser对象，以使我在编辑成员的视图中能够为每个用户提取用户名和ID。RoleModificationModel类是在用户递交他们的修改时，从模型绑定系统接收到的一个类。它含有用户ID的数组，而不是AppUser对象，这是对角色成员进行修改所需要的。

Having defined the view models, I can add the action methods to the controller that will allow role memberships to be defined. Listing 14-14 shows the changes I made to the RoleAdmin controller.
定义了视图模型之后，便可以在控制器中添加动作方法，以便定义角色成员。清单14-14显示了我对AppUser控制器所做的修改。

Listing 14-14. *Adding Action Methods in the RoleAdminController.cs File*
清单14-14. *在RoleAdminController.cs文件中添加动作方法*

```
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
```

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

namespace Users.Controllers {
    public class RoleAdminController : Controller {

          // ...other action methods omitted for brevity...
          // ...出于简化，这里忽略了其他动作方法 ...

        public async Task<ActionResult> Edit(string id) {
            AppRole role = await RoleManager.FindByIdAsync(id);
            string[] memberIDs = role.Users.Select(x => x.UserId).ToArray();
            IEnumerable<AppUser> members
                    = UserManager.Users.Where(x => memberIDs.Any(y => y == x.Id));
            IEnumerable<AppUser> nonMembers = UserManager.Users.Except(members);
            return View(new RoleEditModel {
                Role = role,
                Members = members,
                NonMembers = nonMembers
            });
        }

        [HttpPost]
        public async Task<ActionResult> Edit(RoleModificationModel model) {
            IdentityResult result;
            if (ModelState.IsValid) {
                foreach (string userId in model.IdsToAdd ?? new string[] { }) {
                    result = await UserManager.AddToRoleAsync(userId, model.RoleName);
                    if (!result.Succeeded) {
                        return View("Error", result.Errors);
                    }
                }
                foreach (string userId in model.IdsToDelete ?? new string[] { }) {
                     result = await UserManager.RemoveFromRoleAsync(userId,
                        model.RoleName);
                     if (!result.Succeeded) {
                        return View("Error", result.Errors);
                    }
                }
                return RedirectToAction("Index");
```

```
            }
            return View("Error", new string[] { "Role Not Found" });
        }

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

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

        private AppRoleManager RoleManager {
            get {
                return HttpContext.GetOwinContext().GetUserManager<AppRoleManager>();
            }
        }
    }
}
```

The majority of the code in the GET version of the Edit action method is responsible for generating the sets of members and nonmembers of the selected role, which is done using LINQ. Once I have grouped the users, I call the View method, passing a new instance of the RoleEditModel class I defined in Listing 14-13.
GET版Edit动作方法的主要代码是负责生成一组所选角色的成员和非成员，这是用LINQ完成的。一旦对用户进行了分组，便调用View方法，为其传递了清单14-13所定义的RoleEditModel类的新实例。

The POST version of the Edit method is responsible for adding and removing users to and from roles. The AppUserManager class inherits a number of role-related methods from its base class, which I have described in Table 14-9.
POST版的Edit方法是负责从角色中添加和删除用户。AppUserManager类从它的基类继承了几个与角色有关的方法，描述于表14-9。

Table 14-9. *The Role-Related Methods Defined by the UserManager<T> Class*
表14-9. *UserManager<T>类中所定义的与角色有关的方法*

Name **名称**	**Description** **描述**
AddToRoleAsync(id, name)	Adds the user with the specified ID to the role with the specified name 将指定ID的用户添加到指定name的角色
GetRolesAsync(id)	Returns a list of the names of the roles of which the user with the specified ID is a member

	返回指定ID用户所在的角色名列表
IsInRoleAsync(id, name)	Returns true if the user with the specified ID is a member of the role with the specified name 如果指定ID的用户是指定name角色的成员，返回true
RemoveFromRoleAsync(id, name)	Removes the user with the specified ID as a member from the role with the specified name 在指定name角色的成员中除去指定ID的用户

An oddity of these methods is that the role-related methods operate on user IDs and *role names*, even though roles also have unique identifiers. It is for this reason that my RoleModificationModel view model class has a RoleName property.
这些方法的奇怪之处在于，与角色有关的方法都根据“用户ID”和“角色name（角色名）”进行操作，尽管角色也具有唯一标识符（ID）。这也是在RoleModificationModel视图模型类中使用RoleName属性的原因。

Listing 14-15 shows the view for the Edit.cshtml file, which I added to the Views/RoleAdmin folder and used to define the markup that allows the user to edit role memberships.
清单14-15显示了Edit.cshtml文件的视图，该视图放在Views/RoleAdmin文件中，用它定义了让用户编辑角色成员的标记。

Listing 14-15. *The Contents of the Edit.cshtml File in the Views/RoleAdmin Folder*
清单14-15. *Views/RoleAdmin文件夹中Edit.cshtml文件的内容*

```
@using Users.Models
@model RoleEditModel
@{ ViewBag.Title = "Edit Role";}
@Html.ValidationSummary()
@using (Html.BeginForm()) {
    <input type="hidden" name="roleName" value="@Model.Role.Name" />
    <div class="panel panel-primary">
        <div class="panel-heading">Add To @Model.Role.Name</div>
        <table class="table table-striped">
            @if (Model.NonMembers.Count() == 0) {
                <tr><td colspan="2">All Users Are Members</td></tr>
            } else {
                <tr><td>User ID</td><td>Add To Role</td></tr>
                foreach (AppUser user in Model.NonMembers) {
                    <tr>
                        <td>@user.UserName</td>
                        <td>
                            <input type="checkbox" name="IdsToAdd" value="@user.Id">
                        </td>
                    </tr>
                }
```

```
            }
        </table>
    </div>
    <div class="panel panel-primary">
        <div class="panel-heading">Remove from @Model.Role.Name</div>
        <table class="table table-striped">
            @if (Model.Members.Count() == 0) {
                <tr><td colspan="2">No Users Are Members</td></tr>
            } else {
                <tr><td>User ID</td><td>Remove From Role</td></tr>
                foreach (AppUser user in Model.Members) {
                    <tr>
                        <td>@user.UserName</td>
                        <td>
                            <input type="checkbox" name="IdsToDelete" value="@user.Id">
                        </td>
                    </tr>
                }
            }
        </table>
    </div>
    <button type="submit" class="btn btn-primary">Save</button>
    @Html.ActionLink("Cancel", "Index", null, new { @class = "btn btn-default" })
}
```

The view contains two tables: one for users who are not members of the selected role and one for those who are members. Each user's name is displayed along with a check box that allows the membership to be changed.
该视图含有两个表格：一个用于不是所选角色成员的用户，一个是所选角色成员的用户。每个被显示出来的用户名称旁边都有一个复选框，可以修改其成员情况。

14.3.6 Testing Editing Role Membership
14.3.6 测试角色成员的编辑

Adding the `AppRoleManager` class to the application causes the Entity Framework to delete the contents of the database and rebuild the schema, which means that any users you created in the previous chapter have been removed. So that there are users to assign to roles, start the application and navigate to the `/Admin/Index` URL and create users with the details in Table 14-10.
在应用程序中添加`AppRoleManager`类会导致Entity Framework删除数据库的内容，并重建数据库架构，这意味着在上一章创建的用户都会被删除。因此，为了有用户可以赋予角色，需启动应用程序并导航到`/Admin/Index` URL，先创建一些如表14-10所示的用户。

Table 14-10. *The Values for Creating Example User*

表14-10. *创建示例用户的值*

Name **用户名**	**Email** **E-mail**	**Password** **口令**
Alice	alice@example.com	MySecret
Bob	bob@example.com	MySecret
Joe	joe@example.com	MySecret

Tip deleting the user database is fine for an example application but tends to be a problem in real applications. I show you how to gracefully manage changes to the database schema in Chapter 15.

提示：删除用户对示例应用程序而言没什么问题，但对实际应用程序来说就是一个问题了。第15章将演示如何优雅地修改数据库架构。

To test managing role memberships, navigate to the `/RoleAdmin/Index` URL and create a role called `Users`, following the instructions from the “Testing, Creating, and Deleting Roles” section. Click the Edit button and check the boxes so that Alice and Joe are members of the role but Bob is not, as shown in Figure 14-5.

为了测试角色成员的管理，导航到`/RoleAdmin/Index` URL，并按照“测试角色的创建和删除”小节的说明，创建一个名称为`Users`的角色。点击“Edit”按钮，并选中复选框，使Alice和Joe成为该角色的成员，而Bob别选，如图14-5所示。

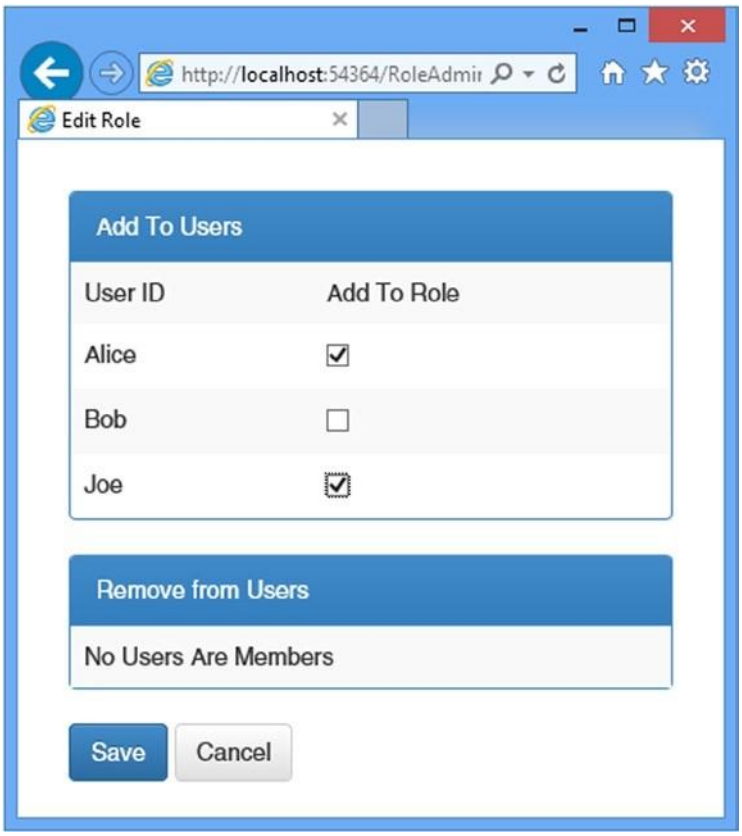

Figure 14-5. *Editing role membership*

图 14-5. *编辑角色成员*

Tip If you get an error that tells you there is already an open a data reader, then you didn't set the `MultipleActiveResultSets` setting to `true` in the connection string in Chapter 13.

提示：如果出现错误，告诉你说，已经有一个打开的数据读取程序，那是因为你并未将第13章连接字符串中的`MultipleActiveResultSets`设置为`true`。

Click the Save button, and the controller will update the role memberships and redirect the browser to the `Index` action. The summary of the `Users` role will show that Alice and Joe are now members, as illustrated by Figure 14-6.

点击“Save”按钮，于是控制器将更新角色成员，并将浏览器重定向到`Index`动作。`Users`角色的摘要将显示Alice和Joe现在已经是成员，如图14-6所示。

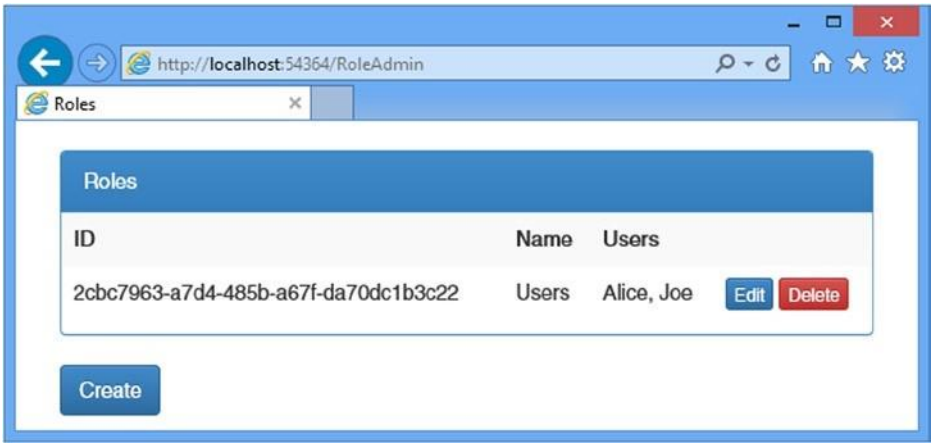

Figure 14-6. *The effect of adding users to a role*

图 14-6. *将用户添加到角色的效果*

14.3.7 Using Roles for Authorization
14.3.7 使用角色进行授权

Now that I have the ability to manage roles, I can use them as the basis for authorization through the `Authorize` attribute. To make it easier to test role-based authorization, I have added a `Logout` method to the `Account` controller, as shown in Listing 14-16, which will make it easier to log out and log in again as a different user to see the effect of role membership.

现在已经能够管理角色了，通过`Authorize`注解属性，还可以将角色作为授权的基础。为了更易于测试基于角色的授权，我在控制器中添加了一个`Logout`方法，如清单14-16所示，这样便很容易注销，也容易作为不同用户登录，以看看角色成员的效果。

Listing 14-16. *Adding a Logout Method to the AccountController.cs File*

清单14-16. *在AccountController.cs文件中添加Logout方法*

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

namespace Users.Controllers {

    [Authorize]
    public class AccountController : Controller {

        [AllowAnonymous]
        public ActionResult Login(string returnUrl) {
```

```
            ViewBag.returnUrl = returnUrl;
            return View();
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> Login(LoginModel details, string returnUrl) {
            // ...statements omitted for brevity...
            // ...出于简化，忽略了一些语句...
        }

        [Authorize]
        public ActionResult Logout() {
            AuthManager.SignOut();
            return RedirectToAction("Index", "Home");
        }

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

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

I have updated the Home controller to add a new action method and pass some information about the authenticated user to the view, as shown in Listing 14-17.
我也更新了Home控制器，添加了一个新的动作方法，并将已认证用户的一些信息传递给视图，如清单14-17所示。

Listing 14-17. *Adding an Action Method and Account Information to the HomeController.cs File*
清单14-17. *在HomeController.cs文件中添加动作方法和账号信息*

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

```
namespace Users.Controllers {

    public class HomeController : Controller {

        [Authorize]
        public ActionResult Index() {
            return View(GetData("Index"));
        }

        [Authorize(Roles="Users")]
        public ActionResult OtherAction() {
            return View("Index", GetData("OtherAction"));
        }

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

I have left the `Authorize` attribute unchanged for the `Index` action method, but I have set the `Roles` property when applying the attribute to the `OtherAction` method, specifying that only members of the `Users` role should be able to access it. I also defined a `GetData` method, which adds some basic information about the user identity, using the properties available through the `HttpContext` object. The final change I made was to the `Index.cshtml` file in the `Views/Home` folder, which is used by both actions in the `Home` controller, to add a link that targets the `Logout` method in the `Account` controller, as shown in Listing 14-18.
我没有改变`Index`动作方法上的`Authorize`注解属性，但将该属性运用于`OtherAction`方法时，已经设置了`Roles`属性，指明只有`Users`角色的成员才能够访问它。我还定义了一个`GetData`方法，它添加了一些有关用户标识的基本信息，这是通过`HttpContext`对象可用的属性获得的。最后所做的修改是`Views/Home`文件夹中的`Index.cshtml`文件，它是由`Home`控制器中的两个动作使用的，我在其中添加了一些以`Account`控制器中的`Logout`方法为目标的链接，如清单14-18所示。

Listing 14-18. *Adding a Sign-Out Link to the Index.cshtml File in the Views/Home Folder*
清单14-18. *在Views/Home文件夹中的Index.cshtml文件中添加Sign-Out（签出）链接*

```
@{ ViewBag.Title = "Index"; }
```

```
<div class="panel panel-primary">
    <div class="panel-heading">User Details</div>
    <table class="table table-striped">
        @foreach (string key in Model.Keys) {
            <tr>
                <th>@key</th>
                <td>@Model[key]</td>
            </tr>
        }
    </table>
</div>

@Html.ActionLink("Sign Out", "Logout", "Account", null, new {@class = "btn btn-primary"})
```

Tip the `Authorize` attribute can also be used to authorize access based on a list of individual usernames. This is an appealing feature for small projects, but it means you have to change the code in your controllers each time the set of users you are authorizing changes, and that usually means having to go through the test-and-deploy cycle again. Using roles for authorization isolates the application from changes in individual user accounts and allows you to control access to the application through the memberships stored by ASP.NET Identity.

提示：`Authorize`注解属性也能够用来根据个别用户名进行授权访问。这是一个对小型项目很吸引人的特性，但这意味着，你每次授权的用户集合发生变化时，必须修改控制器中的代码，这也意味着，要重走一遍从测试到部署的开发周期。使用角色授权将应用程序与修改个别用户账号隔离开来，使你能够通过ASP.NET Identity存储的成员来控制对应用程序的访问。

To test the authentication, start the application and navigate to the `/Home/Index` URL. Your browser will be redirected so that you can enter user credentials. It doesn't matter which of the user details from Table 14-10 you choose to authenticate with because the `Authorize` attribute applied to the `Index` action allows access to any authenticated user.

为了测试认证，启动应用程序，并导航到`/Home/Index` URL。浏览器将被重定向，让你输入用户凭据。选用表14-10中的哪一个用户细节进行认证没有多大关系，因为运用于`Index`动作的`Authorize`注解属性允许任何已认证用户进行访问

However, if you now request the `/Home/OtherAction` URL, the user details you chose from Table 14-10 will make a difference because only `Alice` and `Joe` are members of the `Users` role, which is required to access the `OtherAction` method.

然而，如果你现在请求`/Home/OtherAction` URL，从表14-10所选的用户细节就有区别了，因为只有`Alice`和`Joe`是`Users`角色的成员，这是访问`OtherAction`方法所必须的。

If you log in as Bob, then your browser will be redirected so that you can be prompted for credentials once again.

如果以Bob登录，那么浏览器将被重定向，可能会提示再次输入凭据。

Redirecting an already authenticated user for more credentials is rarely a useful thing to do, so I have modified the `Login` action method in the `Account` controller to check to see whether the user is

authenticated and, if so, redirect them to the shared `Error` view. Listing 14-19 shows the changes.
重定向已认证用户要求更多凭据几乎是一件毫无作用的事，因此，我修改了`Account`控制器中的`Login`动作方法，检查用户是否已认证，如果是，则将他们重定向到共享的`Error`视图，清单14-19显示了所做的修改。

Listing 14-19. *Detecting Already Authenticated Users in the AccountController.cs File*
清单14-19. *在AccountController.cs文件中检测已认证用户*

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

namespace Users.Controllers {

    [Authorize]
    public class AccountController : Controller {

        [AllowAnonymous]
        public ActionResult Login(string returnUrl) {
            if (HttpContext.User.Identity.IsAuthenticated) {
               return View("Error", new string[] { "Access Denied" });
            }
            ViewBag.returnUrl = returnUrl;
            return View();
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> Login(LoginModel details, string returnUrl) {
            // ...code omitted for brevity...
            // ...出于简化，忽略了这里的代码...
        }

        [Authorize]
        public ActionResult Logout() {
            AuthManager.SignOut();
            return RedirectToAction("Index", "Home");
```

```
        }

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

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

Figure 14-7 shows the responses generated for the user Bob when requesting the `/Home/Index` and `/Home/OtherAction` URLs.

图14-7显示了用户Bob在请求`/Home/Index`和`/Home/OtherAction` URL时生成的响应。

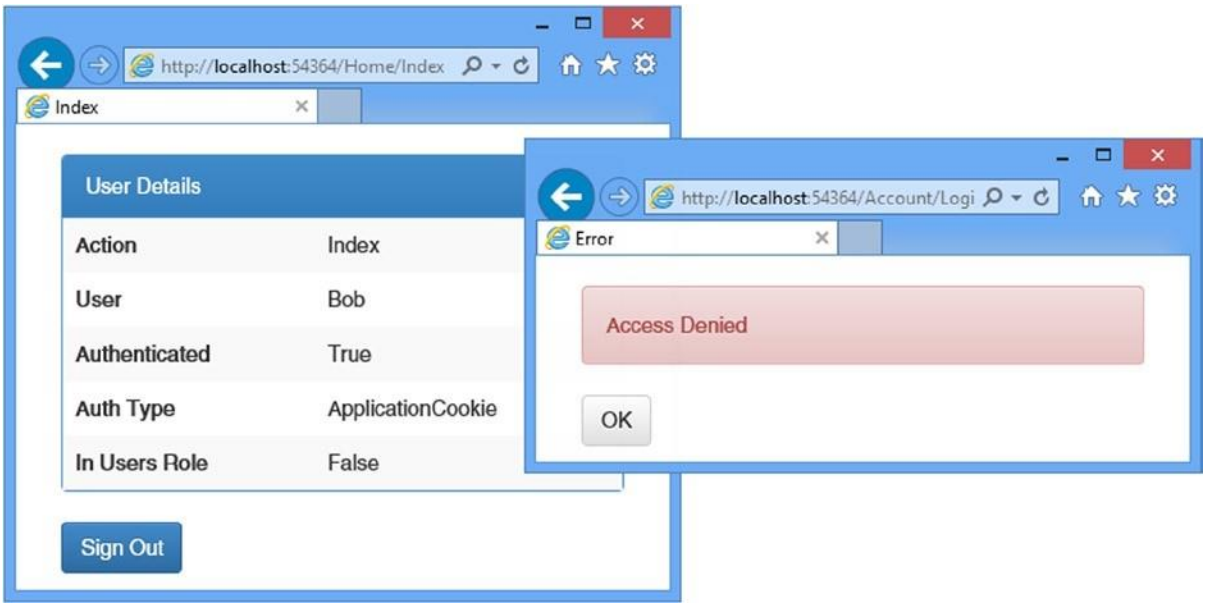

Figure 14-7. *Using roles to control access to action methods*

图 14-7. *使用角色控制对动作方法的访问*

Tip Roles are loaded when the user logs in, which means if you change the roles for the user you are currently authenticated as, the changes won't take effect until you log out and authenticate.

提示：角色在用户登录时就会加载，这意味着，如果修改了当前已认证用户的角色，这些修改是不会生效的，直到他们退出并重新认证。

14.4 Seeding the Database

14.4 种植数据库

One lingering problem in my example project is that access to my `Admin` and `RoleAdmin` controllers is not restricted.
上述示例项目中一直未消除的一个问题是，对`Admin`和`RoleAdmin`控制器的访问是不受限制的。

This is a classic chicken-and-egg problem because in order to restrict access, I need to create users and roles, but the `Admin` and `RoleAdmin` controllers are the user management tools, and if I protect them with the `Authorize` attribute, there won't be any credentials that will grant me access to them, especially when I first deploy the application.
这是一个经典的鸡与蛋的问题，因为，若要限制访问，则需要预先创建一些用户和角色，但`Admin`和`RoleAdmin`控制器又是用户管理工具，如果用`Authorize`注解属性来保护它们，那么就不存在能够对它们访问的凭据，特别是在第一次部署应用程序时。

The solution to this problem is to seed the database with some initial data when the Entity Framework Code First feature creates the schema. This allows me to automatically create users and assign them to roles so that there is a base level of content available in the database.
这一问题的解决方案是，在Entity Framework的Code First特性创建数据库架构时，以一些初始的数据植入数据库。这样能够自动地创建一些用户，并赋予一定的角色，以使数据库中有一个基础级的内容可用。

The database is seeded by adding statements to the `PerformInitialSetup` method of the `IdentityDbInit` class, which is the application-specific Entity Framework database setup class. Listing 14-20 shows the changes I made to create an administration user.
种植数据库的办法是在`IdentityDbInit`类的`PerformInitialSetup`方法中添加一些语句，`IdentityDbInit`是应用程序专用的Entity Framework数据库设置类。清单14-20是为了创建管理用户所做的修改

Listing 14-20. *Seeding the Database in the AppIdentityDbContext.cs File*
清单14-20. *在AppIdentityDbContext.cs文件中种植数据库*

```
using System.Data.Entity;
using Microsoft.AspNet.Identity.EntityFramework;
using Users.Models;
using Microsoft.AspNet.Identity;

namespace Users.Infrastructure {
    public class AppIdentityDbContext : IdentityDbContext<AppUser> {

        public AppIdentityDbContext() : base("IdentityDb") { }

        static AppIdentityDbContext() {
            Database.SetInitializer<AppIdentityDbContext>(new IdentityDbInit());
        }
```

```
        public static AppIdentityDbContext Create() {
            return new AppIdentityDbContext();
        }
    }

    public class IdentityDbInit
            : DropCreateDatabaseIfModelChanges<AppIdentityDbContext> {
        protected override void Seed(AppIdentityDbContext context) {
            PerformInitialSetup(context);
            base.Seed(context);
        }

        public void PerformInitialSetup(AppIdentityDbContext context) {
            AppUserManager userMgr = new AppUserManager(new UserStore<AppUser>(context));
            AppRoleManager roleMgr = new AppRoleManager(new RoleStore<AppRole>(context));

            string roleName = "Administrators";
            string userName = "Admin";
            string password = "MySecret";
            string email = "admin@example.com";

            if (!roleMgr.RoleExists(roleName)) {
                roleMgr.Create(new AppRole(roleName));
            }

            AppUser user = userMgr.FindByName(userName);
            if (user == null) {
                userMgr.Create(new AppUser { UserName = userName, Email = email },
                    password);
                user = userMgr.FindByName(userName);
            }

            if (!userMgr.IsInRole(user.Id, roleName)) {
                userMgr.AddToRole(user.Id, roleName);
            }
        }
    }
}
```

Tip For this example, I used the synchronous extension methods to locate and manage the role and user. As I explained in Chapter 13, I prefer the asynchronous methods by default, but the synchronous methods can be useful when you need to perform a sequence of related operations.

提示：在上述示例中，我使用了同步的扩展方法来定位和管理角色和用户。正如第13章所解释的那样，

一般情况下我更喜欢异步方法，但是，当需要执行一系列相关操作时，同步方法可能是有用的。

I have to create instances of `AppUserManager` and `AppRoleManager` directly because the `PerformInitialSetup` method is called before the OWIN configuration is complete. I use the `RoleManager` and `AppManager` objects to create a role called `Administrators` and a user called `Admin` and add the user to the role.
我必须直接创建`AppUserManager`和`AppRoleManager`的实例，因为`PerformInitialSetup`方法是在OWIN配置完成之前就被调用的。我使用`RoleManager`和`AppManager`对象创建一个名称为`Administrators`的角色，和一个名称为的`Admin`的用户，并将此用户添加到该角色。

■ **Tip** Read Chapter 15 before you add database seeding to your project. I describe database migrations, which allow you to take control of schema changes in the database and which put the seeding logic in a different place.
提示：在项目中添加数据库种植之前，请先阅读第15章。我在其中描述了数据库迁移，这让你能够对数据库中的架构变化进行控制，并可以将种植逻辑放在不同的地方。

With this change, I can use the `Authorize` attribute to protect the `Admin` and `RoleAdmin` controllers. Listing 14-21 shows the change I made to the `Admin` controller.
经过这种修改，我可以使用`Authorize`注解属性来保护`Admin`和`RoleAdmin`控制器。清单14-21显示了对`Admin`控制器所做的修改。

Listing 14-21. *Restricting Access in the AdminController.cs File*
清单14-21. *在AdminController.cs文件中的限制访问*

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

namespace Users.Controllers {

    [Authorize(Roles = "Administrators")]
    public class AdminController : Controller {
        // ...statements omitted for brevity...
        // ...出于简化，忽略了这里的语句...
    }
}
```

Listing 14-22 shows the corresponding change I made to the `RoleAdmin` controller.
清单14-22是对`RoleAdmin`控制器所做的相应修改。

Listing 14-22. *Restricting Access in the RoleAdminController.cs File*

清单14-22. *在RoleAdminController.cs文件中的限制访问*

```
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Users.Infrastructure;
using Users.Models;
using System.Collections.Generic;

namespace Users.Controllers {

    [Authorize(Roles = "Administrators")]
    public class RoleAdminController : Controller {
        // ...statements omitted for brevity...
        // ...出于简化，忽略了这里的语句...
    }
}
```

The database is seeded only when the schema is created, which means I need to reset the database to complete the process. This isn't something you would do in a real application, of course, but I wanted to wait until I demonstrated how authentication and authorization worked before creating the `administrator` account.

只有在创建架构时才会种植数据库，这意味着需要重置数据库才能完成这一过程。当然，这不是在实际项目中可能要做的事情，但我希望等一等，在创建管理员账号之前，完成认证与授权的演示。

To delete the database, open the Visual Studio SQL Server Object Explorer window and locate and right-click the `IdentityDb` item. Select Delete from the pop-up menu and check both of the options in the Delete Database dialog window. Click the OK button to delete the database.

为了删除数据库，请打开Visual Studio中的“SQL Server对象资源管理器”窗口，找到并右击“IdentityDb”条目。从弹出菜单选择“Delete（删除）”，并在“Delete Database（删除数据库）”窗口选中那两个选项。点击“OK”按钮，删除该数据库。

Now create an empty database to which the schema will be added by right-clicking the Databases item, selecting Add New Database, and entering `IdentityDb` in the Database Name field. Click OK to create the empty database.

现在，右击“Databases（数据库）”条目，选择“Add New Database（添加新数据库）”，并在“Database Name（数据库名称）”字段中输入`IdentityDb`。点击OK，创建一个空数据库。

Tip There are step-by-step instructions with screenshots in Chapter 13 for creating the database.

提示：第13章有创建数据库的逐步说明和屏幕截图。

Now start the application and request the `/Admin/Index` or `/RoleAdmin/Index` URL. There will be a delay while the schema is created and the database is seeded, and then you will be prompted to enter your credentials. Use `Admin` as the name and `MySecret` as the password, and you will be granted access to the controllers.

现在，启动应用程序，请求`/Admin/Index`或`/RoleAdmin/Index` URL。在创建数据库架构以及植入数据库期间会有一点延时，然后将提示你输入凭据。使用`Admin`作为用户名，`MySecret`作为口令，将会获得对该控制器的访问。

Caution Deleting the database removes the user accounts you created using the details in table 14-10, which is why you would not perform this task on a live database containing user details.

警告：删除数据库也删去了你用表14-10所创建的用户账号，正是这一原因，一般不会在一个含有用户细节的活动数据库中执行此项任务。

14.5 Summary

14.5 小结

In this chapter, I showed you how to use ASP.NET Identity to authenticate and authorize users. I explained how the ASP.NET life-cycle events provide a foundation for authenticating requests, how to collect and validate credentials users, and how to restrict access to action methods based on the roles that a user is a member of. In the next chapter, I demonstrate some of the advanced features that ASP.NET Identity provides.

在本章中，我演示了如何使用ASP.NET Identity进行用户认证与授权。解释了ASP.NET生命周期事件如何提供认证基础，如何收集和检验用户凭据，以及如何根据用户的成员角色限制对动作方法的访问。下一章将演示ASP.NET Identity所提供的一些高级特性。

CHAPTER 15

15 Advanced ASP.NET Identity
15 ASP.NET Identity高级技术

In this chapter, I finish my description of ASP.NET Identity by showing you some of the advanced features it offers. I demonstrate how you can extend the database schema by defining custom properties on the user class and how to use database migrations to apply those properties without deleting the data in the ASP.NET Identity database. I also explain how ASP.NET Identity supports the concept of claims and demonstrates how they can be used to flexibly authorize access to action methods. I finish the chapter—and the book—by showing you how ASP.NET Identity makes it easy to authenticate users through third parties. I demonstrate authentication with Google accounts, but ASP.NET Identity has built-in support for Microsoft, Facebook, and Twitter accounts as well. Table 15-1 summarizes this chapter.

本章将完成对ASP.NET Identity的描述，向你展示它所提供的一些高级特性。我将演示，你可以扩展ASP.NET Identity的数据库架构，其办法是在用户类上定义一些自定义属性。也会演示如何使用数据库迁移，这样可以运用自定义属性，而不必删除ASP.NET Identity数据库中的数据。还会解释ASP.NET Identity如何支持声明（Claims）概念，并演示如何将它们用于对动作方法进行灵活的授权访问。最后向你展示ASP.NET Identity很容易通过第三方部件来认证用户，以此结束本章以及本书。将要演示的是使用Google账号认证，但ASP.NET Identity对于Microsoft、Facebook以及Twitter账号，都有内建的支持。表15-1是本章概要。

Table 15-1. *Chapter Summary*

表15-1. *本章概要*

Problem **问题**	**Solution** **解决方案**	**Listing** **清单号**
Store additional information about users. 存储附加的用户信息	Define custom user properties. 定义自定义用户属性	1–3, 8–11
Update the database schema without deleting user data. 更新数据库架构而不删除用户数据	Perform a database migration. 执行数据库迁移	4–7
Perform fine-grained authorization. 执行细粒度授权	Use claims. 使用声明（Claims）	12–14
Add claims about a user. 添加用户的声明（Claims）	Use the `ClaimsIdentity.AddClaims` method. 使用`ClaimsIdentity.AddClaims`方法	15–19
Authorize access based on claim	Create a custom authorization filter attribute.	20–21

values. 基于声明（Claims）值授权访问	创建一个自定义的授权过滤器注解属性	
Authenticate through a third party. 通过第三方认证	Install the NuGet package for the authentication provider, redirect requests to that provider, and specify a callback URL that creates the user account. 安装认证提供器的NuGet包，将请求重定向到该提供器，并指定一个创建用户账号的回调URL。	22–25

15.1 Preparing the Example Project

15.1 准备示例项目

In this chapter, I am going to continue working on the `Users` project I created in Chapter 13 and enhanced in Chapter 14. No changes to the application are required, but start the application and make sure that there are users in the database. Figure 15-1 shows the state of my database, which contains the users `Admin`, `Alice`, `Bob`, and `Joe` from the previous chapter. To check the users, start the application and request the `/Admin/Index` URL and authenticate as the `Admin` user.

本章打算继续使用第13章创建并在第14章增强的`Users`项目。对应用程序无需做什么改变，但需要启动应用程序，并确保数据库中有一些用户。图15-1显示了数据库的状态，它含有上一章的用户`Admin`、`Alice`、`Bob`以及`Joe`。为了检查用户，请启动应用程序，请求`/Admin/Index` URL，并以`Admin`用户进行认证。

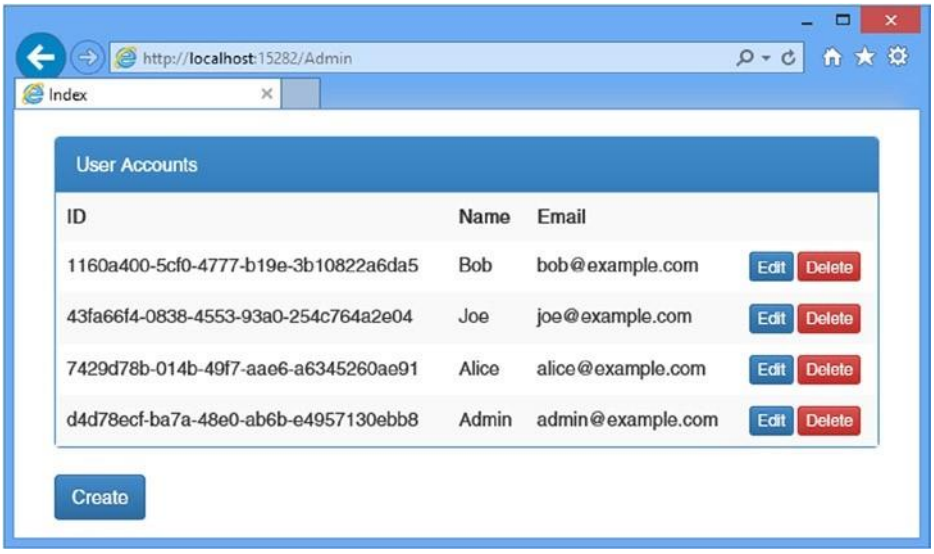

Figure 15-1. *The initial users in the Identity database*

图 15-1. *Identity 数据库中的最初用户*

I also need some roles for this chapter. I used the `RoleAdmin` controller to create roles called `Users` and `Employees` and assigned the users to those roles, as described in Table 15-2.

本章还需要一些角色。我用`RoleAdmin`控制器创建了角色`Users`和`Employees`，并为这些角色指定了一些用户，如表15-2所示。

Table 15-2. *The Types of Web Forms Code Nuggets*

表15-2. 角色及成员（作者将此表的标题写错了——译者注）

Role 角色	Members 成员
Users	Alice, Joe
Employees	Alice, Bob

Figure 15-2 shows the required role configuration displayed by the RoleAdmin controller.
图15-2显示了由RoleAdmin控制器所显示出来的必要的角色配置。

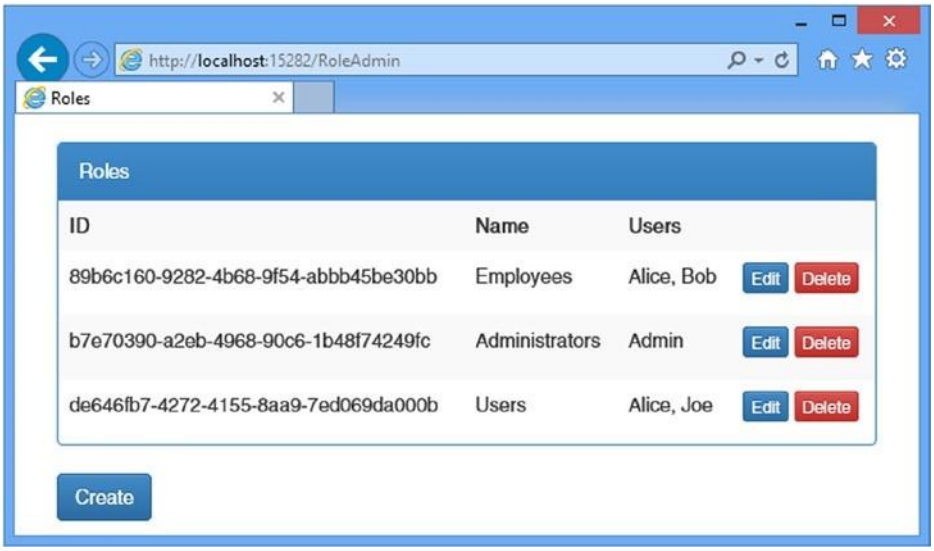

Figure 15-2. *Configuring the roles required for this chapter*
图15-2. *配置本章所需的角色*

15.2 Adding Custom User Properties

15.2 添加自定义用户属性

When I created the AppUser class to represent users in Chapter 13, I noted that the base class defined a basic set of properties to describe the user, such as e-mail address and telephone number. Most applications need to store more information about users, including persistent application preferences and details such as addresses—in short, any data that is useful to running the application and that should last between sessions. In ASP.NET Membership, this was handled through the user profile system, but ASP.NET Identity takes a different approach.
我在第13章创建AppUser类来表示用户时曾做过说明，基类定义了一组描述用户的基本属性，如E-mail地址、电话号码等。大多数应用程序还需要存储用户的更多信息，包括持久化应用程序爱好以及地址等细节——简言之，需要存储对运行应用程序有用并且在各次会话之间应当保持的任何数据。在ASP.NET Membership中，这是通过用户资料（User Profile）系统来处理的，但ASP.NET Identity采取了一种不同的办法。

Because the ASP.NET Identity system uses Entity Framework to store its data by default, defining additional user information is just a matter of adding properties to the user class and letting the Code First

feature create the database schema required to store them. Table 15-3 puts custom user properties in context.
因为ASP.NET Identity默认是使用Entity Framework来存储其数据的，定义附加的用户信息只不过是给用户类添加属性的事情，然后让Code First特性去创建需要存储它们的数据库架构即可。表15-3描述了自定义用户属性的情形。

Table 15-3. *Putting Cusotm User Properties in Context*
表15-3. *自定义用户属性的情形*

Question **问题**	**Answer** **回答**
What is it? 什么是自定义用户属性？	Custom user properties allow you to store additional information about your users, including their preferences and settings. 自定义用户属性让你能够存储附加的用户信息，包括他们的爱好和设置。
Why should I care? 为何要关心它？	A persistent store of settings means that the user doesn't have to provide the same information each time they log in to the application. 设置的持久化存储意味着，用户不必每次登录到应用程序时都提供同样的信息。
How is it used by the MVC framework? 在MVC框架中如何使用它？	This feature isn't used directly by the MVC framework, but it is available for use in action methods. 此特性不是由MVC框架直接使用的，但它在动作方法中使用是有效的。

15.2.1 Defining Custom Properties
15.2.1 定义自定义属性

Listing 15-1 shows how I added a simple property to the `AppUser` class to represent the city in which the user lives.
清单15-1演示了如何给`AppUser`类添加一个简单的属性，用以表示用户生活的城市。

Listing 15-1. *Adding a Property in the AppUser.cs File*
清单15-1. *在AppUser.cs文件中添加属性*

```
using System;
using Microsoft.AspNet.Identity.EntityFramework;

namespace Users.Models {
    public enum Cities {
        LONDON, PARIS, CHICAGO
    }

    public class AppUser : IdentityUser {
        public Cities City { get; set; }
    }
}
```

I have defined an enumeration called `Cities` that defines values for some large cities and added a property called `City` to the `AppUser` class. To allow the user to view and edit their `City` property, I added actions to the `Home` controller, as shown in Listing 15-2.
这里定义了一个枚举，名称为`Cities`，它定义了一些大城市的值，另外给`AppUser`类添加了一个名称为`City`的属性。为了让用户能够查看和编辑`City`属性，给`Home`控制器添加了几个动作方法，如清单15-2所示。

Listing 15-2. *Adding Support for Custom User Properties in the HomeController.cs File*
清单15-2. *在HomeController.cs文件中添加对自定义属性的支持*

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

namespace Users.Controllers {

    public class HomeController : Controller {

        [Authorize]
        public ActionResult Index() {
            return View(GetData("Index"));
        }

        [Authorize(Roles = "Users")]
        public ActionResult OtherAction() {
            return View("Index", GetData("OtherAction"));
        }

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

```
        [Authorize]
        public ActionResult UserProps() {
             return View(CurrentUser);
        }

        [Authorize]
        [HttpPost]
        public async Task<ActionResult> UserProps(Cities city) {
            AppUser user = CurrentUser;
            user.City = city;
            await UserManager.UpdateAsync(user);
            return View(user);
        }

        private AppUser CurrentUser {
            get {
                return UserManager.FindByName(HttpContext.User.Identity.Name);
            }
        }

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

I added a `CurrentUser` property that uses the `AppUserManager` class to retrieve an `AppUser` instance to represent the current user. I pass the `AppUser` object as the view model object in the `GET` version of the `UserProps` action method, and the `POST` method uses it to update the value of the new `City` property. Listing 15-3 shows the `UserProps.cshtml` view, which displays the `City` property value and contains a form to change it.

我添加了一个`CurrentUser`属性，它使用`AppUserManager`类接收了表示当前用户的`AppUser`实例。在`GET`版本的`UserProps`动作方法中，传递了这个`AppUser`对象作为视图模型。而在`POST`版的方法中用它更新了`City`属性的值。清单15-3显示了`UserProps.cshtml`视图，它显示了`City`属性的值，并包含一个修改它的表单。

Listing 15-3. *The Contents of the UserProps.cshtml File in the Views/Home Folder*

清单15-3. *Views/Home文件夹中UserProps.cshtml文件的内容*

```
@using Users.Models
@model AppUser
```

```
@{ ViewBag.Title = "UserProps";}

<div class="panel panel-primary">
    <div class="panel-heading">
        Custom User Properties
    </div>
    <table class="table table-striped">
        <tr><th>City</th><td>@Model.City</td></tr>
    </table>
</div>

@using (Html.BeginForm()) {
    <div class="form-group">
        <label>City</label>
        @Html.DropDownListFor(x => x.City, new SelectList(Enum.GetNames(typeof(Cities))))
    </div>
    <button class="btn btn-primary" type="submit">Save</button>
}
```

■ **Caution** Don't start the application when you have created the view. In the sections that follow, I demonstrate how to preserve the contents of the database, and if you start the application now, the ASP.NET Identity users will be deleted.

警告：创建了视图之后不要启动应用程序。在以下小节中，将演示如何保留数据库的内容，如果现在启动应用程序，将会删除ASP.NET Identity的用户。

15.2.2 Preparing for Database Migration
15.2.2 准备数据库迁移

The default behavior for the Entity Framework Code First feature is to drop the tables in the database and re-create them whenever classes that drive the schema have changed. You saw this in Chapter 14 when I added support for roles: When the application was started, the database was reset, and the user accounts were lost.

Entity Framework Code First特性的默认行为是，一旦修改了派生数据库架构的类，便会删除数据库中的数据表，并重新创建它们。在第14章可以看到这种情况，在我添加角色支持时：当重启应用程序后，数据库被重置，用户账号也丢失。

Don't start the application yet, but if you were to do so, you would see a similar effect. Deleting data during development is usually not a problem, but doing so in a production setting is usually disastrous because it deletes all of the real user accounts and causes a panic while the backups are restored. In this section, I am going to demonstrate how to use the database migration feature, which updates a Code First schema in a less brutal manner and preserves the existing data it contains.

不要启动应用程序，但如果你这么做了，会看到类似的效果。在开发期间删除数据没什么问题，但如果在产品设置中这么做了，通常是灾难性的，因为它会删除所有真实的用户账号，而备份恢复是很痛苦的事。在本小节中，我打算演示如何使用数据库迁移特性，它能以比较温和的方式更新Code First的架构，并保留架构中的已有数据。

The first step is to issue the following command in the Visual Studio Package Manager Console:
第一个步骤是在Visual Studio的“Package Manager Console（包管理器控制台）”中发布以下命令：

```
Enable-Migrations –EnableAutomaticMigrations
```

This enables the database migration support and creates a `Migrations` folder in the Solution Explorer that contains a `Configuration.cs` class file, the contents of which are shown in Listing 15-4.
它启用了数据库的迁移支持，并在“Solution Explorer（解决方案资源管理器）”创建一个`Migrations`文件夹，其中含有一个`Configuration.cs`类文件，内容如清单15-4所示。

Listing 15-4. *The Contents of the Configuration.cs File*
清单15-4. *Configuration.cs文件的内容*

```
namespace Users.Migrations {
using System;
using System.Data.Entity;
using System.Data.Entity.Migrations;
using System.Linq;

    internal sealed class Configuration
            : DbMigrationsConfiguration<Users.Infrastructure.AppIdentityDbContext> {
        public Configuration() {
            AutomaticMigrationsEnabled = true;
            ContextKey = "Users.Infrastructure.AppIdentityDbContext";
        }

        protected override void Seed(Users.Infrastructure.AppIdentityDbContext context) {
            // This method will be called after migrating to the latest version.
            // 此方法将在迁移到最新版本时调用

            // You can use the DbSet<T>.AddOrUpdate() helper extension method
            // to avoid creating duplicate seed data. E.g.
            // 例如，你可以使用DbSet<T>.AddOrUpdate()辅助器方法来避免创建重复的种子数据
            //
            //    context.People.AddOrUpdate(
            //      p => p.FullName,
            //      new Person { FullName = "Andrew Peters" },
            //      new Person { FullName = "Brice Lambson" },
            //      new Person { FullName = "Rowan Miller" }
            //    );
            //
        }
    }
}
```

Tip You might be wondering why you are entering a database migration command into the console used to manage NuGet packages. The answer is that the Package Manager Console is really *PowerShell*, which is a general-purpose tool that is mislabeled by Visual Studio. You can use the console to issue a wide range of helpful commands. See http://go.microsoft.com/fwlink/?LinkID=108518 for details.

提示：你可能会觉得奇怪，为什么要在管理NuGet包的控制台中输入数据库迁移的命令？答案是“Package Manager Console（包管理控制台）”是真正的*PowerShell*，这是Visual studio冒用的一个通用工具。你可以使用此控制台发送大量的有用命令，详见http://go.microsoft.com/fwlink/?LinkID=108518。

The class will be used to migrate existing content in the database to the new schema, and the `Seed` method will be called to provide an opportunity to update the existing database records. In Listing 15-5, you can see how I have used the `Seed` method to set a default value for the new `City` property I added to the `AppUser` class. (I have also updated the class file to reflect my usual coding style.)

这个类将用于把数据库中的现有内容迁移到新的数据库架构，`Seed`方法的调用为更新现有数据库记录提供了机会。在清单15-5中可以看到，我如何用`Seed`方法为新的`City`属性设置默认值，`City`是添加到`AppUser`类中自定义属性。（为了体现我一贯的编码风格，我对这个类文件也进行了更新。）

Listing 15-5. *Managing Existing Content in the Configuration.cs File*

清单15-5. *在Configuration.cs文件中管理已有内容*

```
using System.Data.Entity.Migrations;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Users.Infrastructure;
using Users.Models;

namespace Users.Migrations {

    internal sealed class Configuration
            : DbMigrationsConfiguration<AppIdentityDbContext> {

        public Configuration() {
            AutomaticMigrationsEnabled = true;
            ContextKey = "Users.Infrastructure.AppIdentityDbContext";
        }

        protected override void Seed(AppIdentityDbContext context) {

            AppUserManager userMgr = new AppUserManager(new UserStore<AppUser>(context));
            AppRoleManager roleMgr = new AppRoleManager(new RoleStore<AppRole>(context));

            string roleName = "Administrators";
            string userName = "Admin";
            string password = "MySecret";
            string email = "admin@example.com";
```

```
            if (!roleMgr.RoleExists(roleName)) {
                roleMgr.Create(new AppRole(roleName));
            }

            AppUser user = userMgr.FindByName(userName);
            if (user == null) {
                userMgr.Create(new AppUser { UserName = userName, Email = email },
                    password);
                user = userMgr.FindByName(userName);
            }

            if (!userMgr.IsInRole(user.Id, roleName)) {
                userMgr.AddToRole(user.Id, roleName);
            }

            foreach (AppUser dbUser in userMgr.Users) {
                dbUser.City = Cities.PARIS;
            }
            context.SaveChanges();
        }
    }
}
```

You will notice that much of the code that I added to the `Seed` method is taken from the `IdentityDbInit` class, which I used to seed the database with an administration user in Chapter 14. This is because the new `Configuration` class added to support database migrations will replace the seeding function of the `IdentityDbInit` class, which I'll update shortly. Aside from ensuring that there is an `admin` user, the statements in the `Seed` method that are important are the ones that set the initial value for the `City` property I added to the `AppUser` class, as follows:
你可能会注意到，添加到`Seed`方法中的许多代码取自于`IdentityDbInit`类，在第14章中我用这个类将管理用户植入了数据库。这是因为这个新添加的、用以支持数据库迁移的`Configuration`类，将代替`IdentityDbInit`类的种植功能，我很快便会更新这个类。除了要确保有`admin`用户之外，在`Seed`方法中的重要语句是那些为`AppUser`类的`City`属性设置初值的语句，如下所示：

```
...
foreach (AppUser dbUser in userMgr.Users) {
    dbUser.City = Cities.PARIS;
}
context.SaveChanges();
...
```

You don't have to set a default value for new properties—I just wanted to demonstrate that the `Seed` method in the `Configuration` class can be used to update the existing user records in the database.
你不一定要为新属性设置默认值——这里只是想演示`Configuration`类中的`Seed`方法，可以用它更新

数据库中的已有用户记录。

Caution Be careful when setting values for properties in the `Seed` method for real projects because the values will be applied every time you change the schema, overriding any values that the user has set since the last schema update was performed. I set the value of the `City` property just to demonstrate that it can be done.

警告：在用于真实项目的`Seed`方法中为属性设置值时要小心，因为你每一次修改架构时，都会运用这些值，这会将自执行上一次架构更新之后，用户设置的任何数据覆盖掉。这里设置`City`属性的值只是为了演示它能够这么做。

Changing the Database Context Class
修改数据库上下文类

The reason that I added the seeding code to the `Configuration` class is that I need to change the `IdentityDbInit` class. At present, the `IdentityDbInit` class is derived from the descriptively named `DropCreateDatabaseIfModelChanges<AppIdentityDbContext>` class, which, as you might imagine, drops the entire database when the Code First classes change. Listing 15-6 shows the changes I made to the `IdentityDbInit` class to prevent it from affecting the database.

在`Configuration`类中添加种植代码的原因是我需要修改`IdentityDbInit`类。此时，`IdentityDbInit`类派生于描述性命名的`DropCreateDatabaseIfModelChanges<AppIdentityDbContext>`类，和你相像的一样，它会在Code First类改变时删除整个数据库。清单15-6显示了我对`IdentityDbInit`类所做的修改，以防止它影响数据库。

Listing 15-6. *Preventing Database Schema Changes in the AppIdentityDbContext.cs File*

清单15-6. *在AppIdentityDbContext.cs文件是阻止数据库架构变化*

```
using System.Data.Entity;
using Microsoft.AspNet.Identity.EntityFramework;
using Users.Models;
using Microsoft.AspNet.Identity;

namespace Users.Infrastructure {
    public class AppIdentityDbContext : IdentityDbContext<AppUser> {

        public AppIdentityDbContext() : base("IdentityDb") { }

        static AppIdentityDbContext() {
            Database.SetInitializer<AppIdentityDbContext>(new IdentityDbInit());
        }

        public static AppIdentityDbContext Create() {
             return new AppIdentityDbContext();
        }
    }
     public class IdentityDbInit : NullDatabaseInitializer<AppIdentityDbContext> {
```

```
    }
}
```

I have removed the methods defined by the class and changed its base to `NullDatabaseInitializer<AppIdentityDbContext>`, which prevents the schema from being altered.
我删除了这个类中所定义的方法，并将它的基类改为`NullDatabaseInitializer<AppIdentityDbContext>`，它可以防止架构修改。

15.2.3 Performing the Migration
15.2.3 执行迁移

All that remains is to generate and apply the migration. First, run the following command in the Package Manager Console:
剩下的事情只是生成并运用迁移了。首先，在“Package Manager Console（包管理器控制台）”中执行以下命令：

```
Add-Migration CityProperty
```

This creates a new migration called `CityProperty` (I like my migration names to reflect the changes I made). A class new file will be added to the `Migrations` folder, and its name reflects the time at which the command was run and the name of the migration. My file is called `201402262244036_CityProperty.cs`, for example. The contents of this file contain the details of how Entity Framework will change the database during the migration, as shown in Listing 15-7.
这创建了一个名称为`CityProperty`的新迁移（我比较喜欢让迁移的名称反映出我所做的修改）。这会在文件夹中添加一个新的类文件，而且其命名会反映出该命令执行的时间以及迁移名称，例如，我的这个文件名称为`201402262244036_CityProperty.cs`。该文件的内容含有迁移期间Entity Framework修改数据库的细节，如清单15-7所示。

Listing 15-7. *The Contents of the 201402262244036_CityProperty.cs File*
清单15-7. *201402262244036_CityProperty.cs文件的内容*

```
namespace Users.Migrations {
    using System;
    using System.Data.Entity.Migrations;

    public partial class Init : DbMigration {
        public override void Up() {
            AddColumn("dbo.AspNetUsers", "City", c => c.Int(nullable: false));
        }

        public override void Down() {
            DropColumn("dbo.AspNetUsers", "City");
        }
    }
}
```

The Up method describes the changes that have to be made to the schema when the database is upgraded, which in this case means adding a City column to the AspNetUsers table, which is the one that is used to store user records in the ASP.NET Identity database.
Up方法描述了在数据库升级时，需要对架构所做的修改，在这个例子中，意味着要在AspNetUsers数据表中添加City数据列，该数据表是ASP.NET Identity数据库用来存储用户记录的。

The final step is to perform the migration. Without starting the application, run the following command in the Package Manager Console:
最后一步是执行迁移。无需启动应用程序，只需在“Package Manager Console（包管理器控制台）”中运行以下命令即可：

```
Update-Database –TargetMigration CityProperty
```

The database schema will be modified, and the code in the Configuration.Seed method will be executed. The existing user accounts will have been preserved and enhanced with a City property (which I set to Paris in the Seed method).
这会修改数据库架构，并执行Configuration.Seed方法中的代码。已有用户账号会被保留，且增强了City属性（我在Seed方法中已将其设置为“Paris”）。

15.2.4 Testing the Migration
15.2.4 测试迁移

To test the effect of the migration, start the application, navigate to the /Home/UserProps URL, and authenticate as one of the Identity users (for example, as Alice with the password MySecret). Once authenticated, you will see the current value of the City property for the user and have the opportunity to change it, as shown in Figure 15-3.
为了测试迁移的效果，启动应用程序，导航到/Home/UserProps URL，并以Identity中的用户（例如Alice，口令MySecret）进行认证。一旦已被认证，便会看到该用户City属性的当前值，并可以对其进行修改，如图15-3所示。

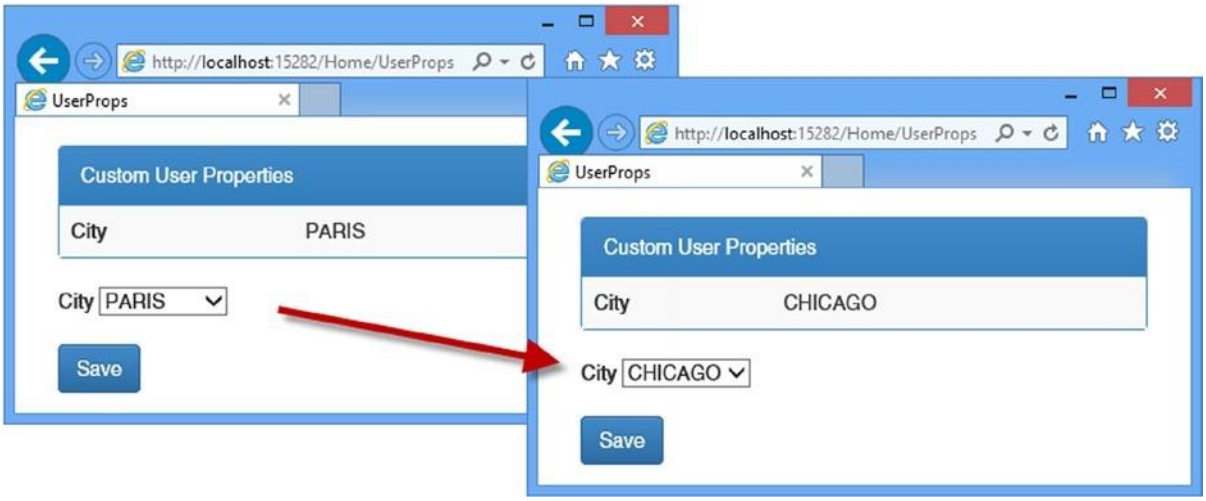

Figure 15-3. *Displaying and changing a custom user property*

图15-3. *显示和个性自定义用户属性*

15.2.5 Defining an Additional Property
15.2.5 定义附加属性

Now that database migrations are set up, I am going to define a further property just to demonstrate how subsequent changes are handled and to show a more useful (and less dangerous) example of using the `Configuration.Seed` method. Listing 15-8 shows how I added a `Country` property to the `AppUser` class.
现在，已经建立了数据库迁移，我打算再定义一个属性，这恰恰演示了如何处理持续不断的修改，也为了演示`Configuration.Seed`方法更有用（至少无害）的示例。清单15-8显示了我在`AppUser`类上添加了一个`Country`属性。

Listing 15-8. *Adding Another Property in the AppUserModels.cs File*

清单15-8. *在AppUserModels.cs文件中添加另一个属性*

```
using System;
using Microsoft.AspNet.Identity.EntityFramework;

namespace Users.Models {

    public enum Cities {
        LONDON, PARIS, CHICAGO
    }

     public enum Countries {
         NONE, UK, FRANCE, USA
    }

    public class AppUser : IdentityUser {
         public Cities City { get; set; }
         public Countries Country { get; set; }
```

```
        public void SetCountryFromCity(Cities city) {
            switch (city) {
                case Cities.LONDON:
                    Country = Countries.UK;
                    break;
                case Cities.PARIS:
                    Country = Countries.FRANCE;
                    break;
                case Cities.CHICAGO:
                    Country = Countries.USA;
                    break;
                default:
                    Country = Countries.NONE;
                    break;
            }
        }
    }
}
```

I have added an enumeration to define the country names and a helper method that selects a country value based on the `City` property. Listing 15-9 shows the change I made to the `Configuration` class so that the `Seed` method sets the `Country` property based on the `City`, but only if the value of Country is NONE (which it will be for all users when the database is migrated because the Entity Framework sets enumeration columns to the first value).
我已经添加了一个枚举，它定义了国家名称。还添加了一个辅助器方法，它可以根据`City`属性选择一个国家。清单15-9显示了对`Configuration`类所做的修改，以使`Seed`方法根据`City`设置`Country`属性，但只当`Country`为`NONE`时才进行设置（在迁移数据库时，所有用户都是NONE，因为Entity Framework会将枚举列设置为枚举的第一个值）。

Listing 15-9. *Modifying the Database Seed in the Configuration.cs File*
清单15-9. *在Configuration.cs文件中修改数据库种子*

```
using System.Data.Entity.Migrations;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Users.Infrastructure;
using Users.Models;

namespace Users.Migrations {

    internal sealed class Configuration
            : DbMigrationsConfiguration<AppIdentityDbContext> {

        public Configuration() {
```

```
            AutomaticMigrationsEnabled = true;
            ContextKey = "Users.Infrastructure.AppIdentityDbContext";
        }

        protected override void Seed(AppIdentityDbContext context) {

            AppUserManager userMgr = new AppUserManager(new UserStore<AppUser>(context));
            AppRoleManager roleMgr = new AppRoleManager(new RoleStore<AppRole>(context));

            string roleName = "Administrators";
            string userName = "Admin";
            string password = "MySecret";
            string email = "admin@example.com";

            if (!roleMgr.RoleExists(roleName)) {
                 roleMgr.Create(new AppRole(roleName));
            }

            AppUser user = userMgr.FindByName(userName);
            if (user == null) {
                userMgr.Create(new AppUser { UserName = userName, Email = email },
                    password);
                user = userMgr.FindByName(userName);
            }

            if (!userMgr.IsInRole(user.Id, roleName)) {
                userMgr.AddToRole(user.Id, roleName);
            }

            foreach (AppUser dbUser in userMgr.Users) {
                if (dbUser.Country == Countries.NONE) {
                    dbUser.SetCountryFromCity(dbUser.City);
                }
            }

            context.SaveChanges();
        }
    }
}
```

This kind of seeding is more useful in a real project because it will set a value for the `Country` property only if one has not already been set—subsequent migrations won't be affected, and user selections won't be lost.

这种种植在实际项目中会更有用，因为它只会在`Country`属性未设置时，才会设置`Country`属性的值—

—后继的迁移不会受到影响，因此不会失去用户的选择。

1. Adding Application Support
1. 添加应用程序支持

There is no point defining additional user properties if they are not available in the application, so Listing 15-10 shows the change I made to the `Views/Home/UserProps.cshtml` file to display the value of the `Country` property.
应用程序中如果没有定义附加属性的地方，则附加属性就无法使用了，因此，清单15-10显示了我对`Views/Home/UserProps.cshtml`文件的修改，以显示`Country`属性的值。

Listing 15-10. *Displaying an Additional Property in the UserProps.cshtml File*
清单15-10. *在UserProps.cshtml文件中显示附加属性*

```
@using Users.Models
@model AppUser
@{ ViewBag.Title = "UserProps";}
<div class="panel panel-primary">
    <div class="panel-heading">
        Custom User Properties
    </div>
    <table class="table table-striped">
        <tr><th>City</th><td>@Model.City</td></tr>
        <tr><th>Country</th><td>@Model.Country</td></tr>
    </table>
</div>
@using (Html.BeginForm()) {
    <div class="form-group">
        <label>City</label>
        @Html.DropDownListFor(x => x.City, new SelectList(Enum.GetNames(typeof(Cities))))
    </div>
    <button class="btn btn-primary" type="submit">Save</button>
}
```

Listing 15-11 shows the corresponding change I made to the `Home` controller to update the `Country` property when the `City` value changes.
为了在`City`值变化时能够更新`Country`属性，清单15-11显示了我对`Home`控制器所做的相应修改。

Listing 15-11. *Setting Custom Properties in the HomeController.cs File*
清单15-11. *在HomeController.cs文件中设置自定义属性*

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

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

namespace Users.Controllers {

    public class HomeController : Controller {
          // ...other action methods omitted for brevity...
          // ...出于简化，这里忽略了其他动作方法...

        [Authorize]
        public ActionResult UserProps() {
            return View(CurrentUser);
        }

        [Authorize]
        [HttpPost]
        public async Task<ActionResult> UserProps(Cities city) {
            AppUser user = CurrentUser;
            user.City = city;
            user.SetCountryFromCity(city);
            await UserManager.UpdateAsync(user);
            return View(user);
        }

        // ...properties omitted for brevity...
        // ...出于简化，这里忽略了一些属性...
    }
}
```

2. Performing the Migration
2. 准备迁移

All that remains is to create and apply a new migration. Enter the following command into the Package Manager Console:
剩下的事情就是创建和运用新的迁移了。在“Package Manager Console（包管理器控制台）”中输入以下命令：

```
Add-Migration CountryProperty
```

This will generate another file in the `Migrations` folder that contains the instruction to add the `Country` column. To apply the migration, execute the following command:
这将在`Migrations`文件夹中生成另一个文件，它含有添加`Country`数据表列的指令。为了运用迁移，

可执行以下命令：

```
Update-Database –TargetMigration CountryProperty
```

The migration will be performed, and the value of the `Country` property will be set based on the value of the existing `City` property for each user. You can check the new user property by starting the application and authenticating and navigating to the `/Home/UserProps` URL, as shown in Figure 15-4.
这将执行迁移，`Country`属性的值将根据每个用户当前的`City`属性进行设置。通过启动应用程序，认证并导航到`/Home/UserProps` URL，便可以查看新的用户属性，如图15-4所示。

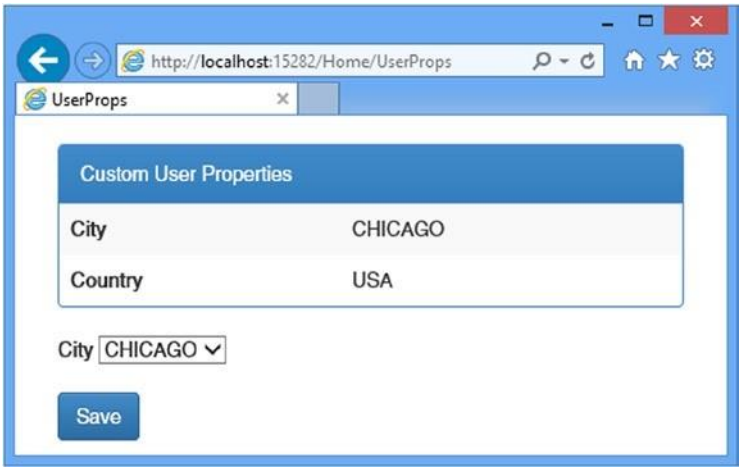

Figure 15-4. *Creating an additional user property*
图 15-4. *创建附加用户属性*

Tip Although I am focused on the process of upgrading the database, you can also migrate back to a previous version by specifying an earlier migration. Use the –Force argument make changes that cause data loss, such as removing a column.
提示：虽然我们关注了升级数据库的过程，但你也可以回退到以前的版本，只需指定一个早期的迁移即可。使用`-Force`参数进行修改，会引起数据丢失，例如删除数据表列。

15.3 Working with Claims

15.3 使用声明（Claims）

In older user-management systems, such as ASP.NET Membership, the application was assumed to be the authoritative source of all information about the user, essentially treating the application as a closed world and trusting the data that is contained within it.
在旧的用户管理系统中，例如ASP.NET Membership，应用程序被假设成是用户所有信息的权威来源，本质上将应用程序视为是一个封闭的世界，并且只信任其中所包含的数据。

This is such an ingrained approach to software development that it can be hard to recognize that's what is happening, but you saw an example of the closed-world technique in Chapter 14 when I authenticated users against the credentials stored in the database and granted access based on the roles associated with

those credentials. I did the same thing again in this chapter when I added properties to the user class. Every piece of information that I needed to manage user authentication and authorization came from within my application—and that is a perfectly satisfactory approach for many web applications, which is why I demonstrated these techniques in such depth.
这是软件开发的一种根深蒂固的方法，使人很难认识到这到底意味着什么，第14章你已看到了这种封闭世界技术的例子，根据存储在数据库中的凭据来认证用户，并根据与凭据关联在一起的角色来授权访问。本章前述在用户类上添加属性，也做了同样的事情。我管理用户认证与授权所需的每一个数据片段都来自于我的应用程序——而且这是许多Web应用程序都相当满意的一种方法，这也是我如此深入地演示这些技术的原因。

ASP.NET Identity also supports an alternative approach for dealing with users, which works well when the MVC framework application isn't the sole source of information about users and which can be used to authorize users in more flexible and fluid ways than traditional roles allow.
ASP.NET Identity还支持另一种处理用户的办法，当MVC框架的应用程序不是有关用户的唯一信息源时，这种办法会工作得很好，而且能够比传统的角色授权更为灵活且流畅的方式进行授权。

This alternative approach uses *claims*, and in this section I'll describe how ASP.NET Identity supports *claims-based authorization*. Table 15-4 puts claims in context.
这种可选的办法使用了“*Claims（声明）*”，因此在本小节中，我将描述ASP.NET Identity如何支持“*Claims-Based Authorization（基于声明的授权）*”。表15-4描述了声明（Claims）的情形。

提示：“Claim”在英文字典中不完全是“声明”的含义，根据本文的描述，感觉把它说成“声明”也不一定合适，所以在之后的译文中基本都写成中英文并用的形式，即“声明（Claims）”。根据表15-4中的定义：**声明（Claims）是关于用户的一些信息片段**。一个用户的信息片段当然有很多，每一个信息片段就是一项声明（Claim），用户的所有信息片段合起来就是该用户的声明（Claims）。请读者注意该单词的单复数形式——译者注

Table 15-4. *Putting Claims in Context*
表15-4. *声明（Claims）的情形*

Question **问题**	**Answer** **答案**
What is it? 什么是声明（Claims）？	Claims are pieces of information about users that you can use to make authorization decisions. Claims can be obtained from external systems as well as from the local Identity database. 声明（Claims）是关于用户的一些信息片段，可以用它们做出授权决定。声明（Claims）可以从外部系统获取，也可以从本地的Identity数据库获取。
Why should I care? 为何要关心它？	Claims can be used to flexibly authorize access to action methods. Unlike conventional roles, claims allow access to be driven by the information that describes the user. 声明（Claims）可以用来对动作方法进行灵活的授权访问。与传统的角色不同，声明（Claims）让访问能够由描述用户的信息进行驱动。
How is it used by the MVC framework? 如何在MVC框架中使用它？	This feature isn't used directly by the MVC framework, but it is integrated into the standard authorization features, such as the `Authorize` attribute. 这不是直接由MVC框架使用的特性，但它集成到了标准的授权特性之中，例如`Authorize`注解属性。

Tip you don't have to use claims in your applications, and as Chapter 14 showed, ASP.NET Identity is

perfectly happy providing an application with the authentication and authorization services without any need to understand claims at all.
提示：你在应用程序中不一定要使用声明（Claims），正如第14章所展示的那样，ASP.NET Identity能够为应用程序提供充分的认证与授权服务，而根本不需要理解声明（Claims）。

15.3.1 Understanding Claims
15.3.1 理解声明（Claims）

A *claim* is a piece of information about the user, along with some information about where the information came from. The easiest way to unpack claims is through some practical demonstrations, without which any discussion becomes too abstract to be truly useful. To get started, I added a `Claims` controller to the example project, the definition of which you can see in Listing 15-12.
一项*声明*（Claim）是关于用户的一个信息片段（请注意这个英文单词的单复数形式——译者注），并伴有该片段出自何处的某种信息。揭开声明（Claims）含义最容易的方式是做一些实际演示，任何讨论都会过于抽象根本没有真正的用处。为此，我在示例项目中添加了一个`Claims`控制器，其定义如清单15-12所示。

Listing 15-12. *The Contents of the ClaimsController.cs File*
清单15-12. *ClaimsController.cs文件的内容*

```
using System.Security.Claims;
using System.Web;
using System.Web.Mvc;

namespace Users.Controllers {
    public class ClaimsController : Controller {

        [Authorize]
        public ActionResult Index() {
            ClaimsIdentity ident = HttpContext.User.Identity as ClaimsIdentity;
            if (ident == null) {
                return View("Error", new string[] { "No claims available" });
            } else {
                return View(ident.Claims);
            }
        }
    }
}
```

Tip You may feel a little lost as I define the code for this example. Don't worry about the details for the moment—just stick with it until you see the output from the action method and view that I define. More than anything else, that will help put claims into perspective.
提示：你或许会对我为此例定义的代码感到有点失望。此刻对此细节不必着急——只要稍事忍耐，当看到该动作方法和视图的输出便会明白。尤为重要的是，这有助于洞察声明（Claims）。

You can get the claims associated with a user in different ways. One approach is to use the Claims property defined by the user class, but in this example, I have used the HttpContext.User.Identity property to demonstrate the way that ASP.NET Identity is integrated with the rest of the ASP.NET platform. As I explained in Chapter 13, the HttpContext.User.Identity property returns an implementation of the IIdentity interface, which is a ClaimsIdentity object when working using ASP.NET Identity. The ClaimsIdentity class is defined in the System.Security.Claims namespace, and Table 15-5 shows the members it defines that are relevant to this chapter.

可以通过不同的方式获得与用户相关联的声明（Claims）。方法之一就是使用由用户类定义的Claims属性，但在这个例子中，我使用了HttpContext.User.Identity属性，目的是演示ASP.NET Identity与ASP.NET平台集成的方式（请注意这句话所表示的含义：用户类的Claims属性属于ASP.NET Identity，而HttpContext.User.Identity属性则属于ASP.NET平台。由此可见，ASP.NET Identity已经融合到了ASP.NET平台之中——译者注）。正如第13章所解释的那样，HttpContext.User.Identity属性返回IIdentity的接口实现，当使用ASP.NET Identity时，该实现是一个ClaimsIdentity对象。ClaimsIdentity类是在System.Security.Claims命名空间中定义的，表15-5显示了它所定义的与本章有关的成员。

Table 15-5. *The Members Defined by the ClaimsIdentity Class*

表15-5. *ClaimsIdentity类所定义的成员*

Name 名称	Description 描述
Claims	Returns an enumeration of Claim objects representing the claims for the user. 返回表示用户声明（Claims）的Claim对象枚举，
AddClaim(claim)	Adds a claim to the user identity. 给用户添加一个声明（Claim）
AddClaims(claims)	Adds an enumeration of Claim objects to the user identity. 将一个Claim对象枚举添加到用户标识。
HasClaim(predicate)	Returns true if the user identity contains a claim that matches the specified predicate. See the “Applying Claims” section for an example predicate. 如果用户含有与指定谓词匹配的声明（Claim）时，返回true。参见“运用声明（Claims）”中的示例谓词
RemoveClaim(claim)	Removes a claim from the user identity. 删除用户的声明（Claim）。

Other members are available, but the ones in the table are those that are used most often in web applications, for reason that will become obvious as I demonstrate how claims fit into the wider ASP.NET platform.

还有一些可用的其它成员，但表中的这些是在Web应用程序中最常用的，随着我演示如何将声明（Claims）融入更宽泛的ASP.NET平台，它们为什么最常用就很显然了。

In Listing 15-12, I cast the IIdentity implementation to the ClaimsIdentity type and pass the enumeration of Claim objects returned by the ClaimsIdentity.Claims property to the View method. A Claim object represents a single piece of data about the user, and the Claim class defines the properties shown in Table 15-6.

在清单15-12中，我将IIdentity实现转换成了ClaimsIdentity类型，并且给View方法传递了ClaimsIdentity.Claims属性所返回的Claim对象的枚举。Claim对象所示表示的是关于用户的一个单

一的数据片段，Claim类定义的属性如表15-6所示。

Table 15-6. *The Properties Defined by the Claim Class*

表15-6. *Claim类定义的属性*

Name 名称	Description 描述
Issuer	Returns the name of the system that provided the claim 返回提供声明（Claim）的系统名称
Subject	Returns the ClaimsIdentity object for the user who the claim refers to 返回声明（Claim）所指用户的ClaimsIdentity对象
Type	Returns the type of information that the claim represents 返回声明（Claim）所表示的信息类型
Value	Returns the piece of information that the claim represents 返回声明（Claim）所表示的信息片段

Listing 15-13 shows the contents of the Index.cshtml file that I created in the Views/Claims folder and that is rendered by the Index action of the Claims controller. The view adds a row to a table for each claim about the user.

清单15-13显示了我在Views/Claims文件夹中创建的Index.cshtml文件的内容，它由Claims控制器中的Index动作方法进行渲染。该视图为用户的每项声明（Claim）添加了一个表格行。

Listing 15-13. *The Contents of the Index.cshtml File in the Views/Claims Folder*

清单15-13. *Views/Claims文件夹中Index.cshtml文件的内容*

```
@using System.Security.Claims
@using Users.Infrastructure
@model IEnumerable<Claim>
@{ ViewBag.Title = "Claims"; }

<div class="panel panel-primary">
    <div class="panel-heading">
        Claims
    </div>
    <table class="table table-striped">
        <tr>
            <th>Subject</th><th>Issuer</th>
            <th>Type</th><th>Value</th>
        </tr>
        @foreach (Claim claim in Model.OrderBy(x => x.Type)) {
            <tr>
                <td>@claim.Subject.Name</td>
                <td>@claim.Issuer</td>
                <td>@Html.ClaimType(claim.Type)</td>
                <td>@claim.Value</td>
```

```
            </tr>
        }
    </table>
</div>
```

The value of the `Claim.Type` property is a URI for a Microsoft schema, which isn't especially useful. The popular schemas are used as the values for fields in the `System.Security.Claims.ClaimTypes` class, so to make the output from the `Index.cshtml` view easier to read, I added an HTML helper to the `IdentityHelpers.cs` file, as shown in Listing 15-14. It is this helper that I use in the `Index.cshtml` file to format the value of the `Claim.Type` property.

`Claim.Type`属性的值是一个**微软模式**（Microsoft Schema）的URI（统一资源标识符），这是特别有用的。`System.Security.Claims.ClaimTypes`类中字段的值使用的是**流行模式**（Popular Schema），因此为了使`Index.cshtml`视图的输出更易于阅读，我在`IdentityHelpers.cs`文件中添加了一个HTML辅助器，如清单15-14所示。`Index.cshtml`文件正是使用这个辅助器格式化了`Claim.Type`属性的值。

Listing 15-14. Adding a Helper to the IdentityHelpers.cs File

清单15-14. 在IdentityHelpers.cs文件中添加辅助器

```
using System.Web;
using System.Web.Mvc;
using Microsoft.AspNet.Identity.Owin;
using System;
using System.Linq;
using System.Reflection;
using System.Security.Claims;

namespace Users.Infrastructure {
    public static class IdentityHelpers {

        public static MvcHtmlString GetUserName(this HtmlHelper html, string id) {
            AppUserManager mgr
                = HttpContext.Current.GetOwinContext().GetUserManager<AppUserManager>();
            return new MvcHtmlString(mgr.FindByIdAsync(id).Result.UserName);
        }

        public static MvcHtmlString ClaimType(this HtmlHelper html, string claimType) {
            FieldInfo[] fields = typeof(ClaimTypes).GetFields();
            foreach (FieldInfo field in fields) {
                if (field.GetValue(null).ToString() == claimType) {
                    return new MvcHtmlString(field.Name);
                }
            }
            return new MvcHtmlString(string.Format("{0}",
                claimType.Split('/', '.').Last()));
```

```
        }
    }
}
```

Note The helper method isn't at all efficient because it reflects on the fields of the `ClaimType` class for each claim that is displayed, but it is sufficient for my purposes in this chapter. You won't often need to display the claim type in real applications.

注：该辅助器并非十分有效，因为它只是针对每个要显示的声明（Claim）映射出`ClaimType`类的字段，但对我要的目的已经足够了。在实际项目中不会经常需要显示声明（Claim）的类型。

To see why I have created a controller that uses claims without really explaining what they are, start the application, authenticate as the user Alice (with the password `MySecret`), and request the /Claims/Index URL. Figure 15-5 shows the content that is generated.

为了弄明白我为何要先创建一个使用声明（Claims）的控制器，而没有真正解释声明（Claims）是什么的原因，可以启动应用程序，以用户Alice进行认证（其口令是`MySecret`），并请求`/Claims/Index` URL。图15-5显示了生成的内容。

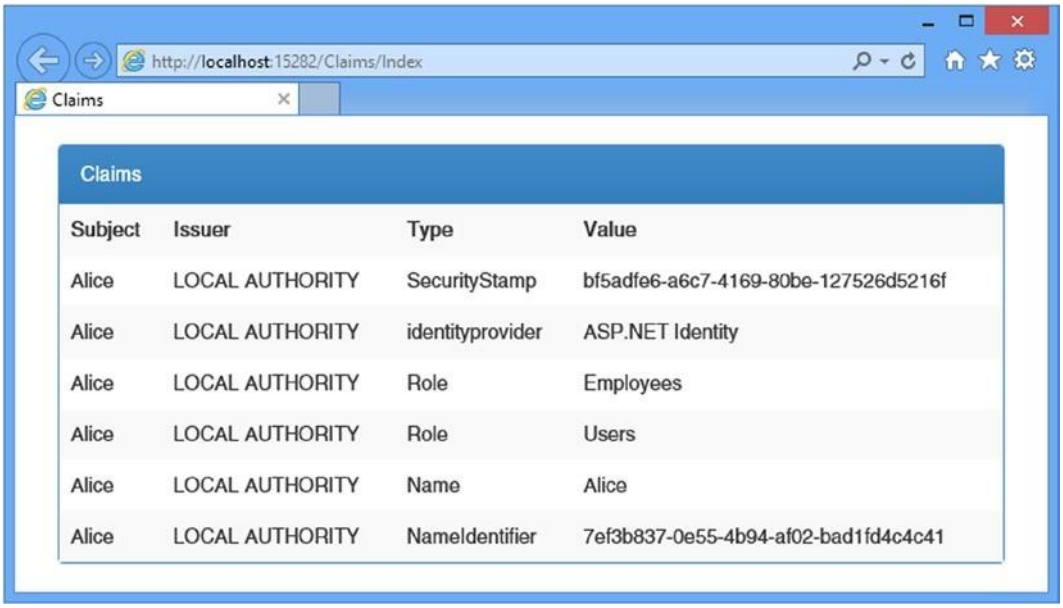

Figure 15-5. *The output from the Index action of the Claims controller*

图 15-5. *Claims 控制器中 Index 动作的输出*

It can be hard to make out the detail in the figure, so I have reproduced the content in Table 15-7.

这可能还难以认识到此图的细节，为此我在表15-7中重列了其内容。

Table 15-7. *The Data Shown in Figure 15-5*

表15-7. *图15-5中显示的数据*

Subject **科目**	**Issuer** **发行者**	**Type** **类型**	**Value** **值**
Alice	LOCAL AUTHORITY	SecurityStamp	Unique ID
Alice	LOCAL AUTHORITY	IdentityProvider	ASP.NET Identity

Alice	LOCAL AUTHORITY	Role	Employees
Alice	LOCAL AUTHORITY	Role	Users
Alice	LOCAL AUTHORITY	Name	Alice
Alice	LOCAL AUTHORITY	NameIdentifier	Alice's user ID

The table shows the most important aspect of claims, which is that I have already been using them when I implemented the traditional authentication and authorization features in Chapter 14. You can see that some of the claims relate to user identity (the `Name` claim is Alice, and the NameIdentifier claim is Alice's unique user ID in my ASP.NET Identity database).
此表展示了声明（Claims）最重要的方面，这些是我在第14章中实现传统的认证和授权特性时，一直在使用的信息。可以看出，有些声明（Claims）与用户标识有关（`Name`声明是`Alice`，`NameIdentifier`声明是Alice在ASP.NET Identity数据库中的唯一用户`ID`号）。

Other claims show membership of roles—there are two `Role` claims in the table, reflecting the fact that `Alice` is assigned to both the `Users` and `Employees` roles. There is also a claim about how Alice has been authenticated: The `IdentityProvider` is set to ASP.NET Identity.
其他声明（Claims）显示了角色成员——表中有两个`Role`声明（Claim），体现出`Alice`被赋予了`Users`和`Employees`两个角色这一事实。还有一个是`Alice`已被认证的声明（Claim）：`IdentityProvider`被设置到了ASP.NET Identity。

The difference when this information is expressed as a set of claims is that you can determine where the data came from. The `Issuer` property for all the claims shown in the table is set to LOCAL AUTHORITY, which indicates that the user's identity has been established by the application.
当这种信息被表示成一组声明（Claims）时的差别是，你能够确定这些数据是从哪里来的。表中所显示的所有声明的`Issuer`属性（发布者）都被设置到了`LOACL AUTHORITY`（本地授权），这说明该用户的标识是由应用程序建立的。

So, now that you have seen some example claims, I can more easily describe what a claim is. A claim is any piece of information about a user that is available to the application, including the user's identity and role memberships. And, as you have seen, the information I have been defining about my users in earlier chapters is automatically made available as claims by ASP.NET Identity.
因此，现在你已经看到了一些声明（Claims）示例，我可以更容易地描述声明（Claim）是什么了。一项声明（Claim）是可用于应用程序中的有关用户的一个信息片段，包括用户的标识以及角色成员等。而且，正如你所看到的，我在前几章定义的关于用户的信息，被ASP.NET Identity自动地作为声明（Claims）了。

15.3.2 Creating and Using Claims
15.3.2 创建和使用声明（Claims）

Claims are interesting for two reasons. The first reason is that an application can obtain claims from multiple sources, rather than just relying on a local database for information about the user. You will see a real example of this when I show you how to authenticate users through a third-party system in the "Using Third-Party Authentication" section, but for the moment I am going to add a class to the example project that simulates a system that provides claims information. Listing 15-15 shows the contents of the `LocationClaimsProvider.cs` file that I added to the `Infrastructure` folder.
声明（Claims）比较有意思的原因有两个。第一个原因是应用程序可以从多个来源获取声明（Claims），而不是只能依靠本地数据库关于用户的信息。你将会看到一个实际的示例，在“使用第三方认证”小节

中，将演示如何通过第三方系统来认证用户。不过，此刻我只打算在示例项目中添加一个类，用以模拟一个提供声明（Claims）信息的系统。清单15-15显示了我添加到Infrastructure文件夹中LocationClaimsProvider.cs文件的内容。

Listing 15-15. *The Contents of the LocationClaimsProvider.cs File*

清单15-15. *LocationClaimsProvider.cs文件的内容*

```
using System.Collections.Generic;
using System.Security.Claims;

namespace Users.Infrastructure {

    public static class LocationClaimsProvider {

        public static IEnumerable<Claim> GetClaims(ClaimsIdentity user) {
            List<Claim> claims = new List<Claim>();
            if (user.Name.ToLower() == "alice") {
                claims.Add(CreateClaim(ClaimTypes.PostalCode, "DC 20500"));
                claims.Add(CreateClaim(ClaimTypes.StateOrProvince, "DC"));
            } else {
                claims.Add(CreateClaim(ClaimTypes.PostalCode, "NY 10036"));
                claims.Add(CreateClaim(ClaimTypes.StateOrProvince, "NY"));
            }
            return claims;
        }

        private static Claim CreateClaim(string type, string value) {
            return new Claim(type, value, ClaimValueTypes.String, "RemoteClaims");
        }
    }
}
```

The GetClaims method takes a ClaimsIdentity argument and uses the Name property to create claims about the user's ZIP code and state. This class allows me to simulate a system such as a central HR database, which would be the authoritative source of location information about staff, for example.
GetClaims方法以ClaimsIdentity为参数，并使用Name属性创建了关于用户ZIP码（邮政编码）和州府的声明（Claims）。上述这个类使我能够模拟一个诸如中心化的HR数据库（人力资源数据库）之类的系统，它可能会成为全体职员的地点信息的权威数据源。

Claims are associated with the user's identity during the authentication process, and Listing 15-16 shows the changes I made to the Login action method of the Account controller to call the LocationClaimsProvider class.
在认证过程期间，声明（Claims）是与用户标识关联在一起的，清单15-16显示了我对Account控制器中Login动作方法所做的修改，以便调用LocationClaimsProvider类。

Listing 15-16. *Associating Claims with a User in the AccountController.cs File*

清单15-16. *AccountController.cs文件中用户用声明的关联*

```
...
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginModel details, string returnUrl) {
    if (ModelState.IsValid) {
        AppUser user = await UserManager.FindAsync(details.Name,
            details.Password);
        if (user == null) {
            ModelState.AddModelError("", "Invalid name or password.");
        } else {
            ClaimsIdentity ident = await UserManager.CreateIdentityAsync(user,
                DefaultAuthenticationTypes.ApplicationCookie);
            ident.AddClaims(LocationClaimsProvider.GetClaims(ident));
            AuthManager.SignOut();
            AuthManager.SignIn(new AuthenticationProperties {
                IsPersistent = false
            }, ident);
            return Redirect(returnUrl);
        }
    }
    ViewBag.returnUrl = returnUrl;
    return View(details);
}
...
```

You can see the effect of the location claims by starting the application, authenticating as a user, and requesting the `/Claim/Index` URL. Figure 15-6 shows the claims for Alice. You may have to sign out and sign back in again to see the change.

为了看看这个地点声明（Claims）的效果，可以启动应用程序，以一个用户进行认证，并请求`/Claim/Index` URL。图15-6显示了Alice的声明（Claims）。你可能需要退出，然后再次登录才会看到发生的变化。

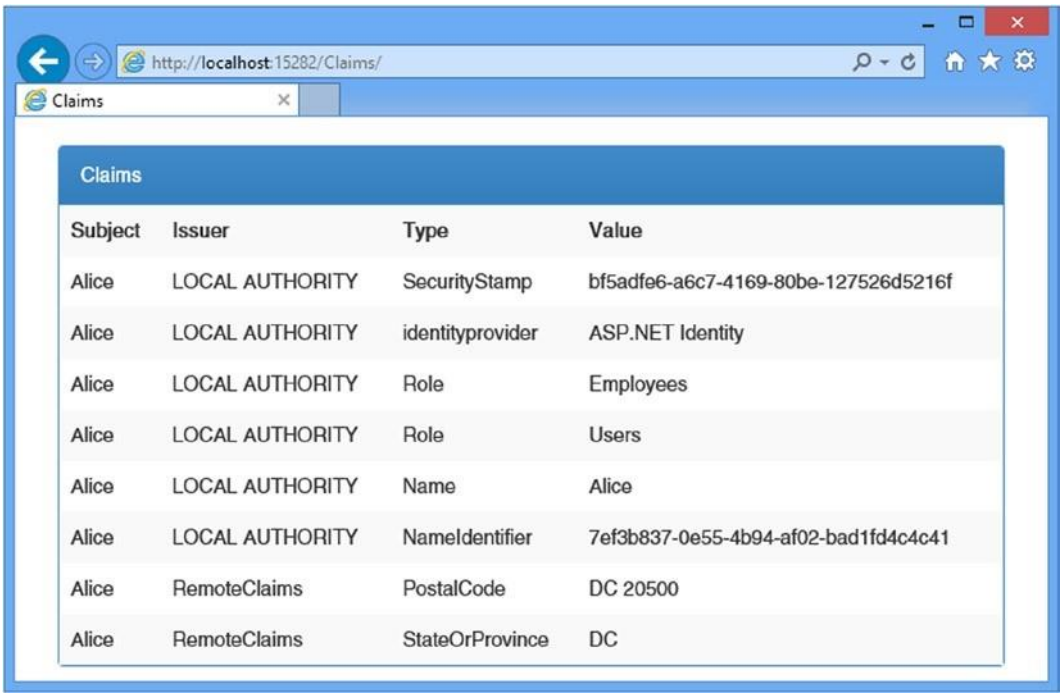

Figure 15-6. *Defining additional claims for users*

图15-6. *定义用户的附加声明*

Obtaining claims from multiple locations means that the application doesn't have to duplicate data that is held elsewhere and allows integration of data from external parties. The `Claim.Issuer` property tells you where a claim originated from, which helps you judge how accurate the data is likely to be and how much weight you should give the data in your application. Location data obtained from a central HR database is likely to be more accurate and trustworthy than data obtained from an external mailing list provider, for example.

从多个地点获取声明（Claims）意味着应用程序不必复制其他地方保持的数据，并且能够与外部的数据集成。`Claim.Issuer`属性（图15-6中的Issuer数据列——译者注）能够告诉你一个声明（Claim）的发源地，这有助于让你判断数据的精确程度，也有助于让你决定这类数据在应用程序中的权重。例如，从中心化的HR数据库获取的地点数据可能要比外部邮件列表提供器获取的数据更为精确和可信。

Applying Claims
运用声明（Claims）

The second reason that claims are interesting is that you can use them to manage user access to your application more flexibly than with standard roles. The problem with roles is that they are static, and once a user has been assigned to a role, the user remains a member until explicitly removed. This is, for example, how long-term employees of big corporations end up with incredible access to internal systems: They are assigned the roles they require for each new job they get, but the old roles are rarely removed. (The unexpectedly broad systems access sometimes becomes apparent during the investigation into how someone was able to ship the contents of the warehouse to their home address—true story.)

声明（Claims）有意思的第二个原因是，你可以用它们来管理用户对应用程序的访问，这要比标准的角色管理更为灵活。角色的问题在于它们是静态的，而且一旦用户已经被赋予了一个角色，该用户便是一个成员，直到明确地删除为止。例如，这意味着大公司的长期雇员，对内部系统的访问会十分惊人：他们每次在获得新工作时，都会赋予所需的角色，但旧角色很少被删除。（在调查某人为何能够将仓库里

的东西发往他的家庭地址过程中发现，有时会出现异常宽泛的系统访问——真实的故事）

Claims can be used to authorize users based directly on the information that is known about them, which ensures that the authorization changes when the data changes. The simplest way to do this is to generate `Role` claims based on user data that are then used by controllers to restrict access to action methods. Listing 15-17 shows the contents of the `ClaimsRoles.cs` file that I added to the `Infrastructure`.

声明（Claims）可以直接根据用户已知的信息对用户进行授权，这能够保证当数据发生变化时，授权也随之而变。此事最简单的做法是根据用户数据来生成`Role`声明（Claim），然后由控制器用来限制对动作方法的访问。清单15-17显示了我添加到`Infrastructure`中的`ClaimsRoles.cs`文件的内容。

Listing 15-17. *The Contents of the ClaimsRoles.cs File*

清单15-17. *ClaimsRoles.cs文件的内容*

```
using System.Collections.Generic;
using System.Security.Claims;

namespace Users.Infrastructure {
    public class ClaimsRoles {

    public static IEnumerable<Claim> CreateRolesFromClaims(ClaimsIdentity user) {
        List<Claim> claims = new List<Claim>();
        if (user.HasClaim(x => x.Type == ClaimTypes.StateOrProvince
                && x.Issuer == "RemoteClaims" && x.Value == "DC")
                && user.HasClaim(x => x.Type == ClaimTypes.Role
                && x.Value == "Employees")) {
            claims.Add(new Claim(ClaimTypes.Role, "DCStaff"));
        }
            return claims;
        }
    }
}
```

The gnarly looking `CreateRolesFromClaims` method uses lambda expressions to determine whether the user has a `StateOrProvince` claim from the `RemoteClaims` issuer with a value of `DC` and a `Role` claim with a value of `Employees`. If the user has both claims, then a `Role` claim is returned for the `DCStaff` role. Listing 15-18 shows how I call the `CreateRolesFromClaims` method from the `Login` action in the `Account` controller.

`CreateRolesFromClaims`是一个粗糙的考察方法，它使用了Lambda表达式，以检查用户是否具有`StateOrProvince`声明（Claim），该声明来自于`RemoteClaims`发行者（Issuer），值为`DC`。也检查用户是否具有`Role`声明（Claim），其值为`Employees`。如果用户这两个声明都有，那么便返回一个`DCStaff`角色的`Role`声明。清单15-18显示了如何在`Account`控制器中的`Login`动作中调用`CreateRolesFromClaims`方法。

Listing 15-18. *Generating Roles Based on Claims in the AccountController.cs File*

清单15-18. *在AccountController.cs中根据声明生成角色*

```
...
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginModel details, string returnUrl) {
    if (ModelState.IsValid) {
        AppUser user = await UserManager.FindAsync(details.Name,
            details.Password);
        if (user == null) {
            ModelState.AddModelError("", "Invalid name or password.");
        } else {
            ClaimsIdentity ident = await UserManager.CreateIdentityAsync(user,
                DefaultAuthenticationTypes.ApplicationCookie);
            ident.AddClaims(LocationClaimsProvider.GetClaims(ident));
            ident.AddClaims(ClaimsRoles.CreateRolesFromClaims(ident));
            AuthManager.SignOut();
            AuthManager.SignIn(new AuthenticationProperties {
                IsPersistent = false
            }, ident);
            return Redirect(returnUrl);
        }
    }
    ViewBag.returnUrl = returnUrl;
    return View(details);
}
...
```

I can then restrict access to an action method based on membership of the `DCStaff` role. Listing 15-19 shows a new action method I added to the `Claims` controller to which I have applied the `Authorize` attribute.

然后我可以根据`DCStaff`角色的成员，来限制对一个动作方法的访问。清单15-19显示了在`Claims`控制器中添加的一个新的动作方法，在该方法上已经运用了`Authorize`注解属性。

Listing 15-19. *Adding a New Action Method to the ClaimsController.cs File*

清单15-19. *在ClaimsController.cs文件中添加一个新的动作方法*

```
using System.Security.Claims;
using System.Web;
using System.Web.Mvc;

namespace Users.Controllers {
    public class ClaimsController : Controller {
```

```
        [Authorize]
        public ActionResult Index() {
            ClaimsIdentity ident = HttpContext.User.Identity as ClaimsIdentity;
            if (ident == null) {
                return View("Error", new string[] { "No claims available" });
            } else {
                return View(ident.Claims);
            }
        }

        [Authorize(Roles="DCStaff")]
        public string OtherAction() {
            return "This is the protected action";
        }
    }
}
```

Users will be able to access `OtherAction` only if their claims grant them membership to the `DCStaff` role. Membership of this role is generated dynamically, so a change to the user's employment status or location information will change their authorization level.
只要用户的声明（Claims）承认他们是`DCStaff`角色的成员，那么他们便能访问`OtherAction`动作。该角色的成员是动态生成的，因此，若是用户的雇用状态或地点信息发生变化，也会改变他们的授权等级。

提示：请读者从这个例子中吸取其中的思想精髓。仁者见仁，智者见智，能领悟多少，全凭各人，译者感觉这里的思想有无数的可能。举例说明：（1）可以根据用户的身份进行授权，比如学生在校时是“学生”，离校时便是“校友”；（2）可以根据用户所处的部门进行授权，人事部用户属于人事团队，销售部用户属于销售团队；（3）下一小节的示例是根据用户的地点授权。简言之：一方面用户的各种声明（Claim）都可以用来进行授权；另一方面用户的声明（Claim）又是可以自定义的。于是可能的运用就无法估计了。总之一句话，这种*基于声明的授权（Claims-Based Authorization）*有无限可能！要是没有我这里的提示，是否所有读者都会有这样的体会？——译者注

15.3.3 Authorizing Access Using Claims
15.3.3 使用声明（Claims）授权访问

The previous example is an effective demonstration of how claims can be used to keep authorizations fresh and accurate, but it is a little indirect because I generate roles based on claims data and then enforce my authorization policy based on the membership of that role. A more direct and flexible approach is to enforce authorization directly by creating a custom authorization filter attribute. Listing 15-20 shows the contents of the `ClaimsAccessAttribute.cs` file, which I added to the `Infrastructure` folder and used to create such a filter.
前面的示例有效地演示了如何用声明（Claims）来保持新鲜和准确的授权，但有点不太直接，因为我要根据声明（Claims）数据来生成了角色，然后强制我的授权策略基于角色成员。一个更直接且灵活的办法是直接强制授权，其做法是创建一个自定义的授权过滤器注解属性。清单15-20演示了`ClaimsAccessAttribute.cs`文件的内容，我将它添加在`Infrastructure`文件夹中，并用它创建了这

种过滤器。

Listing 15-20. *The Contents of the ClaimsAccessAttribute.cs File*

清单15-20. *ClaimsAccessAttribute.cs文件的内容*

```
using System.Security.Claims;
using System.Web;
using System.Web.Mvc;

namespace Users.Infrastructure {
    public class ClaimsAccessAttribute : AuthorizeAttribute {

        public string Issuer { get; set; }
        public string ClaimType { get; set; }
        public string Value { get; set; }

        protected override bool AuthorizeCore(HttpContextBase context) {
            return context.User.Identity.IsAuthenticated
                && context.User.Identity is ClaimsIdentity
                && ((ClaimsIdentity)context.User.Identity).HasClaim(x =>
                    x.Issuer == Issuer && x.Type == ClaimType && x.Value == Value
                );
        }
    }
}
```

The attribute I have defined is derived from the `AuthorizeAttribute` class, which makes it easy to create custom authorization policies in MVC framework applications by overriding the `AuthorizeCore` method. My implementation grants access if the user is authenticated, the `IIdentity` implementation is an instance of `ClaimsIdentity`, and the user has a claim with the `issuer`, `type`, and `value` matching the class properties. Listing 15-21 shows how I applied the attribute to the `Claims` controller to authorize access to the `OtherAction` method based on one of the location claims created by the `LocationClaimsProvider` class.

我所定义的这个注解属性派生于`AuthorizeAttribute`类，通过重写`AuthorizeCore`方法，很容易在MVC框架应用程序中创建自定义的授权策略。在这个实现中，若用户是已认证的、其`IIdentity`实现是一个`ClaimsIdentity`实例，而且该用户有一个带有`issuer`、`type`以及`value`的声明（Claim），它们与这个类的属性是匹配的，则该用户便是允许访问的。清单15-21显示了如何将这个注解属性运用于`Claims`控制器，以便根据`LocationClaimsProvider`类创建的地点声明（Claim），对`OtherAction`方法进行授权访问。

Listing 15-21. *Performing Authorization on Claims in the ClaimsController.cs File*

清单15-21. *在ClaimsController.cs文件中执行基于声明的授权*

```
using System.Security.Claims;
```

```
using System.Web;
using System.Web.Mvc;
using Users.Infrastructure;

namespace Users.Controllers {
    public class ClaimsController : Controller {

        [Authorize]
        public ActionResult Index() {
            ClaimsIdentity ident = HttpContext.User.Identity as ClaimsIdentity;
            if (ident == null) {
                return View("Error", new string[] { "No claims available" });
            } else {
                return View(ident.Claims);
            }
        }
        [ClaimsAccess(Issuer="RemoteClaims", ClaimType=ClaimTypes.PostalCode,
            Value="DC 20500")]
        public string OtherAction() {
            return "This is the protected action";
        }
    }
}
```

My authorization filter ensures that only users whose location claims specify a ZIP code of DC 20500 can invoke the `OtherAction` method.
这个授权过滤器能够确保只有地点声明（Claim）的邮编为`DC 20500`的用户才能请求`OtherAction`方法。

15.4 Using Third-Party Authentication

15.4 使用第三方认证

One of the benefits of a claims-based system such as ASP.NET Identity is that any of the claims can come from an external system, even those that identify the user to the application. This means that other systems can authenticate users on behalf of the application, and ASP.NET Identity builds on this idea to make it simple and easy to add support for authenticating users through third parties such as Microsoft, Google, Facebook, and Twitter.
基于声明的系统，如ASP.NET Identity，的好处之一是任何声明都可以来自于外部系统，即使是将用户标识到应用程序的那些声明。这意味着其他系统可以代表应用程序来认证用户，而ASP.NET Identity就建立在这样的思想之上，使之能够简单而方便地添加第三方认证用户的支持，如微软、Google、Facebook、Twitter等。

There are some substantial benefits of using third-party authentication: Many users will already have an account, users can elect to use two-factor authentication, and you don’t have to manage user credentials in

the application. In the sections that follow, I'll show you how to set up and use third-party authentication for Google users, which Table 15-8 puts into context.
使用第三方认证有一些实际的好处：许多用户已经有了账号、用户可以选择使用双因子认证、你不必在应用程序中管理用户凭据等等。在以下小节中，我将演示如何为Google用户建立并使用第三方认证，表15-8描述了事情的情形。

Table 15-8. *Putting Third-Party Authentication in Context*
表15-8. *第三方认证情形*

Question **问题**	**Answer** **回答**
What is it? 什么是第三方认证？	Authenticating with third parties lets you take advantage of the popularity of companies such as Google and Facebook. 第三方认证使你能够利用流行公司，如Google和Facebook，的优势。
Why should I care? 为何要关心它？	Users don't like having to remember passwords for many different sites. Using a provider with large-scale adoption can make your application more appealing to users of the provider's services. 用户不喜欢记住许多不同网站的口令。使用大范围适应的提供器可使你的应用程序更吸引有提供器服务的用户。
How is it used by the MVC framework? 如何在MVC框架中使用它？	This feature isn't used directly by the MVC framework. 这不是一个直接由MVC框架使用的特性。

Note The reason I have chosen to demonstrate Google authentication is that it is the only option that doesn't require me to register my application with the authentication service. You can get details of the registration processes required at http://bit.ly/1cqLTrE.
提示：我选择演示Google认证的原因是，它是唯一不需要在其认证服务中注册我应用程序的公司。有关认证服务注册过程的细节，请参阅http://bit.ly/1cqLTrE。

15.4.1 Enabling Google Authentication
15.4.1 启用 Google 认证

ASP.NET Identity comes with built-in support for authenticating users through their Microsoft, Google, Facebook, and Twitter accounts as well more general support for any authentication service that supports OAuth. The first step is to add the NuGet package that includes the Google-specific additions for ASP.NET Identity. Enter the following command into the Package Manager Console:
ASP.NET Identity带有通过Microsoft、Google、Facebook以及Twitter账号认证用户的内建支持，并且对于支持OAuth的认证服务具有更普遍的支持。第一个步骤是添加NuGet包，包中含有用于ASP.NET Identity的Google专用附件。请在“Package Manager Console（包管理器控制台）”中输入以下命令：

```
Install-Package Microsoft.Owin.Security.Google -version 2.0.2
```

There are NuGet packages for each of the services that ASP.NET Identity supports, as described in Table 15-9.
对于ASP.NET Identity支持的每一种服务都有相应的NuGet包，如表15-9所示。

Table 15-9. *The NuGet Authenticaton Packages*

***表15-9.** NuGet认证包*

Name **名称**	**Description** **描述**
`Microsoft.Owin.Security.Google`	Authenticates users with Google accounts 用Google账号认证用户
`Microsoft.Owin.Security.Facebook`	Authenticates users with Facebook accounts 用Facebook账号认证用户
`Microsoft.Owin.Security.Twitter`	Authenticates users with Twitter accounts 用Twitter账号认证用户
`Microsoft.Owin.Security.MicrosoftAccount`	Authenticates users with Microsoft accounts 用Microsoft账号认证用户
`Microsoft.Owin.Security.OAuth`	Authenticates users against any OAuth 2.0 service 根据任一OAuth 2.0服务认证用户

Once the package is installed, I enable support for the authentication service in the OWIN startup class, which is defined in the `App_Start/IdentityConfig.cs` file in the example project. Listing 15-22 shows the change that I have made.

一旦安装了这个包，便可以在OWIN启动类中启用此项认证服务的支持，启动类的定义在示例项目的`App_Start/IdentityConfig.cs`文件中。清单15-22显示了所做的修改。

***Listing 15-22.** Enabling Google Authentication in the IdentityConfig.cs File*

***清单15-22.** 在IdentityConfig.cs文件中启用Google认证*

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;
using Users.Infrastructure;
using Microsoft.Owin.Security.Google;

namespace Users {
    public class IdentityConfig {
        public void Configuration(IAppBuilder app) {

            app.CreatePerOwinContext<AppIdentityDbContext>(AppIdentityDbContext.Create);
            app.CreatePerOwinContext<AppUserManager>(AppUserManager.Create);
            app.CreatePerOwinContext<AppRoleManager>(AppRoleManager.Create);

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

            app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);
```

```
            app.UseGoogleAuthentication();
        }
    }
}
```

Each of the packages that I listed in Table 15-9 contains an extension method that enables the corresponding service. The extension method for the Google service is called `UseGoogleAuthentication`, and it is called on the `IAppBuilder` implementation that is passed to the `Configuration` method.
表15-9所列的每个包都含有启用相应服务的扩展方法。用于Google服务的扩展方法名称为`UseGoogleAuthentication`，它通过传递给`Configuration`方法的`IAppBuilder`实现进行调用。

Next I added a button to the `Views/Account/Login.cshtml` file, which allows users to log in via Google. You can see the change in Listing 15-23.
下一步骤是在`Views/Account/Login.cshtml`文件中添加一个按钮，让用户能够通过Google进行登录。所做的修改如清单15-23所示。

Listing 15-23. *Adding a Google Login Button to the Login.cshtml File*
清单15-23. *在Login.cshtml文件中添加Google登录按钮*

```
@model Users.Models.LoginModel
@{ ViewBag.Title = "Login";}
<h2>Log In</h2>

@Html.ValidationSummary()

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

@using (Html.BeginForm("GoogleLogin", "Account")) {
    <input type="hidden" name="returnUrl" value="@ViewBag.returnUrl" />
    <button class="btn btn-primary" type="submit">Log In via Google</button>
}
```

The new button submits a form that targets the `GoogleLogin` action on the `Account` controller. You

can see this method—and the other changes I made the controller—in Listing 15-24.
新按钮递交一个表单，目标是Account控制器中的GoogleLogin动作。可从清单15-24中看到该方法，以及在控制器中所做的其他修改。

Listing 15-24. *Adding Support for Google Authentication to the AccountController.cs File*
清单15-24. *在AccountController.cs文件中添加Google认证支持*

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

namespace Users.Controllers {

    [Authorize]
    public class AccountController : Controller {

        [AllowAnonymous]
        public ActionResult Login(string returnUrl) {
             if (HttpContext.User.Identity.IsAuthenticated) {
                return View("Error", new string[] { "Access Denied" });
            }
              ViewBag.returnUrl = returnUrl;
            return View();
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> Login(LoginModel details, string returnUrl) {
            if (ModelState.IsValid) {
                AppUser user = await UserManager.FindAsync(details.Name,
                    details.Password);
                 if (user == null) {
                    ModelState.AddModelError("", "Invalid name or password.");
                } else {
                    ClaimsIdentity ident = await UserManager.CreateIdentityAsync(user,
                        DefaultAuthenticationTypes.ApplicationCookie);
```

```
            ident.AddClaims(LocationClaimsProvider.GetClaims(ident));
            ident.AddClaims(ClaimsRoles.CreateRolesFromClaims(ident));

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

[HttpPost]
[AllowAnonymous]
public ActionResult GoogleLogin(string returnUrl) {
     var properties = new AuthenticationProperties {
        RedirectUri = Url.Action("GoogleLoginCallback",
             new { returnUrl = returnUrl})
    };
    HttpContext.GetOwinContext().Authentication.Challenge(properties, "Google");
     return new HttpUnauthorizedResult();
}

[AllowAnonymous]
public async Task<ActionResult> GoogleLoginCallback(string returnUrl) {
    ExternalLoginInfo loginInfo = await AuthManager.GetExternalLoginInfoAsync();
    AppUser user = await UserManager.FindAsync(loginInfo.Login);
    if (user == null) {
         user = new AppUser {
            Email = loginInfo.Email,
            UserName = loginInfo.DefaultUserName,
            City = Cities.LONDON, Country = Countries.UK
        };
        IdentityResult result = await UserManager.CreateAsync(user);
        if (!result.Succeeded) {
            return View("Error", result.Errors);
        } else {
            result = await UserManager.AddLoginAsync(user.Id, loginInfo.Login);
            if (!result.Succeeded) {
                return View("Error", result.Errors);
            }
        }
```

```
            }

            ClaimsIdentity ident = await UserManager.CreateIdentityAsync(user,
                DefaultAuthenticationTypes.ApplicationCookie);
             ident.AddClaims(loginInfo.ExternalIdentity.Claims);
             AuthManager.SignIn(new AuthenticationProperties {
                IsPersistent = false }, ident);
             return Redirect(returnUrl ?? "/");
        }

        [Authorize]
        public ActionResult Logout() {
            AuthManager.SignOut();
            return RedirectToAction("Index", "Home");
        }

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

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

The `GoogleLogin` method creates an instance of the `AuthenticationProperties` class and sets the `RedirectUri` property to a URL that targets the `GoogleLoginCallback` action in the same controller. The next part is a magic phrase that causes ASP.NET Identity to respond to an unauthorized error by redirecting the user to the Google authentication page, rather than the one defined by the application:
`GoogleLogin`方法创建了`AuthenticationProperties`类的一个实例，并为`RedirectUri`属性设置了一个URL，其目标为同一控制器中的`GoogleLoginCallback`动作。下一个部分是一个神奇阶段，通过将用户重定向到Google认证页面，而不是应用程序所定义的认证页面，让ASP.NET Identity对未授权的错误进行响应：

```
...
HttpContext.GetOwinContext().Authentication.Challenge(properties, "Google");
return new HttpUnauthorizedResult();
...
```

This means that when the user clicks the Log In via Google button, their browser is redirected to the Google authentication service and then redirected back to the GoogleLoginCallback action method once they are authenticated.
这意味着，当用户通过点击Google按钮进行登录时，浏览器被重定向到Google的认证服务，一旦在那里认证之后，便被重定向回GoogleLoginCallback动作方法。

I get details of the external login by calling the GetExternalLoginInfoAsync of the IAuthenticationManager implementation, like this:
我通过调用IAuthenticationManager实现的GetExternalLoginInfoAsync方法，我获得了外部登录的细节，如下所示：

```
...
ExternalLoginInfo loginInfo = await AuthManager.GetExternalLoginInfoAsync();
...
```

The ExternalLoginInfo class defines the properties shown in Table 15-10.
ExternalLoginInfo类定义的属性如表15-10所示：

Table 15-10. *The Properties Defined by the ExternalLoginInfo Class*
表15-10. *ExternalLoginInfo类所定义的属性*

Name 名称	Description 描述
DefaultUserName	Returns the username 返回用户名
Email	Returns the e-mail address 返回E-mail地址
ExternalIdentity	Returns a ClaimsIdentity that identities the user 返回标识该用户的ClaimsIdentity
Login	Returns a UserLoginInfo that describes the external login 返回描述外部登录的UserLoginInfo

I use the FindAsync method defined by the user manager class to locate the user based on the value of the ExternalLoginInfo.Login property, which returns an AppUser object if the user has been authenticated with the application before:
我使用了由用户管理器类所定义的FindAsync方法，以便根据ExternalLoginInfo.Login属性的值对用户进行定位，如果用户之前在应用程序中已经认证，该属性会返回一个AppUser对象：

```
...
AppUser user = await UserManager.FindAsync(loginInfo.Login);
...
```

If the FindAsync method doesn't return an AppUser object, then I know that this is the first time that this user has logged into the application, so I create a new AppUser object, populate it with values, and save it to the database. I also save details of how the user logged in so that I can find them next time:
如果FindAsync方法返回的不是AppUser对象，那么我便知道这是用户首次登录应用程序，于是便创建

了一个新的AppUser对象，填充该对象的值，并将其保存到数据库。我还保存了用户如何登录的细节，以便下次能够找到他们：

```
...
result = await UserManager.AddLoginAsync(user.Id, loginInfo.Login);
...
```

All that remains is to generate an identity the user, copy the claims provided by Google, and create an authentication cookie so that the application knows the user has been authenticated:
剩下的事情只是生成该用户的标识了，拷贝Google提供的声明（Claims），并创建一个认证Cookie，以使应用程序知道此用户已认证：

```
...
ClaimsIdentity ident = await UserManager.CreateIdentityAsync(user,
    DefaultAuthenticationTypes.ApplicationCookie);
ident.AddClaims(loginInfo.ExternalIdentity.Claims);
AuthManager.SignIn(new AuthenticationProperties { IsPersistent = false }, ident);
...
```

15.4.2 Testing Google Authentication
15.4.2 测试 Google 认证

There is one further change that I need to make before I can test Google authentication: I need to change the account verification I set up in Chapter 13 because it prevents accounts from being created with e-mail addresses that are not within the example.com domain. Listing 15-25 shows how I removed the verification from the AppUserManager class.
在测试Google认证之前还需要一处修改：需要修改第13章所建立的账号验证，因为它不允许example.com域之外的E-mail地址创建账号。清单15-25显示了如何在AppUserManager类中删除这种验证。

Listing 15-25. *Disabling Account Validation in the AppUserManager.cs File*
清单15-25. *在AppUserManager.cs文件中取消账号验证*

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Users.Models;

namespace Users.Infrastructure {
    public class AppUserManager : UserManager<AppUser> {

        public AppUserManager(IUserStore<AppUser> store)
            : base(store) {
        }
```

```
        public static AppUserManager Create(
                IdentityFactoryOptions<AppUserManager> options,
                IOwinContext context) {

            AppIdentityDbContext db = context.Get<AppIdentityDbContext>();
            AppUserManager manager = new AppUserManager(new UserStore<AppUser>(db));

            manager.PasswordValidator = new CustomPasswordValidator {
                RequiredLength = 6,
                RequireNonLetterOrDigit = false,
                RequireDigit = false,
                RequireLowercase = true,
                RequireUppercase = true
            };

            //manager.UserValidator = new CustomUserValidator(manager) {
            //    AllowOnlyAlphanumericUserNames = true,
            //    RequireUniqueEmail = true
            //};

            return manager;
        }
    }
}
```

Tip you can use validation for externally authenticated accounts, but I am just going to disable the feature for simplicity.

提示：也可以使用外部已认证账号的验证，但这里出于简化，取消了这一特性。

To test authentication, start the application, click the Log In via Google button, and provide the credentials for a valid Google account. When you have completed the authentication process, your browser will be redirected back to the application. If you navigate to the `/Claims/Index` URL, you will be able to see how claims from the Google system have been added to the user's identity, as shown in Figure 15-7.

为了测试认证，启动应用程序，通过点击“Log In via Google（通过Google登录）”按钮，并提供有效的Google账号凭据。当你完成了认证过程时，浏览器将被重定向回应用程序。如果导航到`/Claims/Index` URL，便能够看到来自Google系统的声明（Claims），已被添加到用户的标识中了，如图15-7所示。

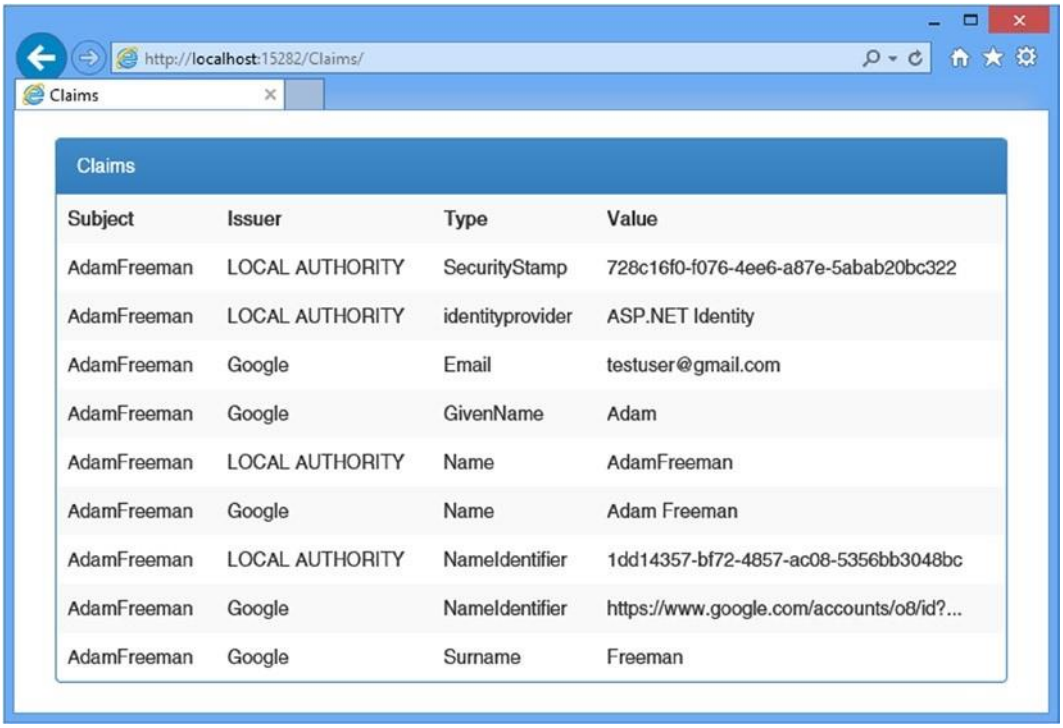

Figure 15-7. *Claims from Google*

图 15-7. *来自 Google 的声明（Claims）*

15.5 Summary

15.5 小结

In this chapter, I showed you some of the advanced features that ASP.NET Identity supports. I demonstrated the use of custom user properties and how to use database migrations to preserve data when you upgrade the schema to support them. I explained how claims work and how they can be used to create more flexible ways of authorizing users. I finished the chapter by showing you how to authenticate users via Google, which builds on the ideas behind the use of claims.

本章向你演示了ASP.NET Identity所支持的一些高级特性。演示了自定义用户属性的使用，还演示了在升级数据架构时，如何使用数据库迁移保护数据。我解释了声明（Claims）的工作机制，以及如何将它们用于创建更灵活的用户授权方式。最后演示了如何通过Google进行认证结束了本章，这是建立在使用声明（Claims）的思想基础之上的。

And that is all I have to teach you about the ASP.NET platform and how it supports MVC framework applications. I started by exploring the request handling life cycle and how it can be extended, managed, and disrupted. I then took you on a tour of the services that ASP.NET provides, showing you how they can be used to enhance and optimize your applications and improve the experience you deliver to your users. When writing MVC framework applications, it is easy to take the ASP.NET platform for granted, but this book has shown you just how much low-level functionality is available and how valuable it can be. I wish you every success in your web application projects, and I can only hope you have enjoyed reading this book as much as I enjoyed writing it.

以上便是我必须教给你的有关ASP.NET平台及其如何支持MVC框架的全部内容。我首先考察了请求处理

生命周期以及如何对它进行扩展、管理和干扰。然后带你浏览了ASP.NET所提供的服务，演示了如何将它们用于增强和优化应用程序并改善用户体验。在编写MVC框架应用程序时，获得ASP.NET平台的好处是理所当然的，但本书只是演示了一些可用的低级功能及其价值。我希望你在Web应用程序项目中获得巨大成功，也希望你阅读本书能够像我写作一样愉快。