**Глава 38.**

**Использование ASP.NET Core Identity**

ASP.NET Core Identity это API от Microsoft для управления пользователями в приложениях ASP.NET Core и включает поддержку аутентификации и авторизации в пайплайн запроса.

ASP.NET Core Identity это набор инструментов, с которым вы создаете функциональность аутентификации и авторизации, требуемую в приложении. В нем содержится бесконечное количество опций для настроек функциональности, такие как двухфакторная аутентификация, федерация, единый вход и самообслуживание учетной записи. Имеются опции, нужные только в большом корпоративном окружении или при использовании облачного управления пользователями.

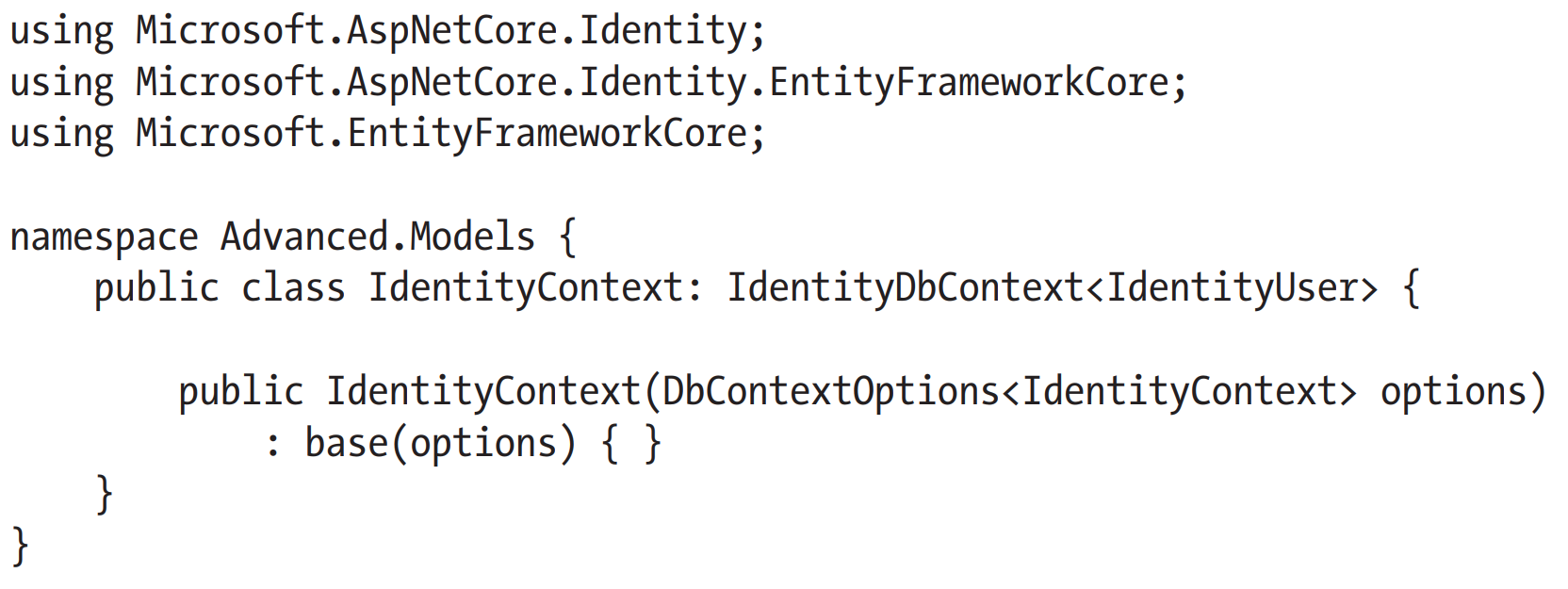
ASP.NET Core Identity развилась в собственный фреймворк и очень большая для описания в деталях в этой книге. Вместо этого я сфокусируюсь на части Identity API, интересной для разработки веб приложений, также как я сделал для Entity Framework Core. В этой главе я покажу, как добавить ASP.NET Core Identity в проект и объясню, как использовать ASP.NET Core Identity API для создания инструментов базового управления пользователями и ролями. В главе 39 я покажу, как использовать ASP.NET Core Identity для аутентификации пользователей и для выполнения авторизации.

**Подготовка проекта для ASP.NET Core Identity**

Процесс установки ASP.NET Core Identity требует установки пакета, конфигурации приложения и подготовки БД. Для начала установим пакет Microsoft.AspNetCore.Identity.EntityFrameworkCore.

**Подготовка базы данных для ASP.NET Core Identity**

Для ASP.NET Core Identity требуется БД, управляемая Entity Framework Core. Для создания класса контекста Entity Framework Core, который будет предоставлять Identity данные, добавим класс IdentityContext.cs в папку Advanced/Models:

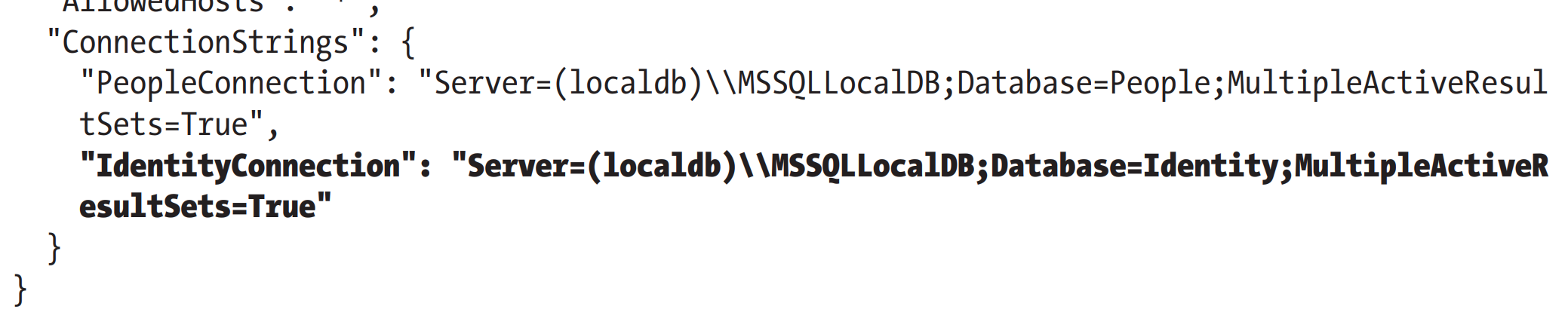


Пакет ASP.NET Core Identity включает в себя класс IdentityDbContext<Т>, используемый для создания класса контекста Entity Framework Core. Дженерик тип аргумента Т используется для указания класса, который будет представлять пользователей в БД. Вы можете создать кастомные классы пользователей, но я использовал базовый класс IdentityUser, предоставляющий основную функциональность Identity.

**Примечание**: Не беспокойтесь, если не понятен смысл классов используемых в вышеуказанном листинге. Если вы не знакомы с Entity Framework Core, тогда я предлагаю вам относиться к классу как к черному ящику. Изменения редко требуются после того, как строительные блоки для ASP.NET Core Identity настроены, и вы можете скопировать файлы из этой главы в ваши собственные проекты.

**Конфигурация строки подключения к базе данных**

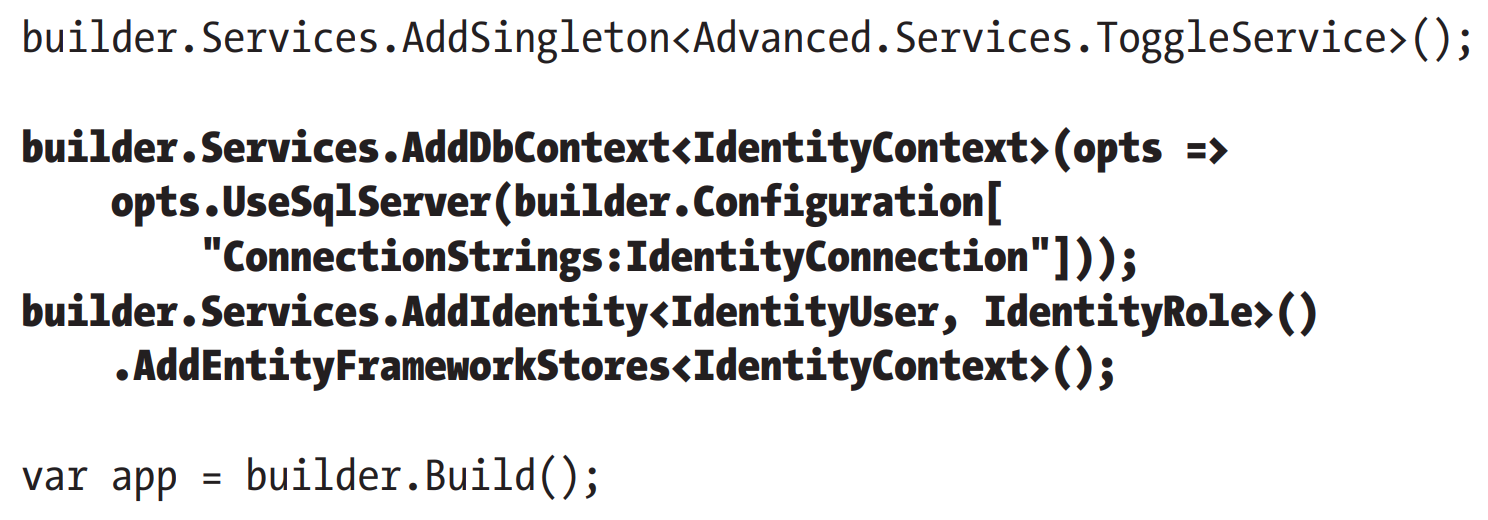
Строка подключения необходима для указания ASP.NET Core Identity места хранения его данных. Добавим строку подключения в файл appsettings.json рядом со строкой подключения, используемой для данных приложения:



Строка подключения устанавливает базу данных LocalDB с именем Identity.

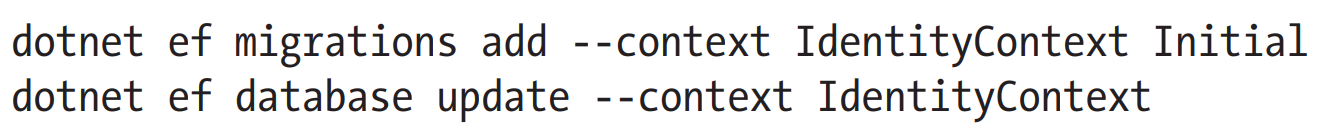
**Конфигурация приложения**

Следующим шагом будет настроить ASP.NET Core так чтобы контекст базы данных Identity был установлен как сервис:



**Создание и применение миграции базы данных Identity**

Оставшийся шаг, это создание миграции базы данных Entity Framework Core и применение ее для создания базы данных. Откроем новое окно PowerShell перейдем в папку проекта Advanced и запустим команды:



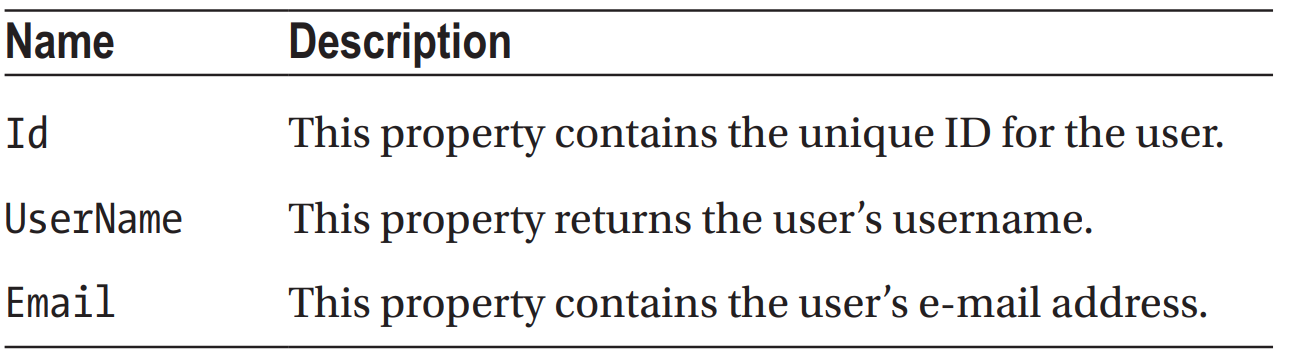
Как я объяснял в прошлых главах Entity Framework Core управляет изменением схемы БД через функциональность, называемую миграциями. Теперь, когда в проекте есть два класса контекста базы данных, инструментам Entity Framework Core необходим аргумент --context, чтобы определить какой класс контекста будет использован. Вышеуказанные команды создают миграцию, содержащую схему ASP.NET Core Identity и применяют ее к БД.

**СБРОС БАЗЫ ДАННЫХ ASP.NET CORE IDENTITY**

Если вам нужно сбросить базу данных, запустите команду «dotnet ef database drop --force --context IdentityContext» в папке проекта Advanced и затем запустите команду «dotnet ef database update --context IdentityContext». Это удалит существующую БД, а взамен создаст новую пустую. Не используйте эти команды на продуктовой системе, потому что вы можете удалить пользовательские реквизиты для входа. Если вам нужно переустановить главную БД, запустите команду «dotnet ef database drop --force --context DataContext», затем команду «dotnet ef database update --context DataContext».

**Создание инструментов менеджмента пользователей**

В этой секции я собираюсь создать инструменты менеджмента пользователей, используя ASP.NET Core Identity. Управление пользователями осуществляется через класс UserManager<T>, где Т, это класс, используемый для представления пользователей в БД. Когда я создавал класс контекста Entity Framework Core, я задал класс IdentityUser для представления пользователей в БД. Это предустановленный класс, предоставляемый ASP.NET Core Identity, и он предоставляет основную функциональность, необходимую в большинстве приложений. В таблице описаны наиболее полезные свойства класса IdentityUser (имеются также и дополнительные свойства, определенные в классе IdentityUser, но эти необходимы в большинстве проектов и только их я использую в данной книге):

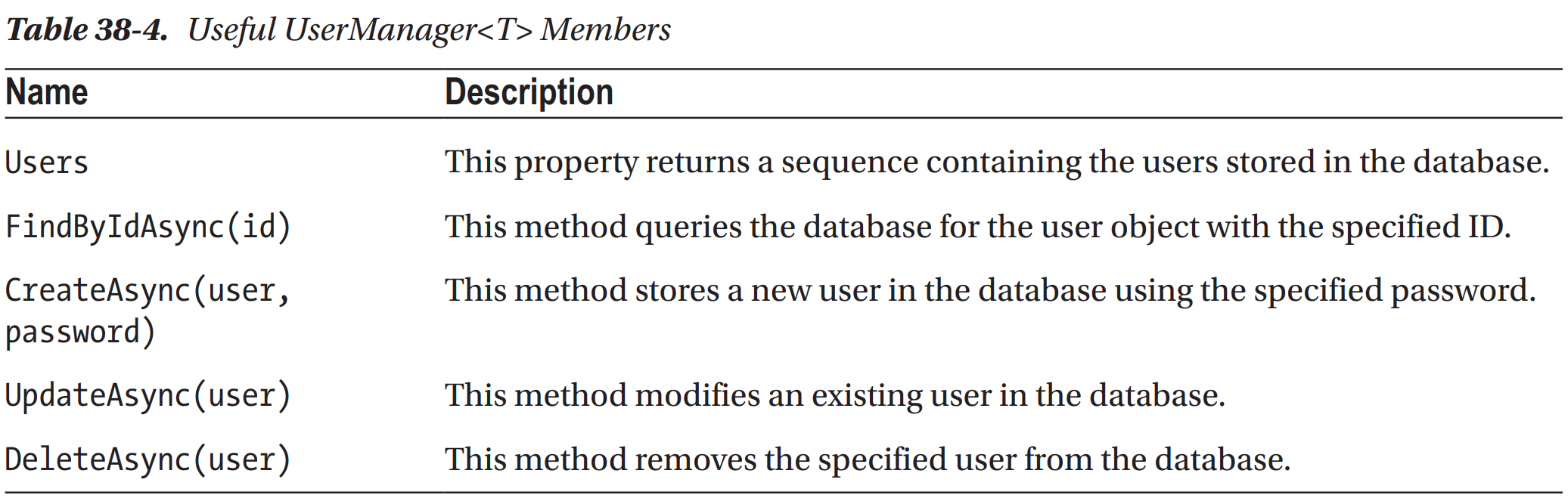


**Вспомогательные инструменты для менеджмента Identity**

Microsoft предоставляет инструмент, генерирующий набор Razor Pages для управления пользователями. Инструмент добавляет дженерик контент, знакомый как scaffolding из шаблона проекта, который вы затем адаптируете к приложению. Я не поклонник scaffolding или шаблонов и это не станет исключением. Шаблоны Microsoft Identity хорошо продуманы, но имеют ограниченное использование, потому что они сфокусированы на самоуправление, позволяющее пользователям создавать аккаунты, изменять пароли и т.д. без вмешательства администратора. Вы можете адаптировать шаблоны, чтобы ограничить круг задач, выполняемых пользователями, но предпосылка, стоящая за функциональностью, остается прежней.

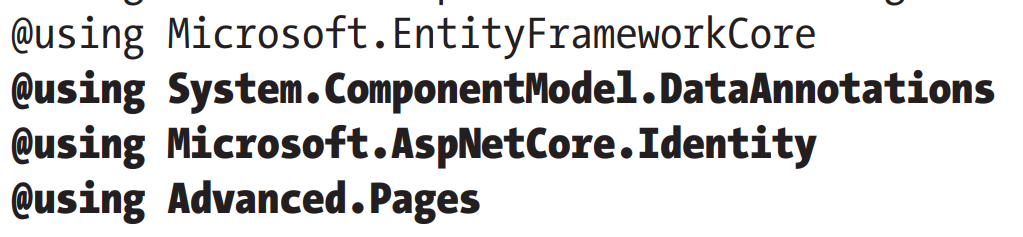
Если вы пишите приложение, где пользователи управляют своими собственными реквизитами для входа, тогда опции scaffolding возможно стоит продумать, как описано в https://docs.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity. Для всех остальных подходов следует использовать API управления пользователями, предоставляемый ASP.NET Core Identity.

Члены класса UserManager<T>, которые я использовал в этой секции для управления пользователями:

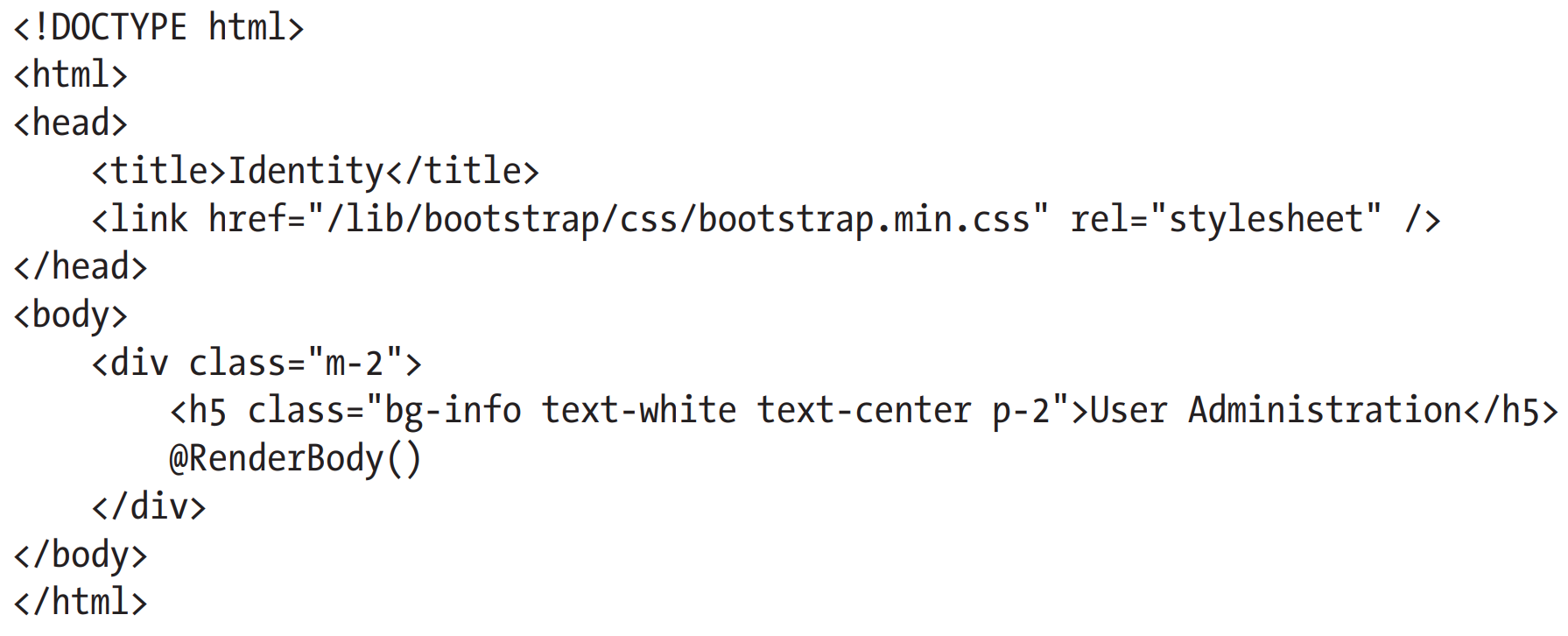


**Подготовка для инструментов управления пользователями**

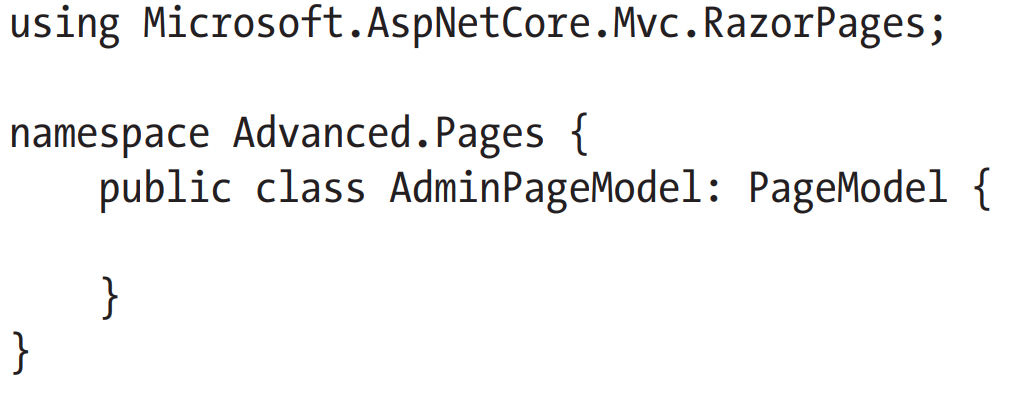
Готовясь к созданию инструментов управления, добавим выражения в файл \_ViewImports.cshtml в папке Pages проекта Advanced:



Далее создадим папку Pages/Users в проекте Advanced и добавим в нее Razor Layout с именем \_Layout.cshtml:



Добавим класс AdminPageModel.cs в папку Pages:



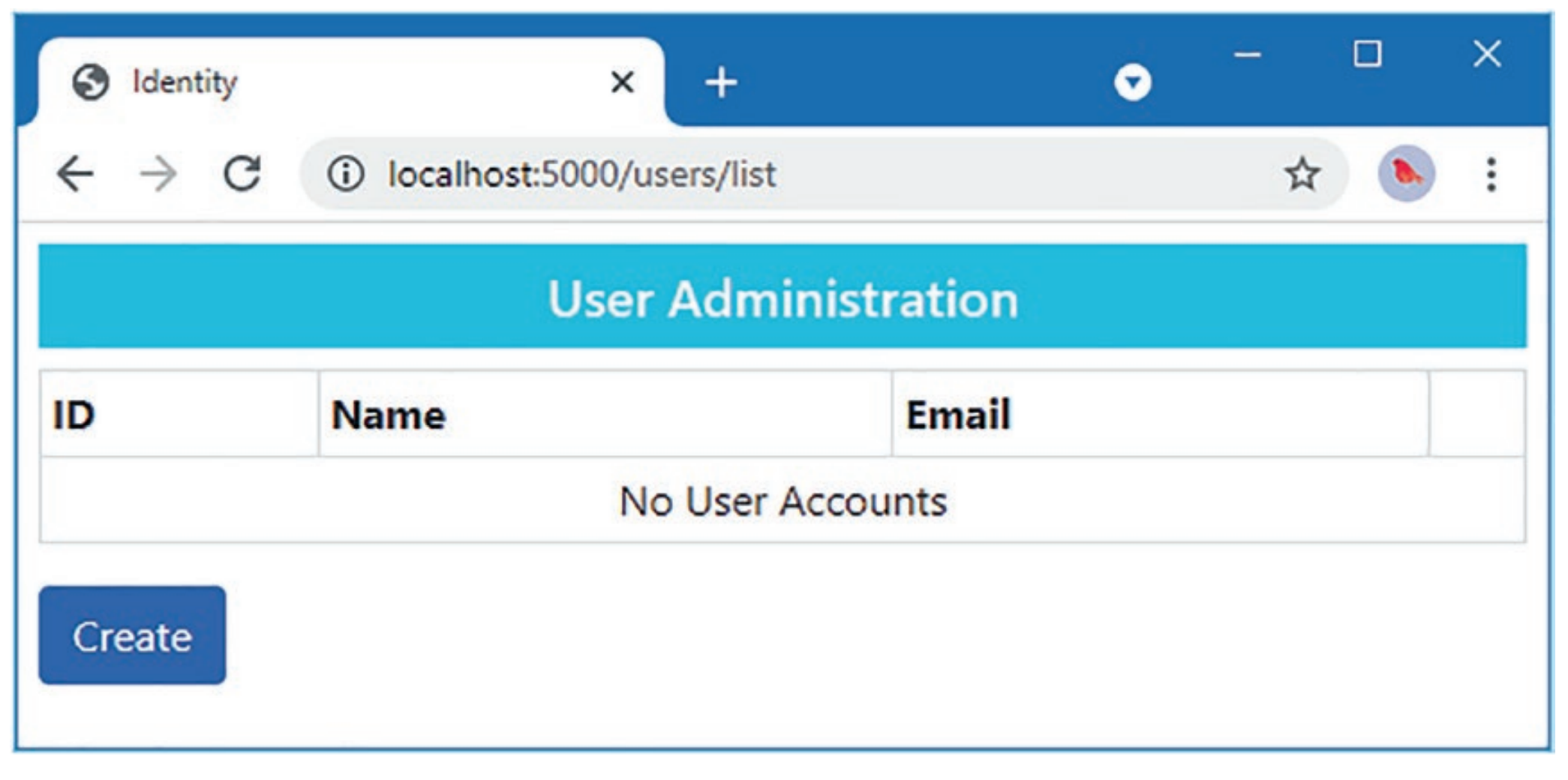
Этот класс будет базовым для классов моделей страниц, определенных в этой секции. Как вы увидите в главе 39, общий базовый класс полезен, когда речь идет о защите приложения.

**Перечисление учетных записей пользователей**

Хотя БД в данный момент пуста, я собираюсь начать с создания Razor Page которая будет перечислять учетные записи пользователей. Добавим Razor Page с именем List.cshtml в папку Pages/Users проекта Advanced.

Класс UserManager<IdentityUser> устанавливается сервисом, чтобы его можно было потреблять через DI. Свойство Users возвращает коллекцию объектов IdentityUser, которые можно использовать для перечисления учетных записей пользователей. Эта Razor Page отображает пользователей в таблице с кнопкой, позволяющей удалить или редактировать каждого пользователя, хотя она не будет видна изначально, потому что сообщение заместитель демонстрируется, когда нет объектов для отображения. Имеется кнопка, направляющая на Razor Page с именем Create, которую я создам в следующей секции.

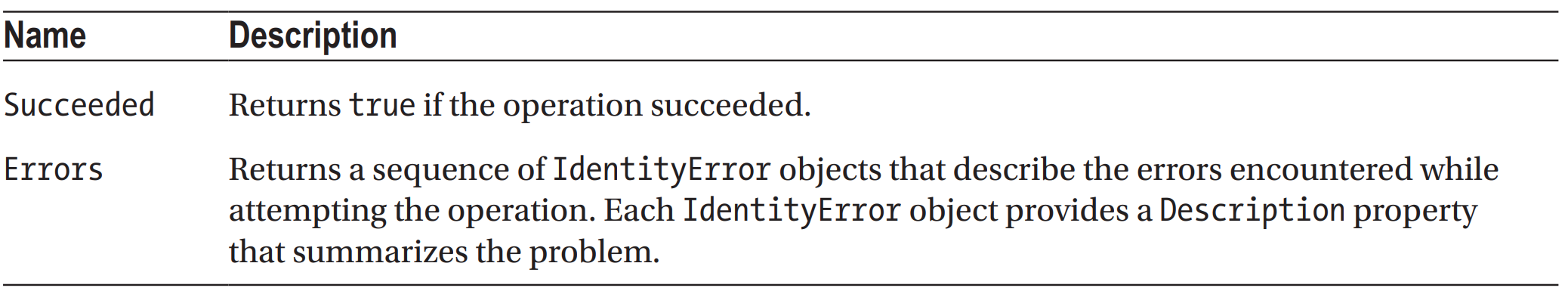
Перезапустим приложение и перейдем по URL http://localhost:5000/users/list



**Создание пользователей**

Добавим Razor Page с именем Create.cshtml в папку Pages/Users.

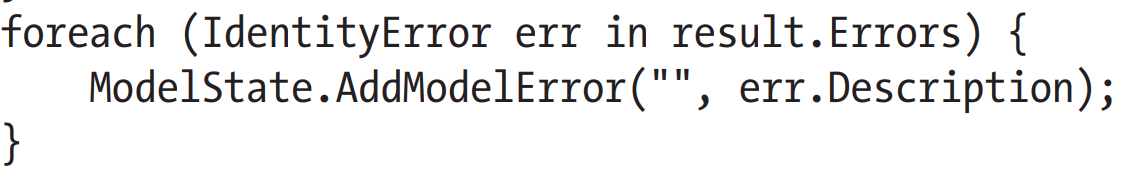
Несмотря на то, что данные ASP.NET Core Identity хранятся с использованием Entity Framework Core, ты не работаешь непосредственно с классом контекста базы данных. Вместо этого данными управляем через методы, предоставляемые классом UserManager<T>. Новый пользователь создается, используя метод CreateAsync, принимающий объект IdentityUser и строку пароля как аргументы. Данная Razor Page определяет три свойства, подлежащие привязке модели. Свойства UserName и Email используются для настройки объекта IdentityUser, сочетающиеся со значением, привязанным к свойству Password для вызова метода CreateAsync. Эти свойства конфигурируются с атрибутами валидации и значения потребуются, потому что типы свойств не могут принимать значения NULL. Результатом метода CreateAsync является объект Task<IdentityResult>, который указывает на результат операции создания, используя свойства из таблицы:



Я проверяю свойство Succeeded, чтобы определить, был ли создан новый пользователь в базе данных. Если данное свойство true, тогда клиент перенаправляется на страницу List, так, что список пользователь отображается с вновь введенной записью:



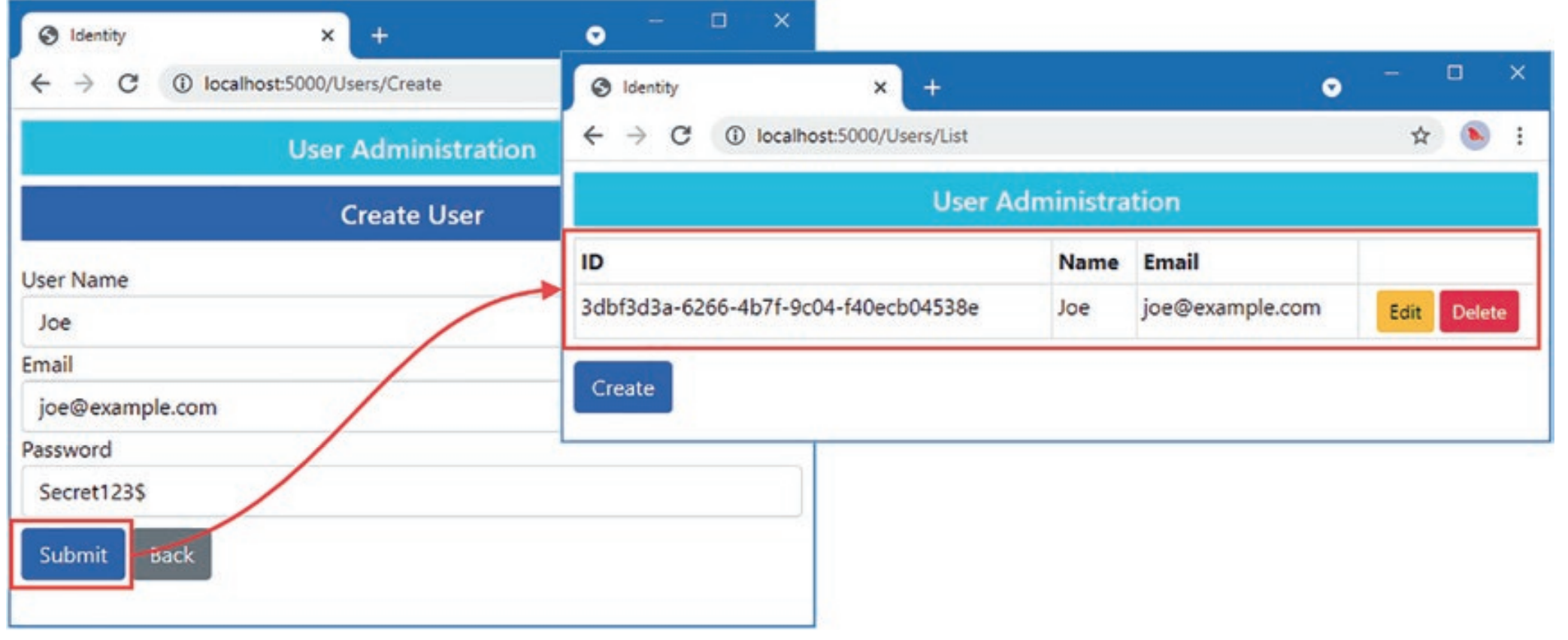
Если свойство Succeeded имеет значение false, тогда последовательность объектов IdentityError предоставляется как свойство Errors с перечислением свойств Description, используемых для создания валидации ошибок уровня модели, используя метод ModelState.AddModelError:



Для теста возможности создания нового пользователя, перезапустим приложение, перейдем на URL http:// localhost:5000/users/list. Кликнем по кнопке Create и заполним форму со значениями из таблицы:

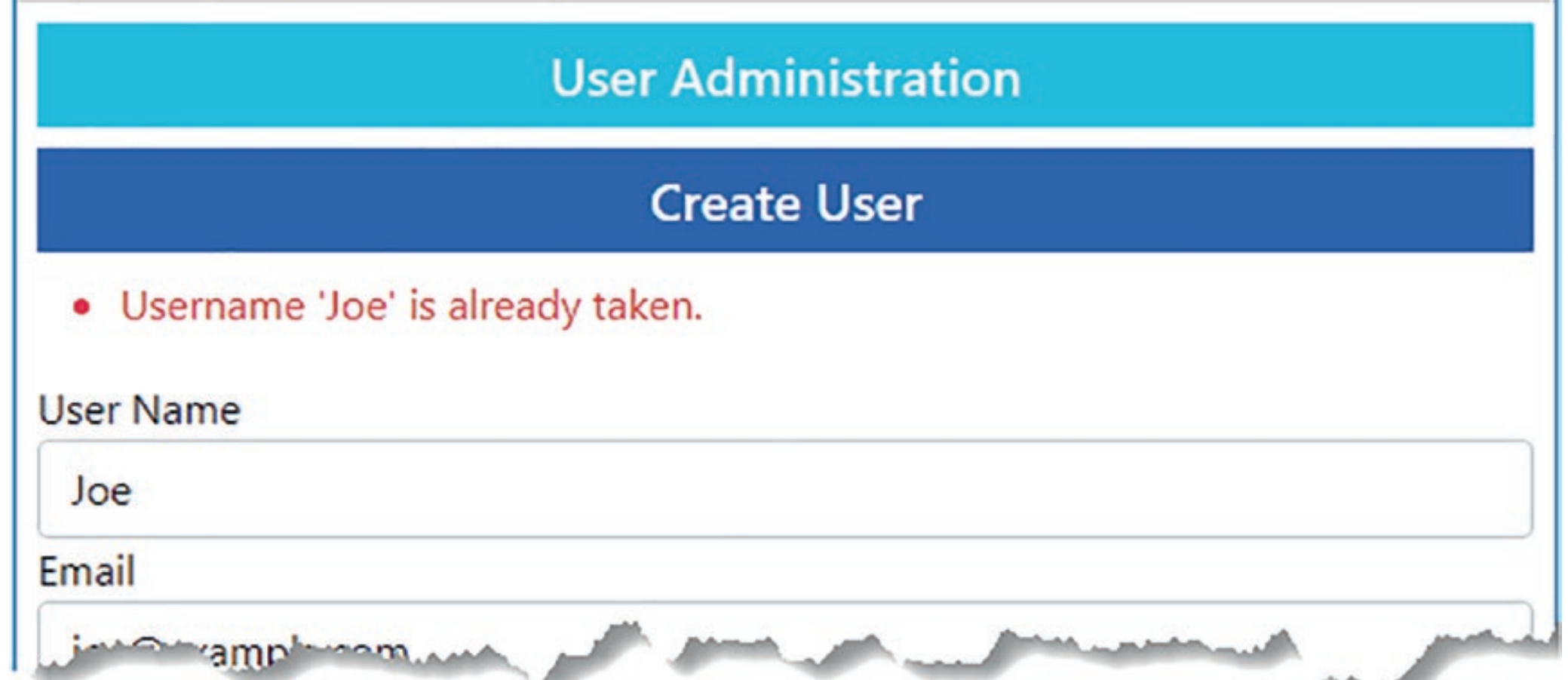


После заполнения формы кликнем по кнопке Submit. ASP.NET Core Identity создаст пользователя в БД и браузер будет перенаправлен обратно на список клиентов с внесенной новой записью:



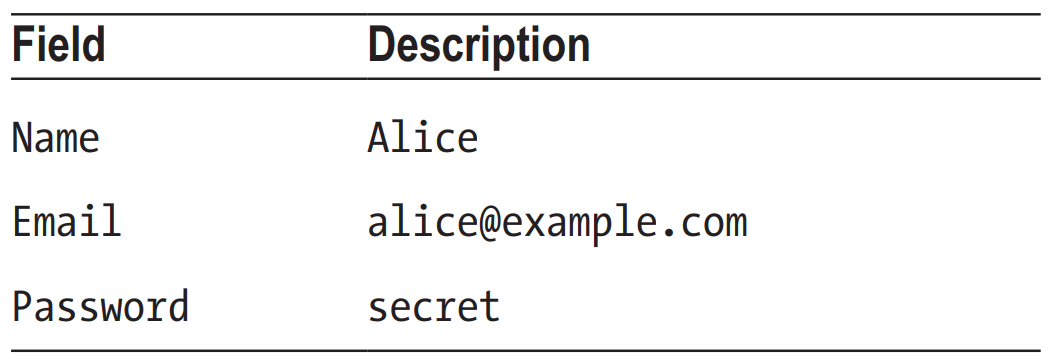
**Замечание**: я использовал регулярный элемент input для поля Password, чтобы облегчить следование примеру. На реальном проекте данное поле следует декорировать атрибутом password, с помощью которого набираемый пароль будет скрыт.

Кликнем по кнопке Create снова и заполним поля теми же самыми значениями, что и в прошлый раз. В этот раз вы увидите ошибку, сообщенную через сводку валидации модели, когда вы нажимаете кнопку «Создать». Это пример ошибки, возвращаемой объектом IdentityResult, произведенным методом CreateAsync:

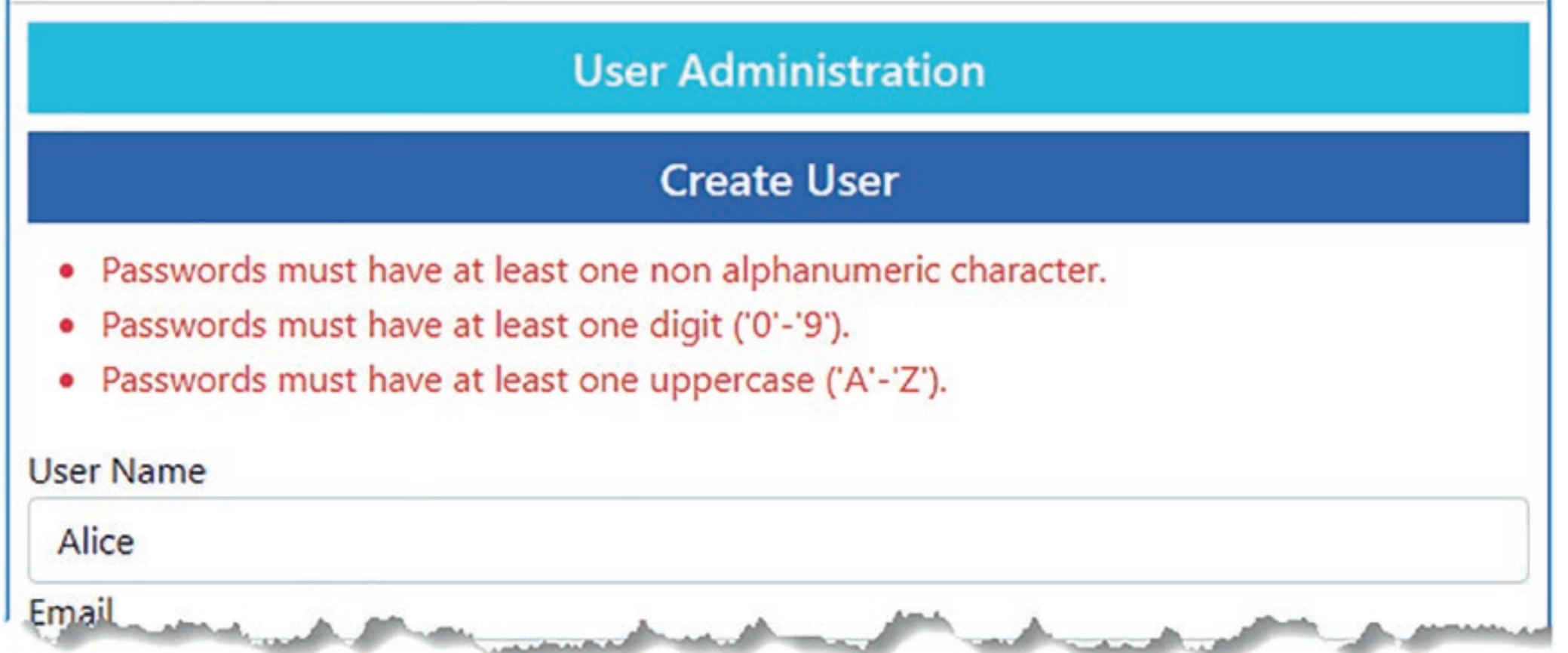


**Валидация пароля**

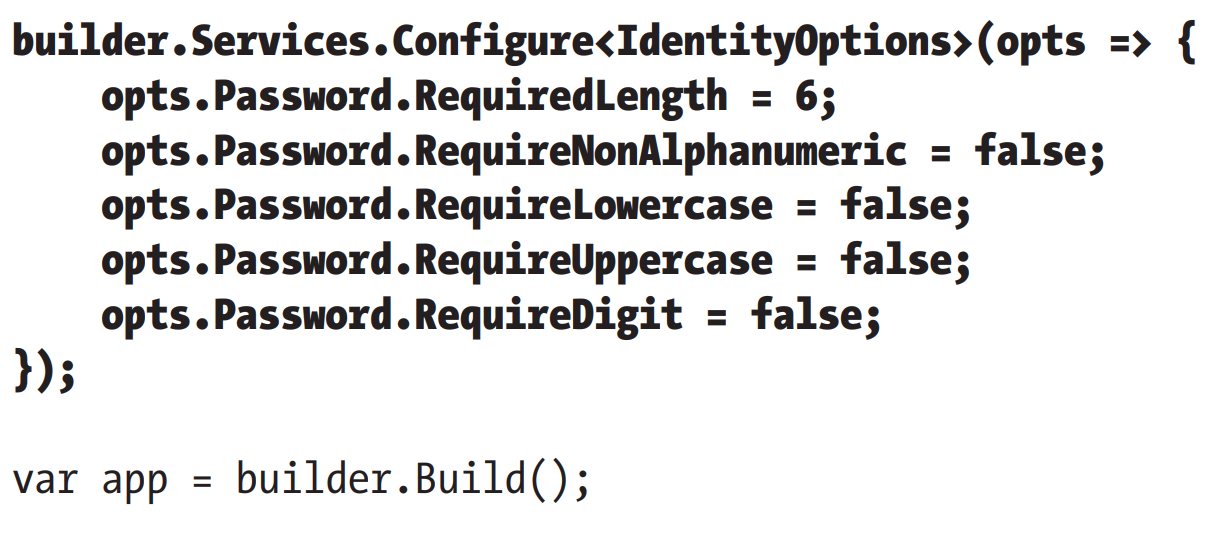
Одним из наиболее распространенных требований, особенно для корпоративных приложений, является принудительное использование политики пароля. Вы можете увидеть дефолтную политику, заполнив форму полями из таблицы:



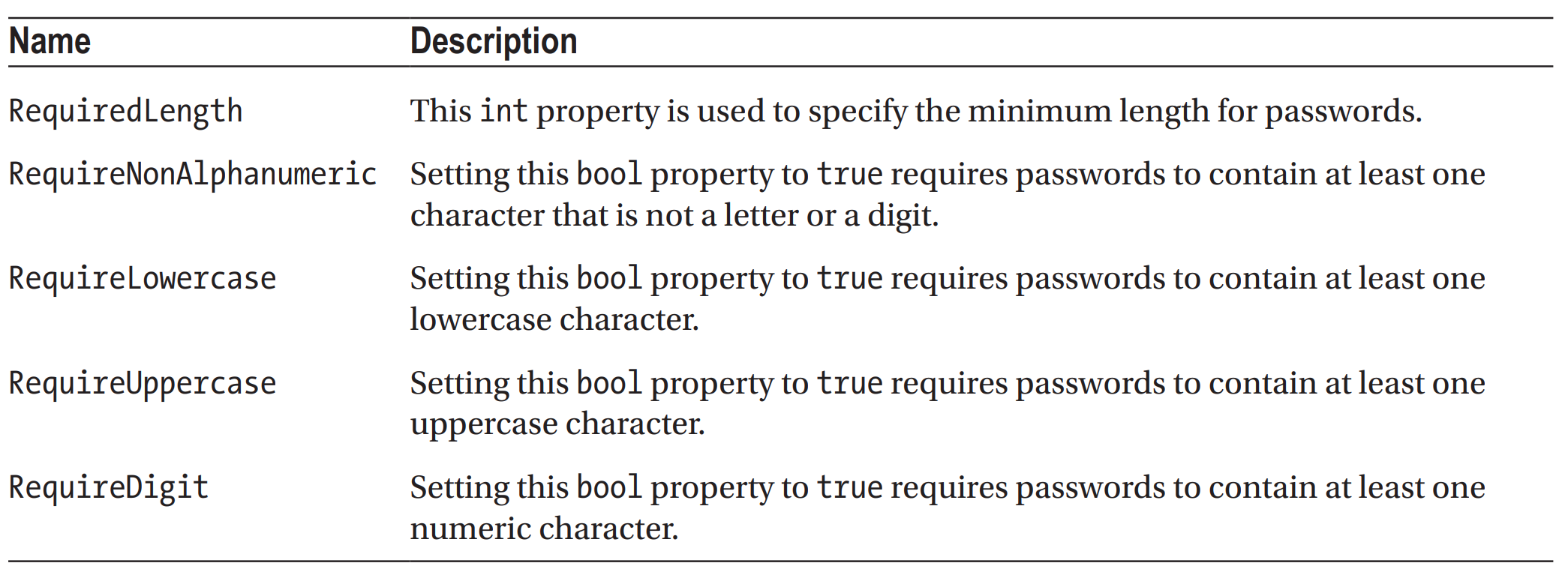
Когда вы отправите такую форму, ASP.NET Core Identity проверит кандидата в пароли и сгенерирует ошибку если он не пройдет соответствие паролям:



Правила валидации паролей настраиваются, используя паттерн options:



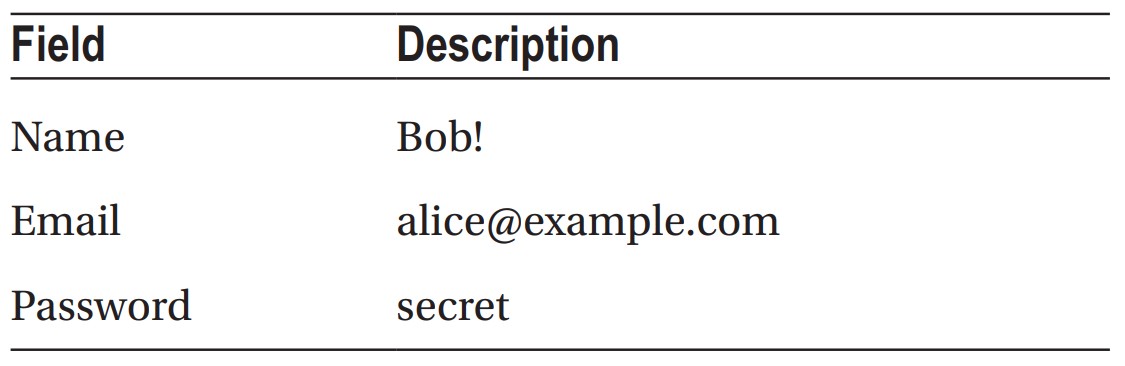
ASP.NET Core Identity настраивается класс IdentityOptions, чье свойство Password возвращает класс PasswordOptions, конфигурирующий валидацию пароля с помощью свойств:



В листинге я задал, чтобы пароль имел длину минимум 6 знаков и деактивировал другие ограничения. Это не то, что вы должны делать без тщательного рассмотрения в реальном проекте, но позволяет провести эффективную демонстрацию. Перезапустим приложение и проделаем те же шаги, что и в прошлый раз с паролем secret. При этом при клике по кнопке Submit валидация пароля проходит успешно с новыми правилами валидации, и новый пользователь будет создан.

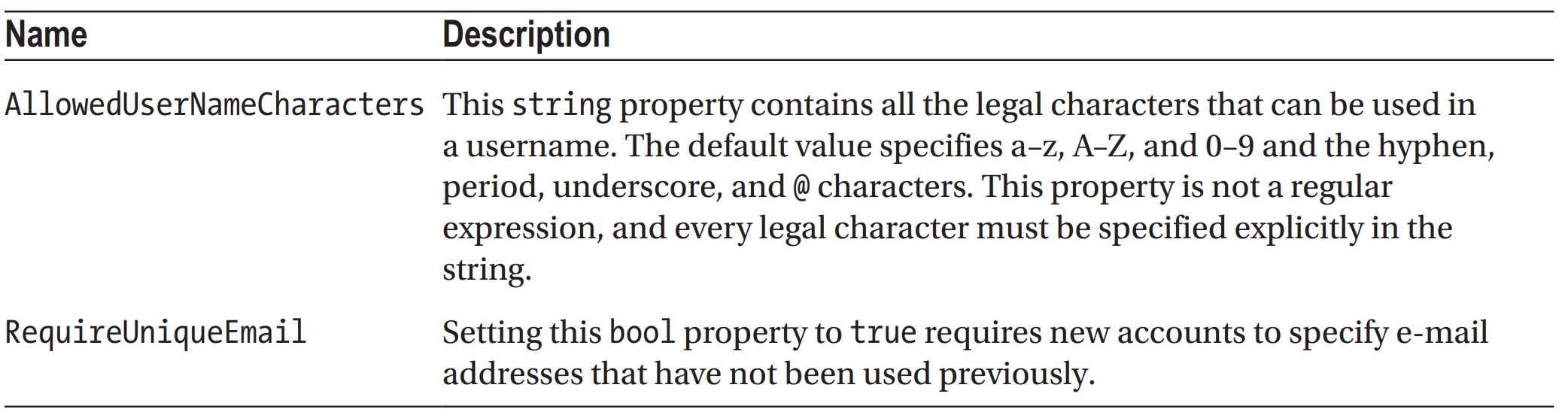
**Валидация полей пользователя**

Валидация полей имени пользователя и e-mail адреса также производится при создании аккаунта. Чтобы увидеть, как применяется валидация, перезапустим приложение и заполним форму создания клиента с такими данными:

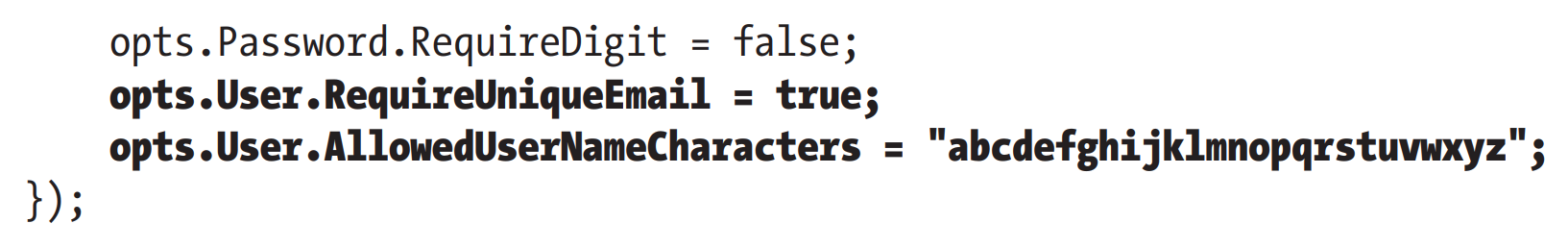




Валидация данных пользователя может быть настроена с использованием паттерна options, используя свойство User, определенное в классе IdentityOptions. Этот класс возвращает класс UserOptions, свойства которого описаны в таблице:



Изменим конфигурацию приложения так, чтобы требовался уникальный e-mail адрес и в имени пользователя были возможны только строчные буквы:

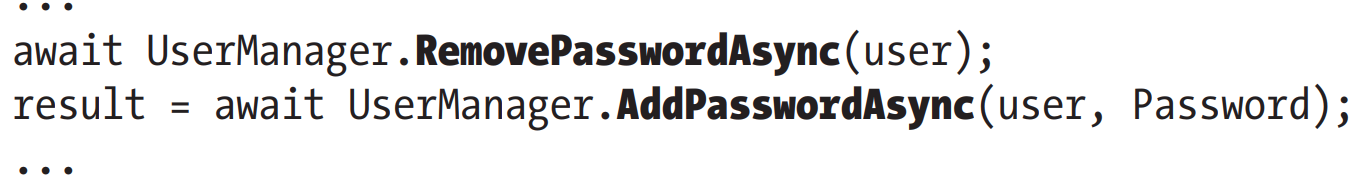


**Редактирование пользователей**

Для добавления поддержки редактирования пользователей, добавим Razor Page с именем Editor.cshtml в папку Pages/Users проекта Advanced.

Страница Editor использует метод UserManager<T>.FindByIdAsync, который находит пользователя, запрашивая БД по значению Id, полученным посредством системы роутинга, который передается как аргумент в метод OnGetAsync. Значение объекта IdentityUser, возвращаемое в результате запроса к БД, используется для заполнения свойств? которые отображаются вью частью страницы, гарантируя, что значения не будут потеряны, если страница отображается повторно из-за ошибок валидации.

Когда пользователь отправляет форму, метод FindByIdAsync используется для запроса к БД, чтобы получить объект IdentityUser, который обновляется значениями свойств UserName и Email из формы. Пароль требует другого подхода и должен быть удален из объекта user перед установлением нового пароля:

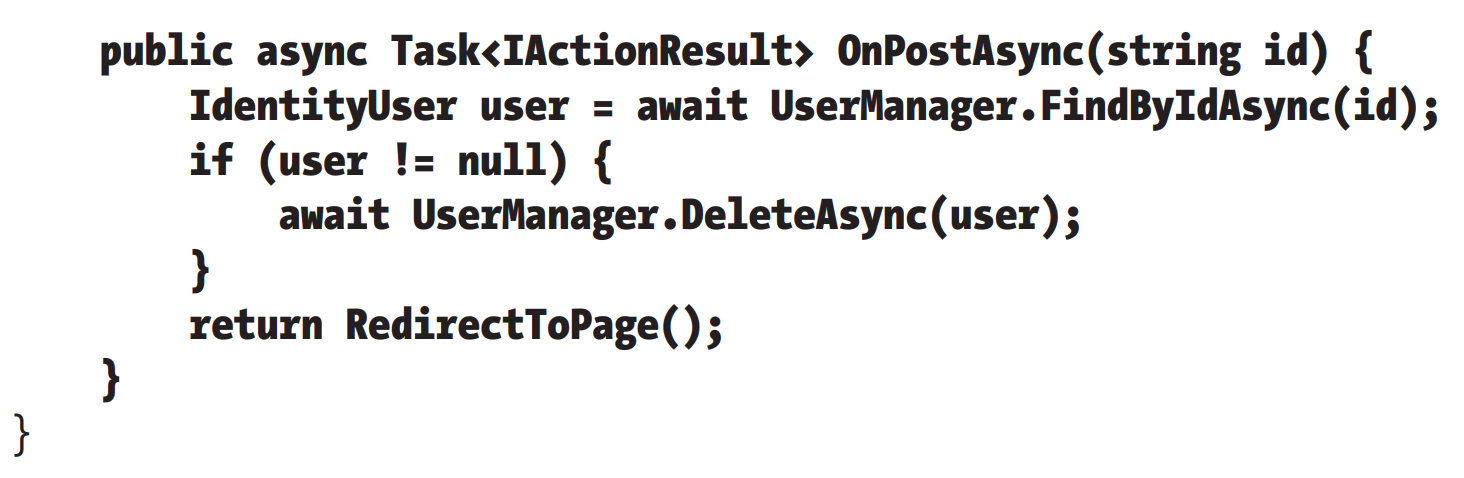


Страница Editor меняет пароль только если форма имеет значение поля Password и если обновления имени пользователя и Email были успешны. Ошибки из ASP.NET Core Identity представляются как сообщения валидации, и браузер перенаправится на страницу List после успешного обновления. Перейдите по URL http://localhost:5000/Users/List, кликните по кнопке Edit объекта Joe и измените имя со всеми строчными буквами. Кликните на кнопку Submit и вы увидите изменение, отображенное в списке пользователей.

**Удаление пользователей**

Последняя функциональность необходимая для базового управления пользователями приложения это возможность удаления пользователей.

Добавим такую функциональность в файл List.cshtml:



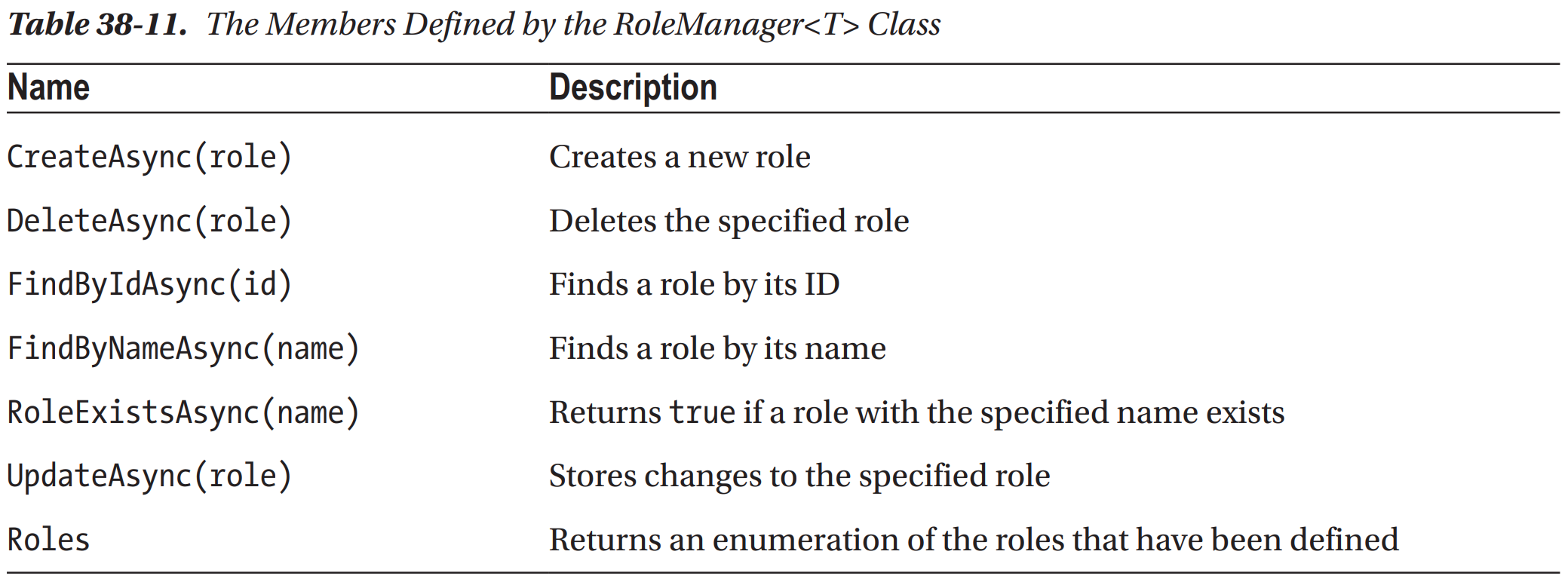
Страница List уже содержит кнопку Delete для каждого пользователя в таблице, которая отправляет запрос POST, содержащий Id для объекта IdentityUser, который должен быть удален. Метод OnPostAsync получает значение Id как параметр и использует его для запроса объекта IdentityUser, используя метод FindByIdAsync и передает его в метод DeleteAsync, который удаляет объект из БД. Для проверки функциональности удаления, удалим один из аккаунтов.

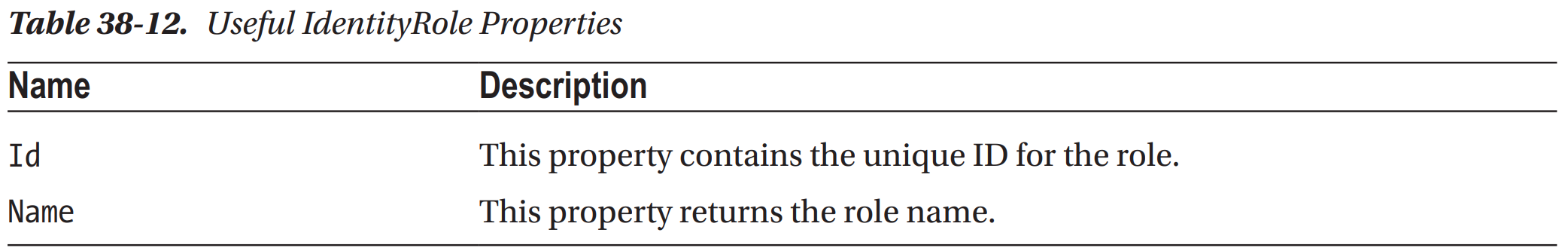
**Создание инструментов управления ролями**

Некоторые приложения обеспечивают соблюдение только двух уровней авторизации: аутентифицированным пользователям позволен доступ ко всей функциональности приложения, а не аутентифицированные пользователи имеют ограниченный доступ или не имеют вообще. Приложение SportsStore в главе 1 следовал такому подходу: там был один пользователь, и один раз аутентифицировавшись, он имел доступ ко всей функциональности приложения, включая инструменты администратора, в то время как не аутентифицированные пользователи ограничены публичной функциональностью магазина.

ASP.NET Core Identity поддерживает роли для приложения, требующие более раздробленной авторизации. Пользователи, которым назначены одна или более ролей и их принадлежность к этим ролям определяет, какие функции доступный. В следующей секции я покажу, как построить инструменты для создания и управления ролями.

Роли управляются посредством класса RoleManager<T> , где Т это представление роли в БД. Когда я настраивал ASP.NET Core Identity в начале главы, я выбрал предустановленный класс IdentityRole, предоставляемый ASP.NET Core Identity для описания роли, что означает, что я буду использовать класс RoleManager< IdentityRole > в этом примере. Класс RoleManager<T> определяет методы и свойства, позволяющие создавать роли и управлять ими:



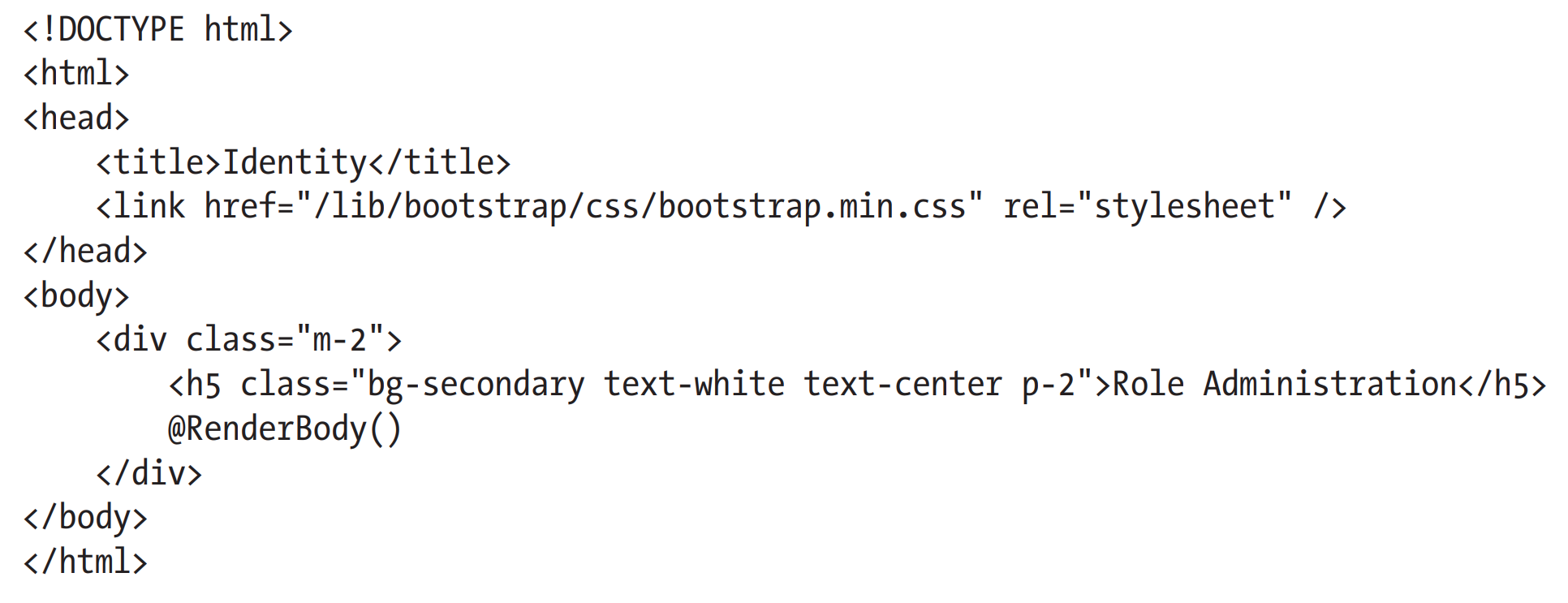


Хотя управление ролями осуществляется через класс RoleManager<T>, отношения ролей управляются через методы, предоставляемые классом UserManager<T>, описанные в таблице:



**Подготовка для использования инструментов управления ролями**

Создадим папку Pages/Roles и добавим в нее файл лейаута \_Layout.cshtml:



Этот лейаут будет гарантировать очевидную разницу между инструментами управления user и role.

**Перечисление и удаление ролей**

Добавим Razor Page с наименованием List.cshtml в папку Pages/Roles с контентом, содержание которого можно найти в проекте. Роли перечисляются вместе с именами до трех членов роли или заполняющим сообщением, если нет членов. Также имеется кнопка Create и каждая роль представлена с кнопками Edit и Delete, следуя тому же шаблону, который я использовал в инструменте управления пользователями. Кнопка Delete отправляет запрос POST к Razor Page. Метод OnPostAsync использует метод FindByIdAsync для получения объекта роли, который передается в метод DeleteAsync для удаления его из БД.

**Создание роли**

Добавим Razor Page с именем Create.cshtml в папку Pages/Roles, содержание которого можно найти в проекте.

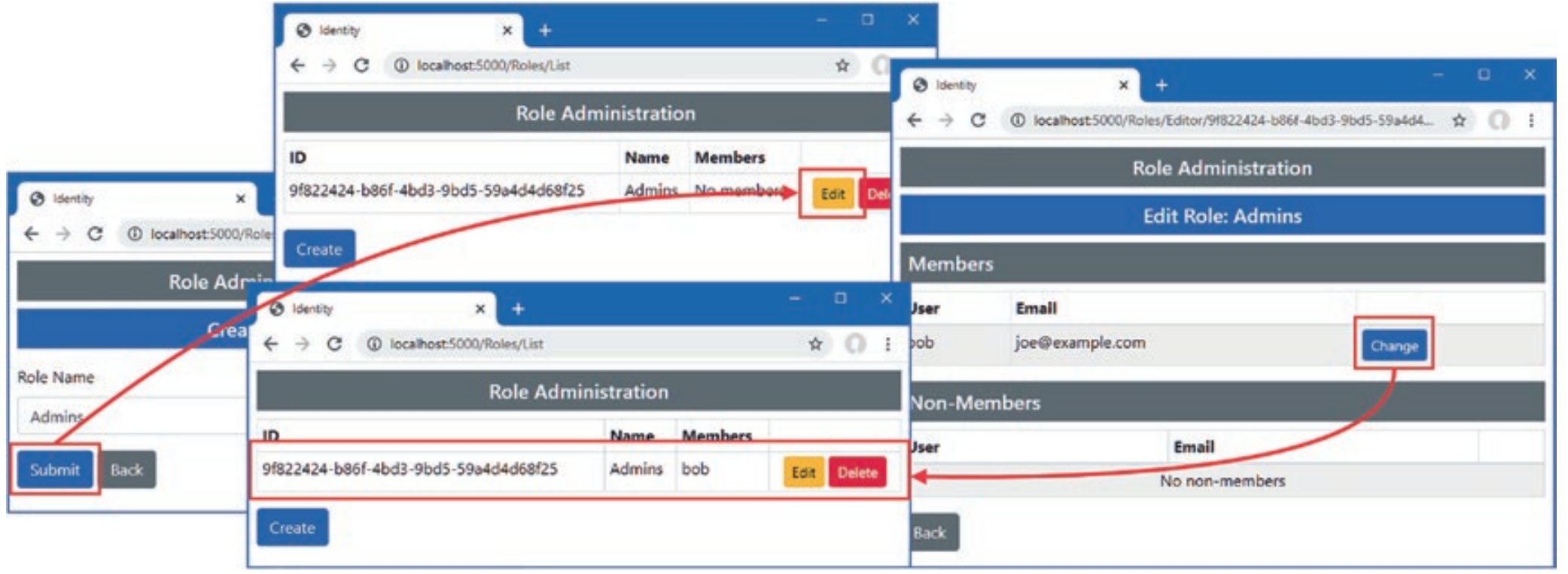
Пользователь представлен с формой, содержащей элемент input для задания имени роли. Когда форма отправлена, метод OnPostAsync создает новый объект IdentityRole и передает его в метод CreateAsync.

**Назначение членов роли**

Для добавления поддержки управления членов роли, добавим Razor Page с именем Editor.cshtml в папку Pages/ Roles, содержание которого можно найти в проекте.

Пользователю предоставляется таблица, показывающая пользователей, которые являются членами роли, и таблица, показывающая не членов данной роли. Каждая строка содержит кнопку Change, которая отправляет форму. Метод OnPostAsync использует метод UserManager.FindByIdAsync для получения объекта пользователя из БД. Метод IsInRoleAsync использован для определения является ли пользователь членом роли, а методы AddToRoleAsync и RemoveFromRoleAsync используются для добавления и удаления пользователя, соответственно.

Перезапустим приложение и перейдем на URL <http://localhost:5000/roles/list>. Список будет пуст, потому что еще нет ролей в БД. Кликнем по кнопке Create, введем Admins в текстовое поле и кликнем по кнопке Submit для создания новой роли. Когда роль будет создана, кликнем по кнопке Edit и вы увидите список пользователей, которые могут быть добавлены к данной роли. Клик по кнопке Change переместит пользователя из одной таблицы в другую. Кликнем по кнопке back и список будет обновлен для показа пользователей, являющихся членами данной роли:



**Внимание**:  ASP.NET Core Identity перепроверяет данные пользователя при изменении назначений ролей, что приведет в ошибке, если вы попытаетесь изменить пользователя, данные которого не соответствуют текущим ограничениям, что происходит, когда ограничения представлены после того, как приложение было развернуто и БД уже заполнена пользователями, созданными под старыми ролями. Именно по этой причине Razor Page в файле Editor.cshtml проверяет результат из операций по добавлению или удалению пользователей из роли и отображает любые ошибки в виде сообщений валидации.

**­­­­­­­­­­­­­­­­­Итог**

В этой главе я показал, как добавлять ASP.NET Core Identity в проект и приготовил БД для пользователей и ролей. Я описал базовый API ASP.NET Core Identity и показал, как она может использоваться для создания инструментов управления пользователями и ролями. В следующей главе я покажу как применять ASP.NET Core Identity для контроля доступа в контроллерах, Razor Pages, Blazor и web сервисах.

UZ4J@V!yxF7m?hN