**Глава 37**

**Использование Blazor Web Assembly**

В этой главе я продемонстрирую использование Blazor Web Assembly, являющейся реализацией Blazor, написанной для Web Assembly.

WebAssembly это виртуальная машина, запускаемая внутри браузера. Высокоуровневый язык компилируется в низкоуровневый независимый от языка формат ассемблера, который может выполняться с производительностью, близкой к собственной.

WebAssembly предоставляет доступ к API, доступной приложениям JavaScript, что означает, что WebAssembly приложение имеет доступ к DOM, использующий CSS и инициализирующий асинхронные HTTP запросы.

Blazor WebAssembly, как следует из названия, это реализация Blazor, запускаемая в виртуальной машине WebAssembly. Blazor WebAssembly убирает зависимость от сервера и выполняет приложение Blazor в браузере. Результатом является действительное клиент-серверное приложение, которое имеет доступ ко всей функциональности Blazor Server, но без необходимости в постоянном HTTP соединении.

Это ранние дни для обоих WebAssembly и Blazor WebAssembly и имеется несколько серьезных ограничений. WebAssembly это новая технология и поддерживается только последними версиями браузеров. Вам может быть не доступно использование WebAssembly если ваш проект должен поддерживать старые браузеры или даже старые версии современных браузеров. Blazor WebAssembly приложения набором API, предоставляемых браузером, это означает, что не вся функциональность .NET может быть использована в WebAssembly приложении. Это не недостаток Blazor в сравнении с фронт-энд фреймворками, но это означает, что такие функции, как Entity Framework Core недоступны, поскольку браузеры ограничивают приложения WebAssembly выполнением HTTP-запросов.

Тем не менее, несмотря на ограничения Blazor WebAssembly, это превосходная технология и предлагает обещание возможности писать настоящие клиентские приложения с использованием C# и ASP.NET Core без необходимости в JavaScript фреймворках.

**Подготовка к данной главе**

В этой главе используется приложение Advanced. Добавим класс контроллера DataController.cs в папку Controllers и используем его для определения web сервиса.

Этот контроллер предоставляет методы action, позволяющие создавать, читать, редактировать и удалять объекты Person. Я также добавил методы action, возвращающие объекты Location и Department. Я обычно создаю отдельные контроллеры для каждого типа данных, но эти методы action необходимы только для поддержки функциональности Person, поэтому я скомбинировал все операции в единственном контроллере.

Запустим приложение и перейдем по URL <http://localhost:5000/api/people>, который произведет представление в формате JSON объектов Person в БД.

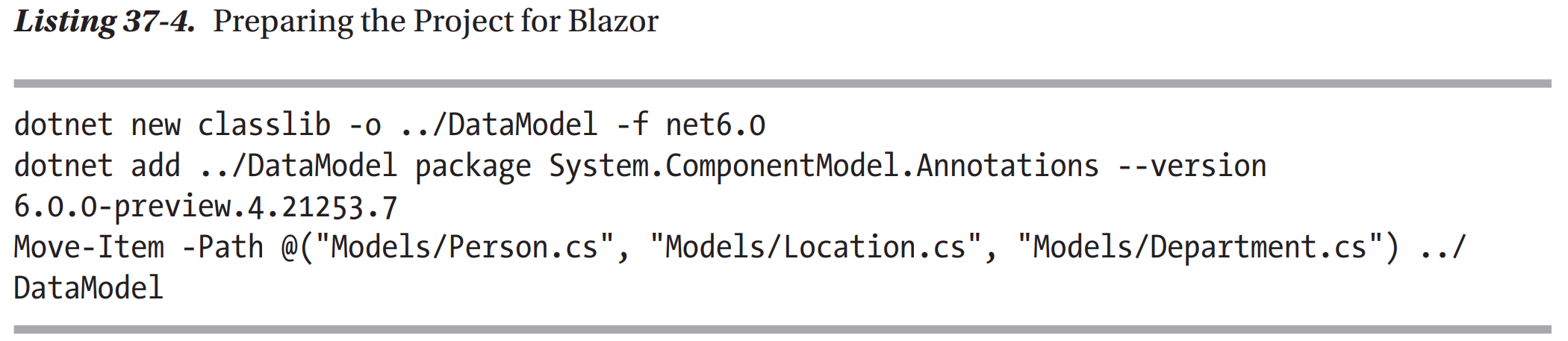
**Настройка Blazor WebAssembly**

Blazor WebAssembly требуется отдельный проект, так, чтобы компоненты Razor могли быть скомпилированы и в готовом виде выполняется браузером. Скомпилированные компоненты могут быть доставлены в браузер с помощью стандартного сервера ASP.NET Core, который также может предоставлять данные с помощью web сервисов. Чтобы облегчить работу с компонентами Blazor WebAssembly для использования данных, предоставляемых сервером ASP.NET Core, требуется третий проект, который содержит те элементы, которые являются общими между ними.

Процесс создания трех проектов сложен, отчасти потому, что я собираюсь переместить некоторые из существующих классов из проекта Advanced в проект модели данных. Хотя можно выполнить некоторые шаги с помощью мастеров Visual Studio, я описал шаги используя командную строку, чтобы минимизировать ошибки.

**Создание общего проекта**

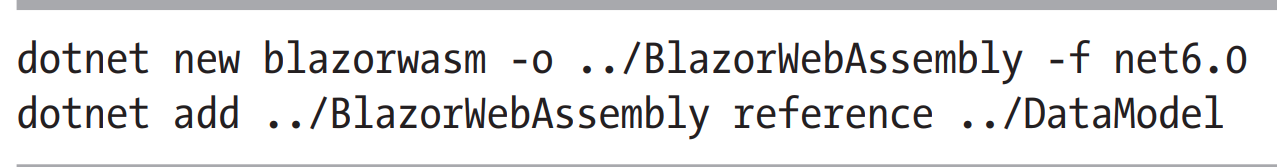
Удостоверьтесь, что Visual Studio или Visual Studio Code закрыты перед тем, как начать. Откройте новую командную строку PowerShell и переместитесь в папку Advanced, которая содержит файл Advanced.csproj и запустите команды из листинга:



Эти команды создают новый проект с именем DataModel, устанавливают в него пакет System.ComponentModel. Annotations (содержащий атрибуты используемые для валидации данных) и перемещают классы моделей данных в новый проект.

**Создание проекта Blazor WebAssembly**

Я обычно предпочитаю начинать с пустого проекта и добавлять пакеты и конфигурационные файлы, требуемые в проекте. Используем командную строку PowerShell, чтобы запустить команды из листинга, находясь в папке Advanced:



Эти команды создают проект Blazor WebAssembly с именем BlazorWebAssembly и добавляют ссылку на проект DataModel, делающую доступными классы Person, Department, и Location.

**Подготовка проекта ASP.NET Core**

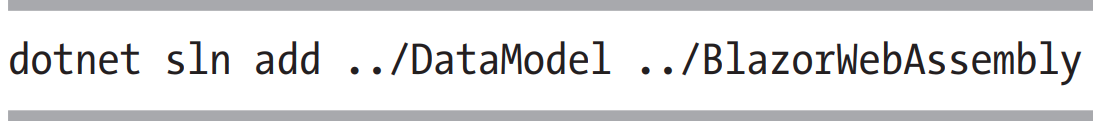
Используем командную строку PowerShell для запуска команд из листинга в папке Advanced:



Эти команды создают ссылки на другие проекты, так, что классы моделей данных и компоненты проекта Blazor WebAssembly могут быть использованы.

**Добавление ссылок на решение**

Запустим команды из листинга в папке Advanced, чтобы добавить ссылки на новый проект в файл решения:

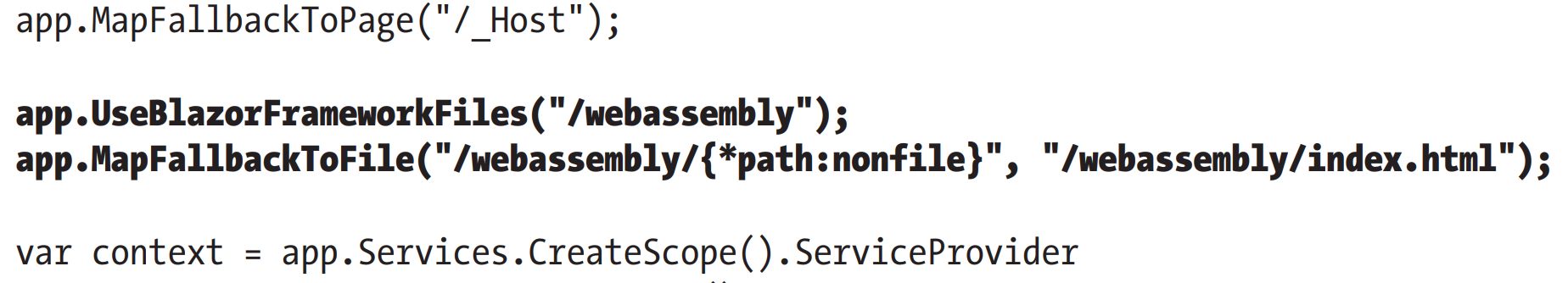


**Откроем проект**

Как только вы установили все три проекта, запустите Visual Studio, откройте файл Advanced.sln. Все три проекта будут открыты для редактирования.

**Завершение настройки Blazor WebAssembly**

Следующий шаг настройки проекта ASP.NET Core так, чтобы он мог доставлять контент проекта Blazor WebAssembly клиенту. Добавьте указанные операторы в файл Program.cs в папке Advanced:



**Осторожно**: важно обращать особое внимание какой файл вы изменяете. Файлы с такими же именами используются в множестве проектов, и если вы тщательно не следуете примерам, вы не закончите с работающим приложением. Будущие версии Blazor могут быть проще в работе, но на данный момент важны детали.

Эти операторы настраивают пайплайн запросов ASP.NET Core так, что запросы для /webassembly обрабатываются Blazor WebAssembly, используя контент проекта BlazorWebAssembly.

**Установка базового URL**

Следующим шагом будет изменение файла HTML, который будет использоваться для ответа на запрос для URL /webassembly. Применим изменения показанные в листинге для файла index.html в папке wwwroot проекта BlazorWebAssembly:



Элемент base устанавливает URL адрес, от которого определяются все относительные URLs в документе, он требуется для корректной работы системы маршрутизации Blazor WebAssembly.

**Установка StaticWebAssetBasePath**

Отредактируем файл проекта BlazorWebAssembly



Контент тега StaticWebAssetBasePath должен быть /webassembly/ с косыми чертами вначале и в конце.

**Осторожно**: Убедитесь, что перед и после webassembly есть символы косой черты (/) в атрибуте StaticWebAssetBasePath базового элемента. Если вы пропустите какой-либо символ, Blazor WebAssembly не будет работать.

**Тестирование компонента заполнителя**

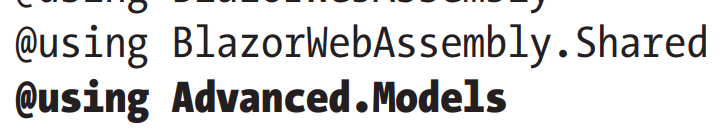
Запустим ASP.NET Core проект Advanced и перейдем на URL <http://localhost:5000/webassembly> и вы увидите контент заместитель, добавленный в шаблоне при создании проекта BlazorWebAssembly.

**Создание компонента Blazor WebAssembly**

Blazor WebAssembly использует тот же подход, что и Blazor Server, полагаясь на компоненты как на строительные блоки приложения, соединенные через систему роутинга и отображая общий контент посредством лейаутов. В данной секции я покажу как создать компонент Razor, работающий с Blazor WebAssembly, и затем, я пересоздам простую форму приложения из главы 36.

**Импортирование пространства имен модели данных**

Компоненты, которые я использую в этой главе, используют классы из общего проекта DataModel. Вместо того, чтобы добавлять выражения @using в каждый компонент, добавим пространство имен для классов моделей данных в файл \_Imports. Razor в корневой папке проекта BlazorWebAssembly:



Имейте в виду, хотя я перенес файлы классов моделей в проект DataModel, я задал пространство имен Advanced.Models. Это потому, что в классах, файлы которых я перенес, namespace заданны Advanced.Models. Это означает, что перемещение файлов не изменяет пространство имен, определенные в классах.

**Создание компонента**

В предыдущих главах, я определял свои компоненты Razor в папке Blazor, чтобы держать новый контент отдельно от других частей ASP.NET Core. В проекте BlazorWebAssembly имеется только Blazor контент, поэтому я собираюсь следовать конвенции, принятой в шаблоне проекта и использовать папки Pages и Shared.

Добавим компонент Razor с наименованием List.razor в папку Pages проекта BlazorWebAssembly.

Если вы сравните этот компонент с Blazor Server из главы 36, вы увидите, что они по большей части одинаковы. Оба типа Blazor используют одинаковый набор базовой функциональности, именно поэтому контент использует те же директивы Razor, обработчики событий с атрибутами @onclick и использует секцию @code для кода C#. Компонент Blazor WebAssembly компилируется в класс C#, аналогично Blazor Server. Ключевое отличие конечно в том, что сгенерированный класс C# выполняется в браузере — и в этом причина отличий от компонента в главе 36.

**Навигация в компоненте Blazor WebAssembly**

Имейте в виду, что URLs, используемые для навигации, объявляются без начальной косой черты:



Корневая URL для приложения определена с использованием элемента base, и использование относительных URL-адресов гарантируют, что навигация выполняется относительно корня. В данном случае относительный URL forms/create дополняется /webassembly/ и навигация будет к /webassembly/forms/create. Включение начальной косой черты вместо этого приведет к /forms/create, находящийся находится за пределами набора URL-адресов, которыми управляет часть приложения Blazor WebAssembly. Это изменение требуется только для URL-адресов навигации. URL-адресов, указанных с помощью директивы @page, это не касается.

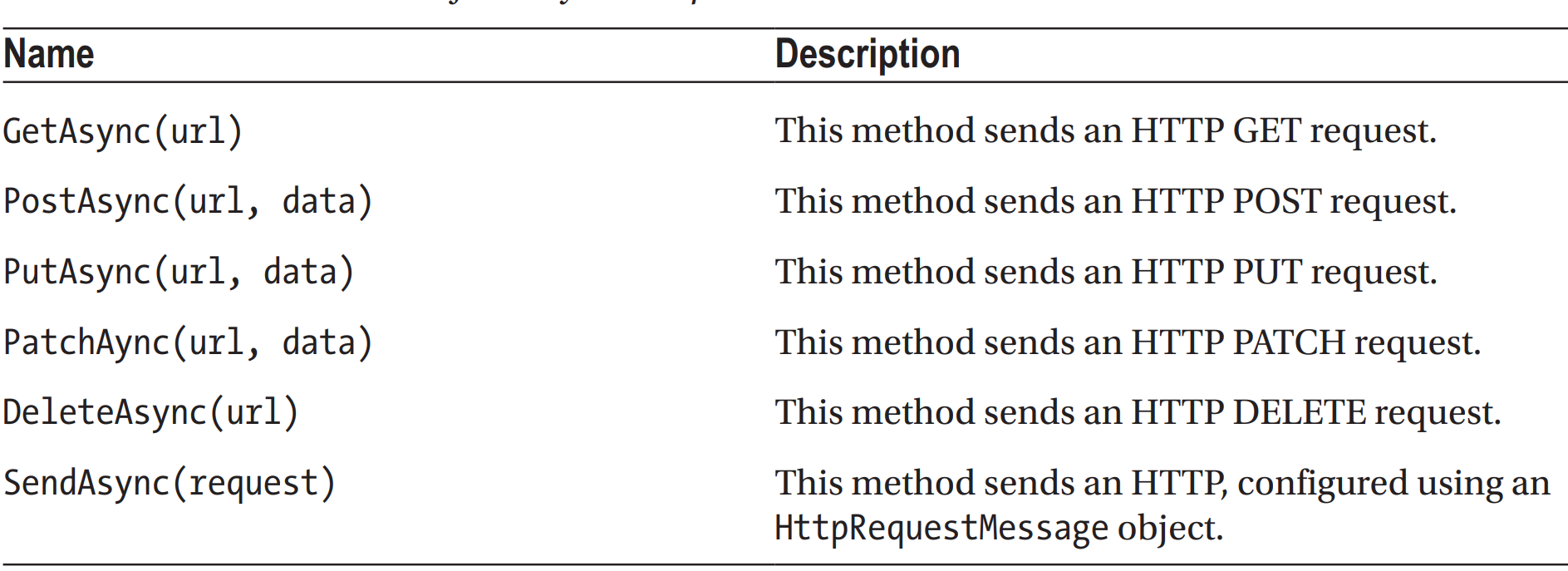
**Получение данных в компоненте Blazor WebAssembly**

Наибольшее различие Blazor WebAssembly в том, что нельзя использовать Entity Framework Core. Хотя runtime может быть способен выполнить классы Entity Framework Core, браузер ограничивает WebAssembly приложение HTTP запросами, предотвращая использование SQL. Для получения данных, приложение Blazor WebAssembly пользуется web сервисами, вот почему я добавил API контроллер в проект Advanced в начале главы.

В рамках запуска приложения Blazor WebAssembly, создается сервис класса HttpClient, компоненты которого можно получать с помощью стандартных функций DI. Компонент List получает компонент HttpClient через свойство, декорированное атрибутом Inject:

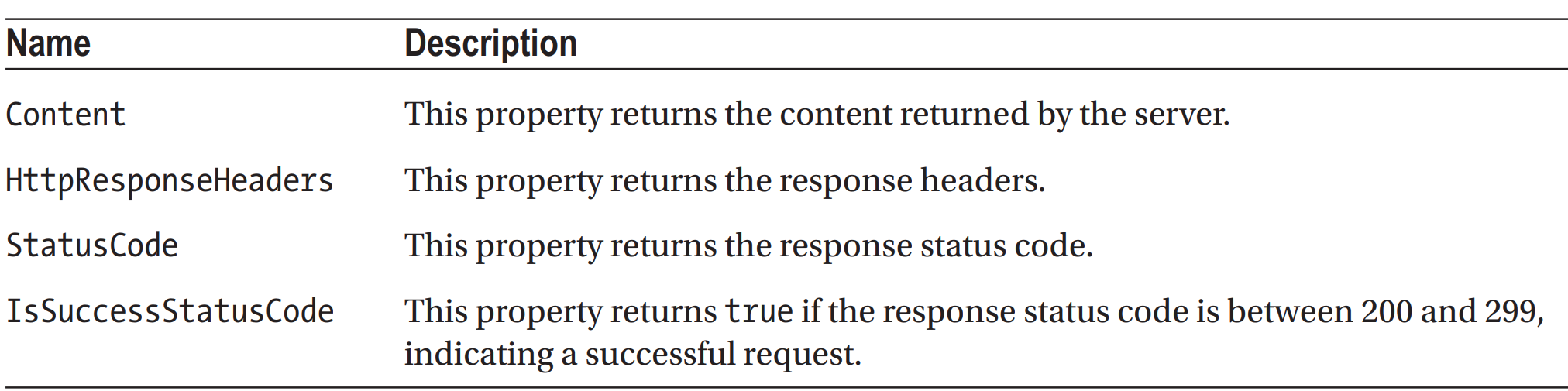


Класс HttpClient предоставляет следующие методы:

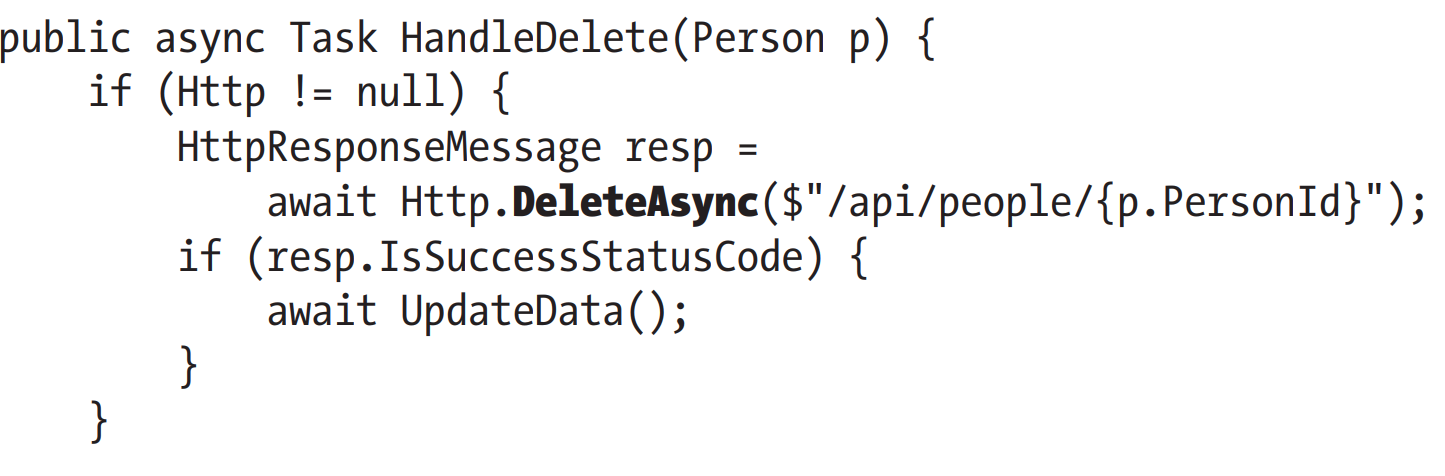


Данные методы возвращают Task< HttpResponseMessage >, описывающие ответ, полученный от HTTP-сервера на асинхронный запрос.

Самые полезные свойства HttpResponseMessage представлены в таблице:

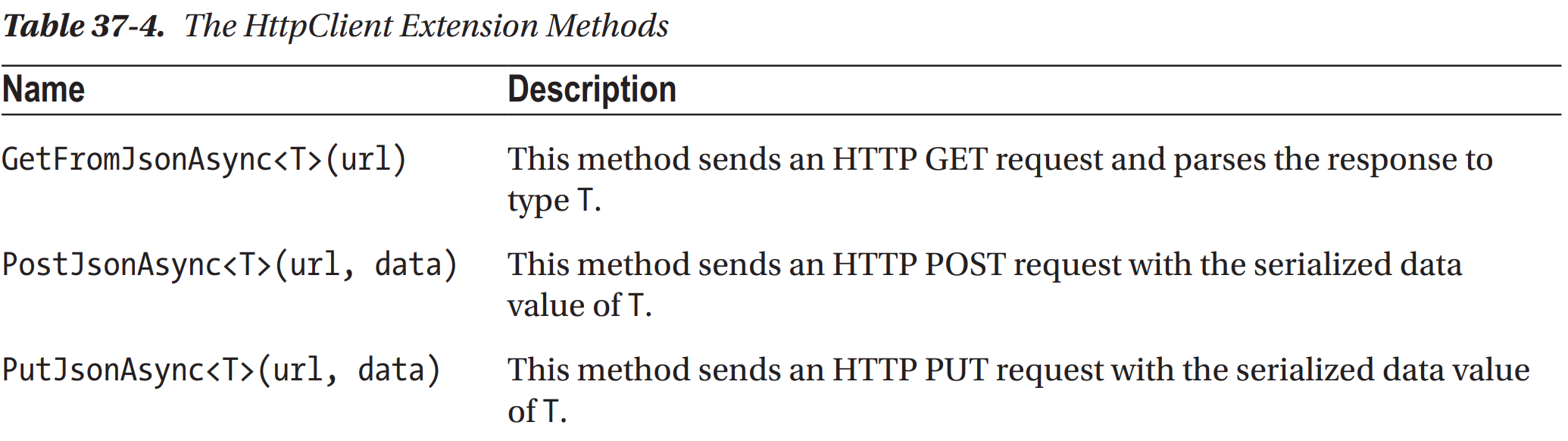


Компонент List использует метод DeleteAsync, чтобы попросить веб-службу удалить объекты, когда пользователь нажимает кнопку Удалить:

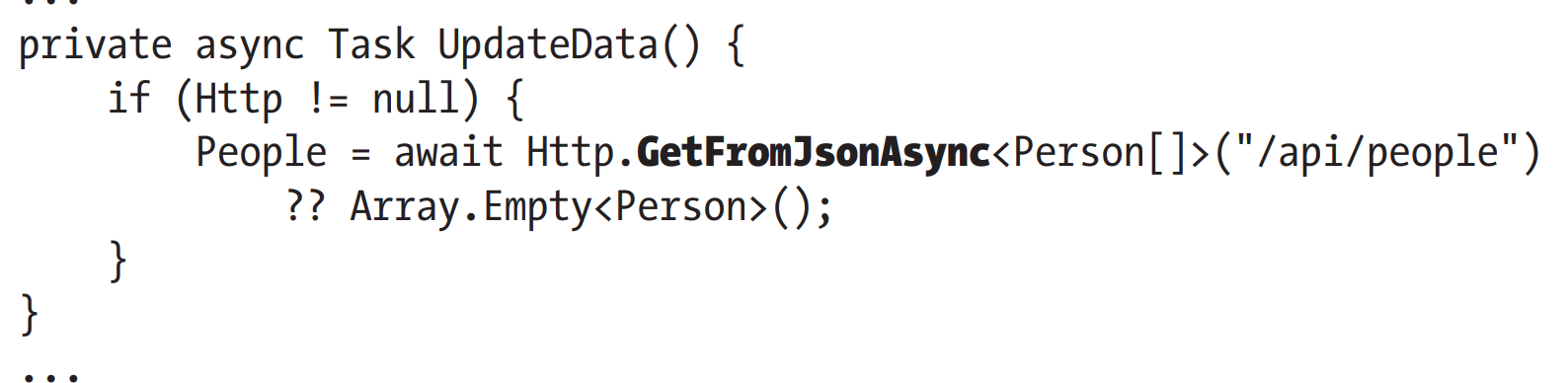


Эти методы полезны когда вам не нужно работать с данными, которые веб сервис возвращает обратно, как в данной ситуации, когда я только проверяю что запрос DELETE был успешен. Обратите внимание, что я указываю путь для URL-адреса запроса только при использовании службы HttpClient, поскольку веб-служба доступна используя ту же схему, хост и порт, что и приложение.

Для операций, в которых веб сервис возвращает данные, методы расширения класса HttpClient, описанные в таблице, более полезны. Эти методы сериализуют данные в JSON, чтобы их можно было отправить на сервер и парсить ответы JSON в объекты C#. Для запросов, возвращающих no result, дженерик тип аргумента может быть упущен:



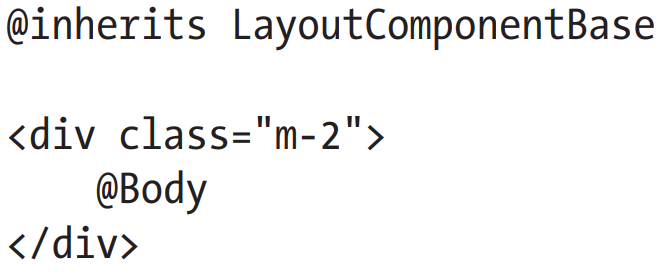
Компонент List использует метод GetJsonAsync<T> для запроса данных с веб-сервиса:



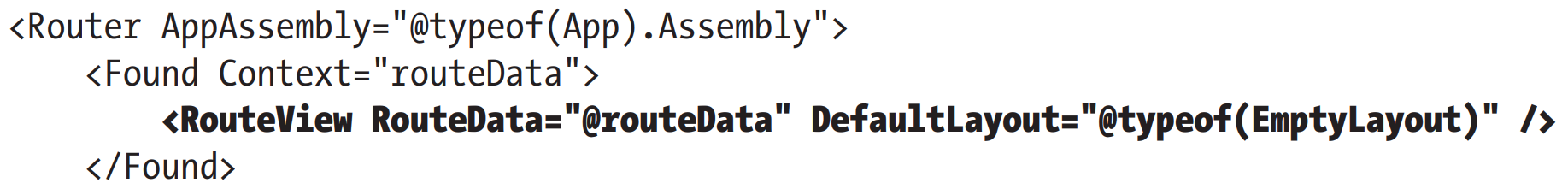
Установка дженерик типа аргумента Person[] говорит HttpClient парсить ответ в массив объектов Person.

**Создание лейаута**

Шаблон, используемый для создания проекта Blazor WebAssembly, включает лейаут, представляющий функциональность навигации для контента заполнителя. Мне не нужна данная функциональность, поэтому для начала я создам новый лейаут. Добавим компонент Razor с наименованием EmptyLayout.razor в папку Shared проекта BlazorWebAssembly:



Я бы мог применить новый лейаут с выражением @layout, как я делал в главе 36, но я собираюсь использовать этот лейаут как дефолтный, изменив конфигурацию роутинга, которая определена в файле App.razor проекта BlazorWebAssembly:





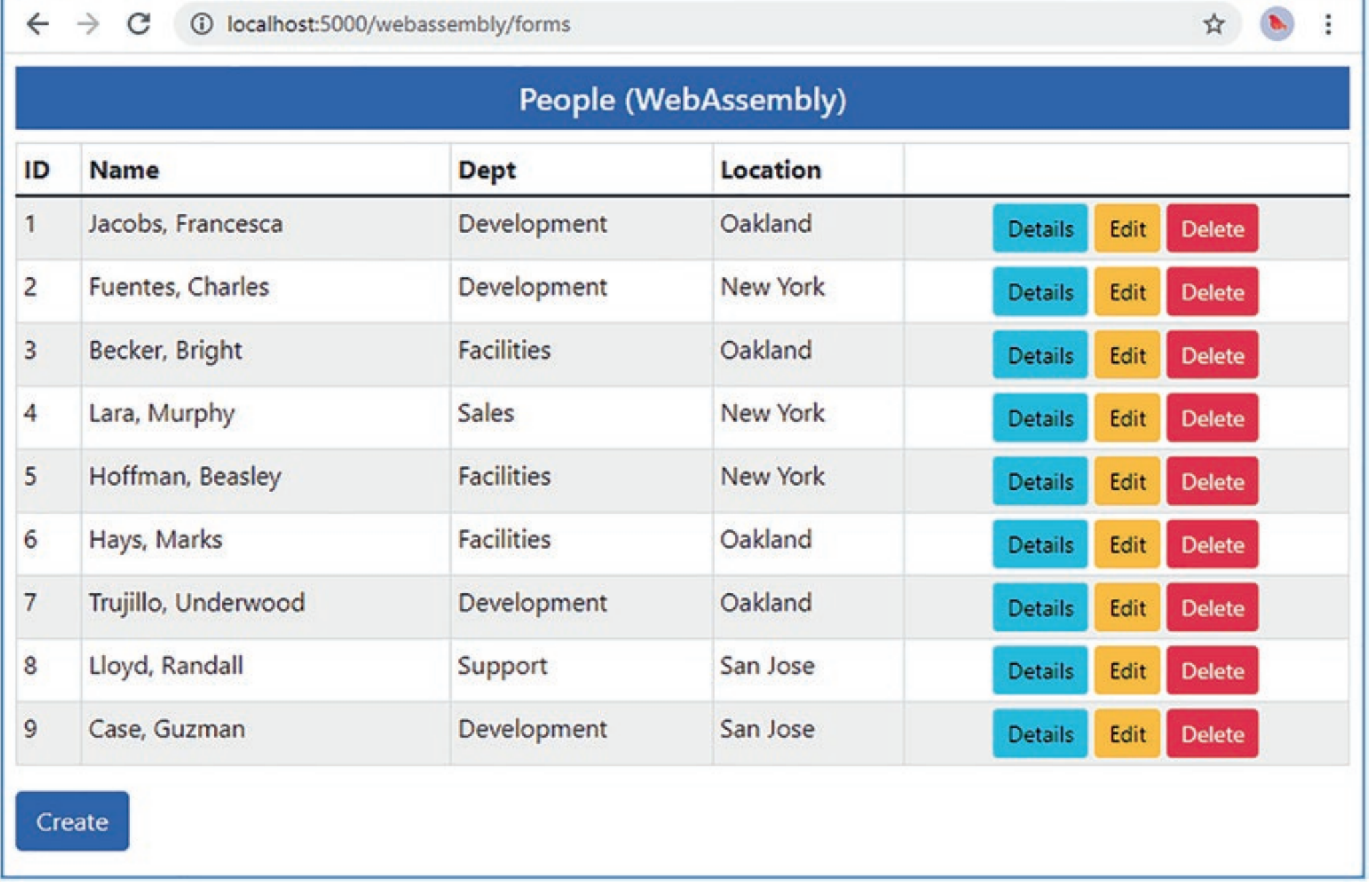
Глава 35 описывает компоненты Router, RouteView, Found, и NotFound.

**Определение CSS стилей**

Шаблон создает проект Blazor WebAssembly с его собственной копией фреймвока Bootstrap CSS с дополнительной таблицей стилей, которая объединяет стили, необходимые для настройки Blazor WebAssembly с элементами ошибок и валидации. Заменим элемент link в файле index.html как показано в листинге и применим стили непосредственно к элементу error. Это имеет эффект удаление стилей, используемых лейаутом шаблона, и использование таблицы стилей Bootstrap CSS, которая была добавлена ​​в проект Advanced:



Чтобы увидеть новый компонент перезапустим приложение и перейдем по URL <http://localhost:5000/webassembly/forms>, который произведет ответ:



Компоненты Blazor WebAssembly следуют стандартному жизненному циклу Blazor и отображают данные, полученные из веб сервиса.

**Завершение приложения Blazor WebAssembly Form**

Только кнопка удаления, отображаемая компонентом List работает корректно. В следующих секциях я завершу приложение Blazor WebAssembly Form созданием дополнительных компонентов.

**Создание компонента Details**

Добавим компонент Razor с наименованием Details.razor в папку Pages проекта BlazorWebAssembly.

Компонент Details имеет только два отличия от аналогичного компонента Blazor Server, следуя за паттерном, установленным компонентом List: данные получаются посредством HttpClient сервиса, и навигационные цели выражаются с помощью относительных URLs. Во всем остальном, как например, получение параметров из роутинга, Blazor WebAssembly работает таким же образом как и Blazor Server.

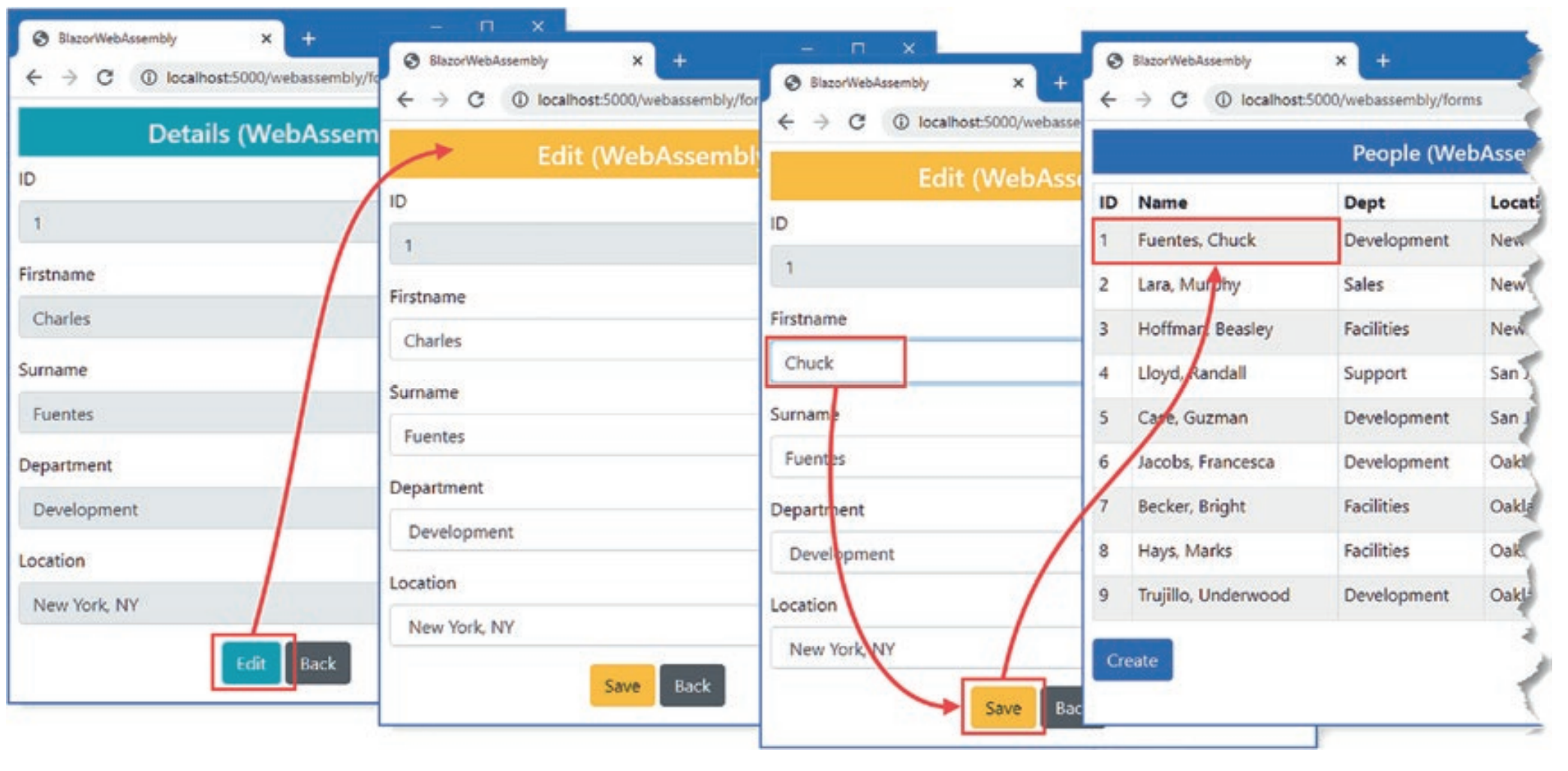
**Создание компоента Editor**

Для завершения приложения формы добавим компонент Razor с именем Editor.razor в папку Pages проекта BlazorWebAssembly.

Этот компонент использует функциональность Blazor form, описанную в главе 36, но использует HTTP запросы к веб сервису, созданному в начале главы, для чтения и записи данных. Метод GetFromJsonAsync<T> используется для чтения данных из веб сервиса, а методы PostAsJsonAsync и PutAsJsonAsync используются для отправки POST или PUT запросов, когда пользователь отправляет форму.

Заметьте, что я не использовал кастомный компонент select или компонент валидации, которые я создал в главе 36. Совместное использование компонентов между проектами, особенно когда Blazor WebAssembly вводится после начала разработки – неуклюже. Я ожидаю, что этот процесс улучшится в будущих выпусках, но для этой главы я просто обошелся без данного функционала. Как результат, элемент select не запускает валидацию когда значение выбрано, кнопка submit не деактивируется автоматически и нет ограничений на комбинацию department и location.

Перезапустим приложение, перейдем по URL <http://localhost:5000/webassembly/forms> и мы увидим Blazor WebAssembly версию приложения формы. Кликнем кнопку Details на первом объекте таблицы и увидим поля выбранного объекта. Кликнем кнопку Edit и вам представится доступная форма. Сделаем изменения и кликнем на кнопку Save, изменения будут отправлены на веб сервис и отображены в таблице:



**Итог**

В этой главе я описал Blazor WebAssembly, показал, как добавить его в проект, и продемонстрировал, как это похоже, но не полностью, на Blazor Server, описанный в главе 36. В следующей главе я объясню как использовать ASP.NET Core Identity для защиты приложения.