Содержание

Предисловие

Я работаю разработчиком ASP.NET уже более семнадцати лет. Мне нравится работать с ASP.NET Core и C#. Но мне всегда чего-то не хватало…

С юных лет я любил создавать пользовательские интерфейсы для веб-приложений. Когда мне было 15, мы с моим лучшим другом решили создать сайт об играх *Quake*, в которые нам нравилось играть. Он создал серверную часть, а я – пользовательский интерфейс. Помню, как я тратил часы и дни на создание вложенных таблиц и встроенных стилей, чтобы придать сайту нужный нам вид. Сейчас это кажется мучительным процессом, но тогда мне это было по душе. На протяжении всей своей карьеры мне очень нравилось создавать клиентскую часть, но это всегда отвлекало меня от C# и ASP.NET Core. Вместо этого я изучал JavaScript и различные фреймворки и инструменты, популярные в данной экосистеме. Хотя мне нравился JavaScript, в действительности при создании клиентской части веб-приложения я хотел использовать свой любимый язык, C#.

Как-то раз в феврале 2018 г. я наткнулся на видео Стива Сандерсона с конференции NDC в Осло, которая состоялась в 2017 г. (https://youtu.be/MiLAE6HMr10). В своем выступлении он представил проведенный им эксперимент, в котором использовалась переносимая среда выполнения .NET, Dot Net Anywhere, которая была скомпилирована в приложение формата Web Assembly. Стив использовал его в качестве основы для создания фреймворка, позволяющего разрабатывать клиентскую часть веб-приложений с использованием Razor (смесь C#, HTML и CSS), которая полностью работала в браузере. Он назвал его Blazor.

Первая экспериментальная предварительная версия Blazor была выпущена компанией Microsoft 22 марта 2018 г., а новые версии появлялись почти каждый месяц. Я следил за каждой версией, пробовал

новые возможности и писал в блоге о своем опыте. 18 апреля 2019 г. Дэниел Рот опубликовал пост в блоге, в котором объявил, что проект выходит из экспериментальной фазы, и Microsoft обязалась выпустить его в качестве поддерживаемого фреймворка для разработки пользовательских интерфейсов для веб-приложений. Наконец-то, недостающий элемент!

После этого поста в блоге Blazor стал набирать силу. Были добавлены дополнительные модели размещения, позволяющие запускать Blazor в самых разных местах. После выхода .NET 6 мы стали свидетелями одного из самых больших прорывов в этом направлении. Был представлен режим AOT (Ahead-of-time), обеспечивающий значительное повышение производительности приложений Blazor WebAssembly. Эволюция Xamarin, платформа .NET MAUI, позволяет Blazor выйти из браузера и использовать его для создания кроссплатформенных настольных и мобильных приложений.

Эта книга – результат моего путешествия в мире Blazor, начиная с того самого просмотра презентации Стива на конференции NDC в Осло и заканчивая созданием реальных бизнес-приложений. На сегодняшний день я опубликовал более 75 постов о Blazor в своем личном блоге и написал много других публикаций. Blazor привил мне страсть к публичным выступлениям, сначала в группах пользователей .NET. и в конечном итоге – на международных конференциях. Мне даже удалось выступить с докладом о Blazor на конференции NDC в Осло в комнате 7, той же комнате, в которой Стив впервые представил свой экспериментальный проект несколькими годами ранее.

О книге

Книга «*Blazor в действии*» была написана с целью превратить вас из новичка в опытного и уверенного разработчика приложений Blazor. Она начинается с описания высокоуровневых концепций, среди которых – модели размещения и компоненты, после чего в ней подробно рассматриваются конкретные функциональные возможности фреймворка, в частности маршрутизация, формы и валидация, а также шаблонные компоненты.

Чтобы помочь вам внедрить различные концепции и возможности, глава за главой мы будем создавать реальное приложение – Blazing Trails. К концу книги у вас будет полноценное эталонное приложение, к которому вы можете обращаться в любое время.

Кому адресована эта книга

Данная книга предназначена для разработчиков, знакомых с основами .NET, C# и веб-технологий (HTML, JavaScript и CSS). Если вы занимаетесь разработкой веб-приложений с использованием Razor Pages или MVC, то погружение в Blazor пройдет для вас гладко. Если вы создавали приложения с использованием веб-API ASP.NET Core и фреймворков JavaScript, в частности React, Vue.js или Angular, то это еще лучше.

Структура книги

Книга состоит из одиннадцати глав и двух приложений:

- глава 1 знакомит вас с Blazor, пользовательскими интерфейсами на основе компонентов и моделями размещения. В ней рассказывается, что такое Blazor и почему можно его использовать, а также почему компоненты лучше подходят для создания пользовательских интерфейсов и как Blazor использует этот

подход. В этой главе также описано, что такое модели размещения, и обсуждаются преимущества и компромиссы каждой из них;

- в главе 2 мы начинаем наше путешествие по созданию приложения Blazing Trails. Она начинается с рассказа о выборе правильного шаблона проекта для нового приложения Blazor, а также способах его создания и запуска. После чего будут рассмотрены ключевые части приложения, а в завершение мы поговорим об организации файлов с разделением по функциональности и о том, как написать свои первые компоненты;
- в главе 3 более подробно рассматривается компонентная модель Blazor. В ней рассказывается, как структурировать компоненты, что такое методы жизненного цикла и в каком порядке они выполняются, как работать с родительскими и дочерними компонентами, а также идет речь о компонентах стилизации и использовании препроцессоров CSS с Blazor;
- в главе 4 рассматривается маршрутизация на стороне клиента, показано, как определять компоненты страницы и перемещаться между ними. В ней также затрагиваются и более сложные темы, в частности передача данных в URL-адресе и навигация программными средствами;
- глава 5 – первая из двух глав, посвященных формам и валидации. В ней рассматриваются такие основы, как использование встроенных компонентов формы Blazor, проверка пользовательского ввода и отправка данных на сервер;
- глава 6 основана на предыдущей главе и охватывает более сложные темы, среди которых – создание собственных компонентов формы, работа с файлами и адаптация формы для редактирования существующих данных;
- глава 7 исследует, как сделать компоненты более пригодными для повторного использования. В ней представлены шаблонные компоненты и способы их дальнейшего улучшения с помощью обобщенных типов;
- в главе 8 показано, как использовать взаимодействие с JavaScript для интеграции существующих библиотек JavaScript в приложение Blazor, а также рассматриваются методы, позволяющие коду C# вызывать код JavaScript, и наоборот;
- глава 9 посвящена защите приложений Blazor. В ней показано, как настроить интеграцию с поставщиком идентификационной информации Auth0;
- в главе 10 рассматривается управление состоянием и реализуется хранилище состояния в памяти. В ней описывается проектирование хранилища состояний и способы хранения состояния с помощью API-интерфейсов локального хранилища браузера;
- глава 11 посвящена тестированию компонентов с использованием фреймворка тестирования bUnit. Рассматриваются пять

ключевых сценариев: тестирование отображаемой разметки, запуск обработчиков событий из тестового кода, имитация аутентификации и авторизации, эмуляция взаимодействия с JavaScript и совместное тестирование нескольких компонентов;
- в приложениях A и B описывается рефакторинг кода, который необходимо выполнять по мере роста приложения. В приложении A описано, как добавить в решение веб-API. Если вы создаете приложение, следуя инструкциям, приведенным в книге, то приложение A должно быть полностью изучено, когда вы будете находиться между главами 4 и 5. В приложении B описывается рефакторинг остальной части приложения для использования веб-API, представленного в приложении A. Приложение B следует изучить после завершения главы 6 и перед началом главы 7.

Соглашения об оформлении программного кода

Эта книга содержит множество примеров исходного кода, как в пронумерованных листингах, так и в виде обычного текста. В обоих случаях исходный код выделен `моноширинным шрифтом, как этот`, чтобы отделить его от остального текста. Иногда используется **`жирный шрифт`**, чтобы выделить код, который изменился по сравнению с предыдущими шагами в главе, например при добавлении новой функции в существующую строку кода.

Во многих случаях исходный код был переформатирован. Мы добавили разделители строк и переделали отступы, чтобы уместить их по ширине книжных страниц. В редких случаях, когда этого оказалось недостаточно, в листинги были добавлены символы продолжения строки (↪). Также из многих листингов, описываемых в тексте, мы убрали комментарии. Некоторые листинги сопровождают аннотации, выделяющие важные понятия.

Исполняемые фрагменты кода можно найти в онлайн-версии книги на странице https://livebook.manning.com/book/blazor-in-action. Исходный код для глав с 2 по 11 доступен в моем репозитории GitHub: https://github.com/chrissainty/blazor-in-action. Код, добавленный в оба приложения, включен в главы, которым они предшествуют.

Весь код из этой книги был создан с использованием .NET 6 SDK и Visual Studio 2022. Однако другие инструменты, в частности Visual Studio Code и .NET CLI или Rider от компании JetBrains, также подойдут для его запуска.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Крис Сэйнти – веб-разработчик с более чем семнадцатилетним стажем. Он работает с Blazor еще со времен первого экспериментального выпуска этого фреймворка в марте 2018 г. и был одним из первых, кто начал вести блог о нем. Крис опубликовал более 75 постов о Blazor в своем блоге, а также пишет статьи для *Visual Studio Magazine*, Progress Telerik и Stack Overflow. Он активный разработчик ПО с открытым исходным кодом и в настоящее время занимается сопровождением некоторых наиболее популярных пакетов NuGet для Blazor, охватывающих интеграцию с API локального хранилища браузера для компонентов пользовательского интерфейса, среди которых – модальные окна и всплывающие сообщения. Когда Крис не сидит за клавиатурой, то выступает на конференциях и рассказывает о Blazor на мероприятиях по всему миру. Все это помогло ему получить награду Microsoft MVP (Most Valuable Professional).

1 Начинаем путешествие в Blazor

В этой главе:

- причины выбрать Blazor для создания приложения;
- преимущества использования компонентов для создания пользовательского интерфейса;
- модели размещения для Blazor.

Сейчас удивительное время для разработчиков на платформе .NET. Мы можем создавать приложения для любой операционной системы, будь то Windows, Linux, iOS, Android или macOS, и, конечно же, разрабатывать потрясающие веб-приложения с помощью ASP.NET MVC, Razor Pages и веб-API – инструментов, которые позволяют создавать надежные и масштабируемые приложения, которые могут работать долгие годы.

Тем не менее этой мозаике давно недоставало одного фрагмента. Общим для всех веб-решений с использованием ASP.NET является то, что они генерируются на сервере. Нам никогда не удавалось использовать мощь C# и .NET для написания клиентских веб-приложений; это всегда было прерогативой JavaScript, но теперь это уже не так.

В первой главе я познакомлю вас с Blazor, революционным фреймворком для разработки клиентских приложений. Будучи созданным на основе веб-стандартов, он позволяет создавать многофункциональные привлекательные пользовательские интерфейсы с использованием C# и .NET. Мы узнаем, как Blazor может сделать процесс

разработки более эффективным и повысить вашу продуктивность, особенно если на вашем сервере также используется .NET. Мы рассмотрим модели размещения – важную концепцию, которую необходимо усвоить, начиная работу с Blazor. Далее изучим компоненты и преимущества их использования для создания пользовательских интерфейсов. Наконец, мы обсудим причины, по которым следует рассмотреть использование Blazor для вашего следующего проекта.

1.1 Почему стоит выбрать Blazor для создания приложения?

Возможно, самой сложной частью запуска нового проекта в последнее время был выбор технологий – ведь доступно так много вариантов. Это особенно актуально, когда речь идет о разработке клиентской части. Нужно выбрать фреймворк (Angular, React, Vue.js), язык (TypeScript, CoffeeScript, Dart) и инструмент сборки (webpack, Parcel, Browserify). Если команда – новичок в этой экосистеме, то попытка определить, какая комбинация технологий поможет сделать проект успешным, может показаться почти невыполнимой задачей; даже опытным командам будет непросто сделать это!

Рассмотрим основные причины выбора Blazor для создания проекта и то, как он может помочь вам избежать проблем, о которых я только что упомянул:

- *C#, современный и многофункциональный язык* – Blazor создан на основе C#, восьмого по популярности языка, согласно опросу разработчиков Stack Overflow от 2021 г. (http://mng.bz/p240). Это мощный, простой в освоении и универсальный язык программирования. Хотя C# – это объектно ориентированный язык, он перенимает все больше и больше возможностей для реализации функционального подхода, если он вам больше по душе. Статическая типизация помогает разработчикам выявлять ошибки во время сборки, ускоряя жизненный цикл разработки и делая его эффективнее. Данный язык существует уже давно, и в настоящее время вышла одиннадцатая версия C#. Он стабилен, хорошо спроектирован, и у него хорошая поддержка;
- *отличный набор инструментов* – сообществу .NET посчастливилось иметь потрясающие инструменты. Visual Studio – чрезвычайно мощная, многофункциональная и расширяемая IDE (интегрированная среда разработки). Она полностью бесплатна для индивидуальных разработчиков, работы с открытым исходным кодом или для некорпоративных команд до пяти человек. Однако если вам нравится что-то более легковесное, то для этого случая есть Visual Studio Code (VS Code), и это один из самых популярных редакторов кода на сегодняшний день. И Visual Studio, и VS Code доступны для разных платформ. Vi-

sual Studio доступен для ОС Windows и macOS, а VS Code – для Windows, macOS и Linux. Существует также отличная альтернативная IDE от компании JetBrains под названием Rider. Это кросс-платформенное приложение, которое может работать как на Windows, так и на macOS и Linux;

- *полноценная экосистема .NET* – как правило, новые фреймворки, не имеют своего окружения, и должно пройти время, пока вокруг на них будет создана экосистема. Однако в Blazor можно подключиться к уже существующей экосистеме .NET. На момент написания книги приложения Blazor доступны в версии .NET 6 и теоретически могут использовать любые совместимые пакеты NuGet. Я говорю *теоретически*, т. к. некоторые пакеты выполняют действия, которые запрещены в случае с WebAssembly, в частности изменение файловой системы;
- *непредвзятость* – в то время как другие фреймворки определяют способ написания приложения, в Blazor этого нет. Для разработки с Blazor нет предпочтительных паттернов или практик; вы можете писать приложения, используя то, что вам знакомо и удобно. Если вам нравится подход MVVM (модель – представление – модель представления), то пробуйте именно его. Если вы предпочитаете использование стиля как в JavaScript библиотеки Redux, то вперед, выбор за вами;
- *плавная кривая обучения* – если вы уже являетесь разработчиком .NET, то кривая обучения Blazor будет для вас очень плавной. Синтаксис Razor, C#, внедрение зависимостей и структура проекта покажутся вам знакомыми, а поскольку Blazor не требует использования ограниченного набора паттернов, вы можете просто использовать то, с чем вы знакомы и что более эффективно для вас. Все это означает, что вы можете быстрее сосредоточиться на использовании возможностей, а не на изучении фреймворка;
- *повторное использование кода* – если вы используете C# на сервере, то Blazor станет отличным дополнением. Одной из самых неприятных проблем с разными клиентскими и серверными языками является невозможность повторного использования кода. Модели или объекты передачи данных (DTO) должны дублироваться между сервером и клиентом; их нужно постоянно обновлять, синхронизировать. Это можно делать вручную или автоматически с использованием генерирования кода, но это просто еще одна сущность, которую нужно настроить и поддерживать. С Blazor все заточено на C# ,и любой общий код можно поместить в общую библиотеку классов .NET и использовать и на сервере, и на клиенте;
- *открытый исходный код* – как и многие проекты Microsoft, Blazor – это проект с полностью открытым исходным кодом, который находится в свободном доступе на GitHub, и вы можете просмотреть, скачать или создать собственную копию. Коман-

да работает в открытом режиме и руководствуется запросами и отзывами разработчиков. При желании вы тоже можете поучаствовать.

1.2 Компоненты как более эффективный способ создания пользовательского интерфейса

В Blazor, как и во многих современных фреймворках, для разработки пользовательских интерфейсов используется концепция компонентов. Все является компонентом – страницы, части страницы, макеты. В Blazor есть различные типы компонентов, а также несколько способов их разработки, и все они будут рассмотрены в последующих главах. Но научиться мыслить с точки зрения компонентов необходимо для написания приложений Blazor.

1.2.1 Что такое компонент?

Компонент можно воспринимать как некий строительный блок. Эти блоки соединяются воедино, чтобы сформировать приложение. Они могут быть большими или маленькими в зависимости от поставленных задач. Однако создание пользовательского интерфейса в виде единого огромного компонента – не лучшая идея. Компоненты демонстрируют свои преимущества, только когда используются как способ разделения логических областей пользовательского интерфейса. Рассмотрим пример пользовательского интерфейса, структурированного в виде компонентов (рис. 1.1).

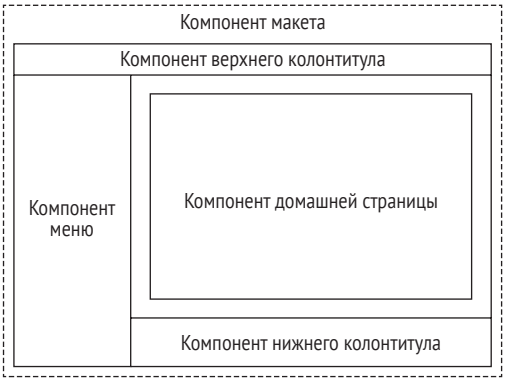

Рис. 1.1 Пример макета интерфейса, разделенного на компоненты

Каждая область интерфейса является компонентом, и у каждого из них определенная ответственность. Также можно заметить, что

здесь формируется иерархия. Компонент макета находится в верхней части дерева; меню, верхний колонтитул, домашняя страница и нижний колонтитул – все это дочерние компоненты компонента макета. У этих компонентов могут быть и, вероятно, будут свои дочерние компоненты. Например, компонент верхнего колонтитула может содержать компоненты логотипа и поиска (рис. 1.2).

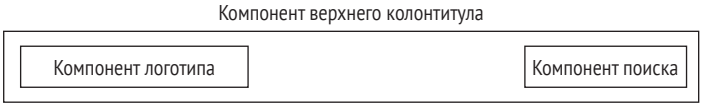

Рис. 1.2 Пример вложенных компонентов для формирования дерева компонентов

1.2.2 Преимущества пользовательского интерфейса на основе компонентов

Как правило, пользовательские интерфейсы содержат повторяющиеся элементы. Весомым преимуществом использования компонентов является тот факт, что вы можете определить элемент в компоненте, а затем повторно использовать компонент везде, где повторяется элемент. Это может значительно сократить количество повторяющегося кода в приложении, а также упрощает сопровождение приложения: если дизайн этого элемента изменится, нужно будет обновить его только в одном месте.

В более сложных сценариях компоненты могут определять собственные API, позволяющие передавать данные и события. Представим себе некое бизнес-приложение. Можно с уверенностью предположить, что в этом приложении будет много данных, которые отображаются в формате таблиц. Один из подходов заключается в создании каждой таблицы в виде отдельного компонента; однако это означает, что в конечном итоге мы получим большое количество компонентов, отображающих данные в таблице. Более эффективный подход – определить отдельный компонент, который принимает набор данных в качестве *параметра*, а затем отображает его в виде таблицы. Теперь у нас есть отдельный компонент для отображения данных в таблице, который можно использовать повторно по всему приложению. Также можно добавить в этот компонент функциональные возможности, в частности сортировку или разбиение на страницы. Эта функциональность будет автоматически доступна для всех таблиц в приложении, поскольку все они повторно используют один и тот же компонент.

Несмотря на то что компоненты часто являются автономными, можно заставить их работать совместно, чтобы создать более сложный пользовательский интерфейс. Например, возьмем сценарий с таблицами данных, о котором мы только что говорили. Это может

быть отдельный компонент, и потенциально он может быть довольно большим. Еще один подход состоит в том, чтобы разделить его на несколько мелких компонентов, каждый из которых выполняет определенную работу. У нас может быть компонент заголовка таблицы, компонент тела таблицы и даже компонент ячейки таблицы. Каждый из них выполняет определенную работу, но они попрежнему являются частью общего компонента таблицы.

1.2.3 Структура компонента Blazor

Теперь, когда мы лучше понимаем, что такое компоненты в общем смысле, рассмотрим пример компонента в Blazor. Для этого мы возьмем компонент из шаблона проекта Blazor. На рис. 1.3 показан пример компонента из стандартного шаблона проекта Blazor – `Counter.razor`.

```
@page "/counter"                                                    ❶

<h1>Counter</h1>                                                    ❷

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {                                                             ❸
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Рис. 1.3 Структура компонента в Blazor

Данный конкретный компонент относится к *маршрутизируемым компонентам*, т. к. у него есть *директива @page*, объявленная в верхней части. И по сути, такие маршрутизируемые компоненты являются просто страницами приложения. Когда пользователь переходит по маршруту `/counter`, этот компонент будет загружен маршрутизатором Blazor. Он отображает простой счетчик с кнопкой, и когда пользователь нажимает кнопку, счетчик увеличивается на единицу, и пользователь видит новое значение.

Хотя понимание кода на данном этапе не важно, мы можем понять структуру компонента. Рисунок 1.3 разделен на три секции, у каждой из которых определенная ответственность:

- *секция 1* – используется для определения директив, добавления операторов `using`, внедрения зависимостей или любой иной типовой конфигурации, применяемой ко всему компоненту;
- *секция 2* – определяет разметку компонента; она написана на встроенном в ASP.NET языке Razor, представляющем собой

смесь C# и HTML. Здесь мы определяем визуальные элементы, составляющие компонент;

- *секция 3* – блок кода. Используется для определения логики компонента. В этой секции можно писать любой допустимый код на C#. Вы можете определить поля, свойства или даже целые классы, если это необходимо.

Мы подробнее будем рассматривать компоненты на протяжении остальной части книги, поэтому пока оставим эту тему. Однако уже сейчас вы можете понять, что представляет собой компонент в Blazor и из чего он состоит.

1.3 Blazor – фреймворк для создания современных пользовательских интерфейсов с помощью C#

Blazor – это полнофункциональный фреймворк для создания современных клиентских приложений, использующий всю мощь C# и .NET. Он позволяет разработчикам создавать популярные приложения, которые работают практически на любой платформе, включая веб-приложения, мобильные и настольные приложения.

Blazor – это альтернатива фреймворкам и библиотекам JavaScript, в частности Angular, Vue.js и React. Если у вас имеется опыт работы с любым из них, то вы, вероятно, обратите внимание на знакомые понятия. Самое заметное заимствование из этих фреймворков в Blazor – это возможность создавать пользовательские интерфейсы с помощью компонентов. И именно эту технологию мы и будем изучать подробнее далее в данной главе.

Поскольку Blazor создан на основе веб-стандартов, он не требует, чтобы конечный пользователь устанавливал на своем компьютере .NET или какой-либо плагин либо расширение для браузера. На самом деле в случае с приложениями Blazor WebAssembly нам даже не нужна платформа .NET, работающая на сервере; эту версию Blazor можно разместить в виде простых статических файлов.

Тот факт, что Blazor создан на основе платформы .NET, означает, что у нас есть доступ ко всему спектру пакетов NuGet. В распоряжении разработчиков на Blazor имеются лучшие в своем классе инструменты, включая Visual Studio, VS Code и Rider от JetBrains. Кроме того, поскольку код .NET является кросс-платформенным, разрабатывать приложения Blazor можно на любой понравившейся нам платформе, будь то ОС Windows, macOS или Linux.

Хотя основное внимание в этой книге уделено разработке веб-приложений на Blazor, хочу подчеркнуть, что модель программирования Blazor также можно использовать для создания кросс-платформенных настольных приложений. С появлением .NET 6 был представлен Blazor Hybrid. Будучи созданным на базе нового фрейм-

ворка для создания кросс-платформенных приложений (также известного как MAUI), он работает аналогично приложениям Electron. Содержимое из приложения Blazor отображается с помощью элемента управления BlazorWebView. Это предлагает свободу выбора относительно того, как структурированы данные приложения. Разработчики могут использовать Blazor и веб-технологии для создания всего пользовательского интерфейса – за исключением chrome, внешнего контейнера приложения, который включает строку заголовка. Либо только часть интерфейса может быть написана с помощью Blazor и работать наряду с нативными элементами управления.

Но это еще не все. Существует длительный экспериментальный проект под названием *Mobile Blazor Bindings* (http://mng.bz/OGOO), возникший в рамках сотрудничества между командами ASP.NET Core и .NET MAUI с целью исследования потенциала и спроса на использование модели программирования Blazor для создания нативных мобильных приложений! Это действительно делает Blazor привлекательной для изучения технологией, поскольку он позволяет разработчикам создавать пользовательские интерфейсы практически для любой платформы или устройства.

Надеюсь, вам уже ясно, что Blazor – это захватывающая технология с большим потенциалом. Но прежде чем мы пойдем дальше, есть одна ключевая концепция, которую важно понимать,– модели размещения. Их мы и рассмотрим далее.

1.3.1 Модели размещения

При первом знакомстве с Blazor вы сразу столкнетесь с моделями размещения. По сути, модели размещения – это место запуска приложения Blazor. В настоящее время у Blazor есть две модели размещения для веб-приложений – Blazor WebAssembly и Blazor Server. Независимо от того, какую модель вы выберете для своего приложения, модель компонентов одна и та же, т. е. компоненты написаны одинаково и могут быть взаимозаменяемы независимо от модели (рис. 1.4).

Рис. 1.4 В Blazor существует разделение между моделями размещения и моделью приложения или компонента. Это означает, что компоненты, написанные для одной модели размещения, можно использовать и для другой

На рис. 1.4 показано абстрактное представление архитектуры Blazor с разделением между моделями приложений и компонентов и моделями размещения. Один из интересных аспектов Blazor состоит в том, что с течением времени становятся доступными другие модели размещения. Это позволяет Blazor работать в самых разных местах, и его можно использовать для создания дополнительных видов пользовательских интерфейсов.

1.3.2 Blazor WebAssembly

Blazor WebAssembly позволяет приложению работать полностью внутри браузера клиента, что делает его прямой альтернативой фреймворкам JavaScript для создания одностраничных приложений или так называемых SPA (single-page application). Чтобы вам было проще понять, как работает эта модель размещения, мы рассмотрим процесс инициализации приложения Blazor WebAssembly (рис. 1.5).

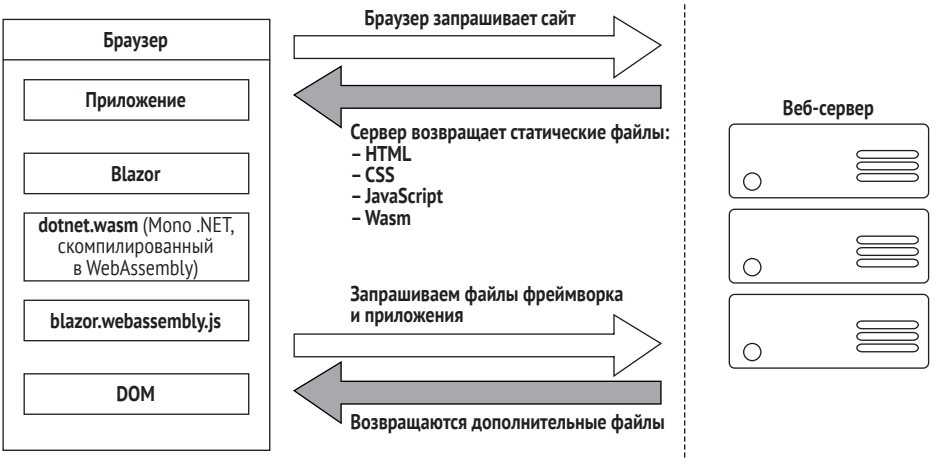

Рис. 1.5 Загрузка приложения Blazor WebAssembly, где показано взаимодействие между браузером клиента и веб-сервером

Процесс начинается, когда браузер выполняет запрос к веб-серверу. Веб-сервер возвращает набор файлов, необходимых для загрузки приложения. К ним относятся главная страница приложения, которая обычно называется index.html; любые статические ресурсы, необходимые приложению, в частности изображения; файлы CSS и JavaScript, а также специальный файл, *blazor.webassembly.js*.

В модели размещения Blazor WebAssembly часть фреймворка Blazor написана на JavaScript и содержится в файле blazor.webassembly.js. Эта часть выполняет три основные функции:

- загружает и инициализирует приложение Blazor в браузере;
- обеспечивает прямые манипуляции с DOM (Document Object Model – объектная модель документа), чтобы Blazor мог выполнять обновления пользовательского интерфейса;

- предоставляет API для сценариев взаимодействия с JavaScript, которые мы подробно обсудим в последующих главах.

На данном этапе вам, наверное, интересно, для чего нам файл JavaScript. Одним из главных преимуществ Blazor является возможность писать логику пользовательского интерфейса с использованием C# вместо JavaScript, верно? Да, это правда. Но на данный момент у WebAssembly есть существенное ограничение: он не может изменять DOM или напрямую вызывать веб-API. Это запланированная функциональность, над которой ведется работа в следующей фазе WebAssembly, но пока она не появится, JavaScript – единственный способ выполнять эти задачи.

Не исключено, что в будущем этот файл больше не понадобится. Все будет зависеть от того, насколько быстро необходимая функциональность будет добавлена в WebAssembly и принята браузерами. Но на данный момент это неотъемлемая часть фреймворка.

Теперь, когда мы это прояснили, вернемся к загрузке приложения Blazor. Хочу отметить, что все файлы, возвращаемые сервером, являются статическими; они не требуют никакой компиляции или манипуляций на стороне сервера. Это означает, что их можно разместить на любом сервисе, который предлагает хостинг статических файлов.

Наличие среды выполнения .NET на сервере не требуется. Впервые это предоставляет разработчикам .NET бесплатные варианты хостинга, в частности страницы GitHub (это относится только к автономным приложениям Blazor WebAssembly).

Как только браузер получит все исходные файлы от веб-сервера, он может обработать их и создать DOM. Далее выполняется файл blazor.webassembly.js. Он выполняет много действий, но в контексте запуска Blazor WebAssembly он скачивает файл blazor.boot.json. Этот файл содержит перечень всех файлов фреймворка и приложений, необходимых для запуска приложения. После скачивания он используется для загрузки остальных файлов, необходимых для запуска приложения.

Большинство этих файлов представляют собой обычные сборки .NET; в них нет ничего особенного, и их можно запускать в любой совместимой среде выполнения .NET. Но есть и другой вид скачиваемого файла – *dotnet.wasm*. Это полноценная среда выполнения .NET, скомпилированная в WebAssembly.

По умолчанию в WebAssembly компилируется только среда выполнения .NET – файлы фреймворка и приложения являются стандартными сборками .NET. Однако в .NET 6 был представлен режим AOT (ahead-of-time), позволяющий разработчикам компилировать свои приложения в WebAssembly. Его преимущество заключается в значительном повышении производительности кода с интенсивным использованием ЦП. При использовании AOT код с интенсивным использованием ЦП, скомпилированный в WebAssembly, будет во много раз более производительным, нежели в случае с интерпре-

тируемым подходом, используемым по умолчанию. Однако здесь есть и некоторый минус или, можно сказать, компромисс. И это размер такого кода. Код, скомпилированный AOT, примерно в два раза больше, чем стандартные сборки, что означает гораздо больший общий размер скачивания для приложения.

WebAssembly

WebAssembly – это низкоуровневый язык, напоминающий ассемблер, который можно запускать в современных веб-браузерах с почти нативной производительностью. Хотя можно писать код WebAssembly напрямую, чаще он используется в качестве цели компиляции для языков более высокого уровня, в частности C/C++ и Rust. Он спроектирован так, что может запускаться с JavaScript, позволяя ему вызывать WebAssembly, и наоборот. WebAssembly работает в той же отдельной защищенной области, или, можно сказать, песочнице безопасности, что и приложения на JavaScript. Подробную информацию о WebAssembly вы найдете на странице https://webassembly.org.

После загрузки файла blazor.boot.json и загрузки перечисленных в нем файлов происходит запуск приложения. Инициализируется среда выполнения WebAssembly .NET, которая, в свою очередь, загружает фреймворк Blazor и, наконец, само приложение. После этого приложение на Blazor WebAssembly начинает свою работу, при этом оно находится полностью внутри браузера клиента. И далее, за исключением ситуаций, когда требуется получать дополнительные данные с сервера, сервер больше не требуется для работы приложения.

Расчет обновлений пользовательского интерфейса

Разобравшись в том, как загружается приложение Blazor WebAssembly, нужно понять, как происходит обновление пользовательского интерфейса. Как и в случае с процессом инициализации, рассмотрим это на примере (рис. 1.6).

В нашем сценарии у нас есть приложение Blazor WebAssembly с двумя страницами, содержащими только заголовок: домашняя страница (Home) и страница счетчика (Counter). Пользователь находится на домашней странице приложения и щелкает по ссылке, чтобы перейти на страницу счетчика. Мы проследим за действиями, которые Blazor выполняет для обновления пользовательского интерфейса при переходе с одной страницы на другую.

Когда пользователь нажимает на ссылку Counter, событие навигации перехватывается средой выполнения JavaScript (blazor.webassembly.js). После чего оно передается фреймворку Blazor, работающему в среде выполнения WebAssembly (dotnet.wasm), и обрабатывается компонентом маршрутизатора Blazor.

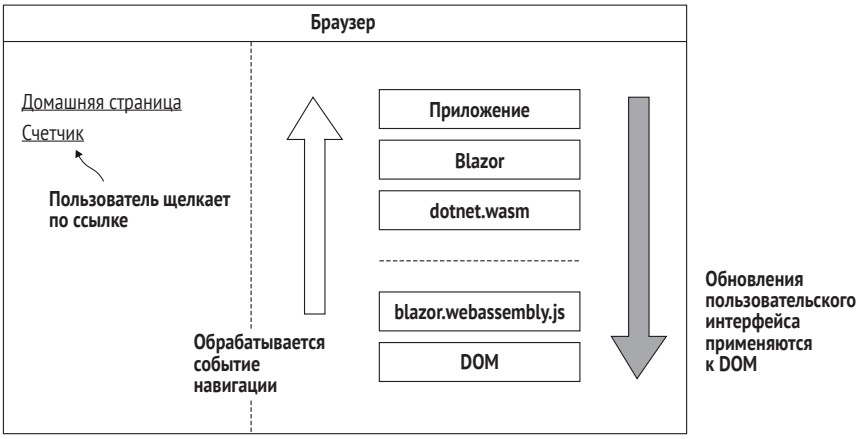

Рис. 1.6 Процесс навигации на стороне клиента в Blazor WebAssembly, от перехода по ссылке до применения обновлений пользовательского интерфейса

Маршрутизатор проверяет свою таблицу маршрутизации на предмет наличия любых маршрутизируемых компонентов, соответствующих маршруту, по которому пытался перейти пользователь. В нашем случае он найдет совпадение с компонентом Counter. Будет создан новый экземпляр этого компонента, и будут выполнены соответствующие методы жизненного цикла.

После завершения Blazor определит минимальное количество изменений, необходимых для обновления DOM, чтобы отобразить компонент Counter. Затем эти изменения будут переданы обратно в среду выполнения Blazor JavaScript, которая применит их к физической модели DOM. На данном этапе пользовательский интерфейс обновится, и пользователь окажется на странице счетчика.

Все это происходит на стороне клиента в пользовательском браузере, и у нас ни разу не возникла необходимость в сервере. Стоит отметить, что в реальных приложениях, скорее всего, в какой-то момент этого процесса произойдет обращение к серверу. Обычно это происходит во время выполнения методов жизненного цикла компонента, к которому осуществляется переход, чтобы загрузить начальные данные для компонента. Но все зависит от конкретного приложения.

Преимущества и недостатки Blazor WebAssembly

Теперь, когда мы знаем немного больше о том, как работает модель размещения Blazor WebAssembly, поговорим о преимуществах и недостатках выбора этой модели. Начнем с преимуществ:

- *приложения выполняются на стороне клиента* – это означает, что нагрузка на сервер намного меньше, так как большая часть работы происходит на стороне клиента. Это может привести

к значительной экономии средств на серверную инфраструктуру и улучшить масштабируемость приложения;

- *возможность работы в автономном режиме* – поскольку приложение полностью работает в браузере, постоянное подключение к серверу не требуется. Это делает приложения более устойчивыми к нестабильным сетевым подключениям. Также проще применить технологию Progressive Web App (PWA). На самом деле в Blazor WebAssembly есть такая возможность, и вы можете выбрать этот вариант при создании своего приложения;
- *развертывание в виде статических файлов* – поскольку приложения Blazor WebAssembly – это просто статические файлы, их можно развернуть в любом месте, где доступен хостинг данного вида файлов. Это открывает возможности, которые ранее никогда не были доступны разработчикам .NET. Такие сервисы, как страницы GitHub, Netlify, Azure Blob Storage, контейнеры объектов AWS (Amazon Web Services) S3 и Azure Static Web Apps, – все это варианты размещения автономных приложений Blazor WebAssembly. Стоимость развертывания статических файлов у всех ведущих поставщиков облачных сервисов ниже по сравнению с размещением веб-приложений;
- *повторное использование кода* – возможно, одно из самых значительных преимуществ Blazor WebAssembly, которое проявляется, если вы используете C# на сервере. Теперь вы можете использовать на своем клиенте те же объекты C#, что и на сервере. Времена, когда использовалась синхронизация моделей TypeScript с их эквивалентами в C# и наоборот, прошли безвозвратно.

Конечно, идеального решения не существует, поэтому разберем недостатки данной модели:

- *полезная нагрузка* – по сравнению с некоторыми приложениями JavaScript, начальный размер скачивания приложений Blazor может быть намного больше (хотя с каждым релизом ситуация улучшается). При публикации можно создать минимальное приложение Blazor размером около 1 МБ; однако другие приложения могут быть значительно больше. Все приложения разные, и у приложений Blazor нет стандартного размера. Однако это единовременные затраты, так как среда выполнения и многие сборки фреймворка кешируются при первой загрузке, а это означает, что последующие загрузки могут быть небольшими, размером всего несколько килобайт;
- *время загрузки* – побочным эффектом размера полезной нагрузки может быть время на ее загрузку. Если у пользователя плохое подключение к интернету, количество времени, необходимое для скачивания исходных файлов, будет больше, что приведет к задержке запуска приложения, выдавая пользователю сообщение о загрузке. Это можно как-то компенсировать,

используя предварительную отрисовку на стороне сервера; однако, несмотря на то что пользователь быстрее увидит первоначальное содержимое, приложение по-прежнему не будет интерактивным, пока все файлы не будут скачаны и инициализированы. Для предварительной отрисовки на стороне сервера приложениям Blazor WebAssembly также требуется элемент ASP.NET Core на сервере, что сводит на нет любые варианты бесплатного хостинга;

- *ограниченная среда выполнения* – возможно, это не недостаток как таковой, но для разработчиков .NET, которые привыкли иметь относительно полную свободу действий в отношении компьютера, на котором работают их приложения, об этом следует знать. Приложения WebAssembly работают в той же изолированной программной среде браузера, что и приложения JavaScript. Это означает, например, что вам будет запрещено обращаться к компьютеру пользователя и делать такие вещи, как обращение к локальной файловой системе;
- *безопасность кода* – как и в случае с приложениями JavaScript, код скачивается и запускается в браузере. Таким образом, пользователь имеет доступ к DLL-библиотекам ваших приложений. Это означает, что вы не должны включать код, содержащий интеллектуальную собственность, в приложение Blazor WebAssembly. Любой код, представляющий ценность, должен храниться на сервере как часть API.

Подводя итог, можно сказать, что Blazor WebAssembly – это модель размещения, которую следует выбирать, если вы хотите заменить фреймворк для разработки одностраничных приложений, например Angular, React или Vue.js. Несмотря на ряд недостатков, которые следует учитывать, выбор этой модели имеет некоторые существенные преимущества.

1.3.3 Blazor Server

Теперь, когда мы увидели, как работает Blazor WebAssembly, обратим наше внимание на модель размещения Server и посмотрим, чем она отличается. Blazor Server была первой моделью размещения для Blazor, выпущенной для промышленного использования примерно за 8 месяцев до версии WebAssembly. Как и в случае с предыдущей моделью, мы рассмотрим инициализацию приложения Blazor Server, чтобы помочь вам понять, как все работает (рис. 1.7).

Процесс начинается с запроса на загрузку сайта из браузера. Когда этот запрос попадает на веб-сервер, могут произойти две вещи: приложение запустится или, если оно уже работает, будет установлен новый сеанс. Почему приложение уже будет работать? В отличие от Blazor WebAssembly, которая больше похожа на настольное приложение, где у каждого пользователя есть свой экземпляр, Blazor Server

запускает один экземпляр приложения, к которому подключаются все пользователи. Следовательно, приложение уже может работать, а новый запрос просто устанавливает новый сеанс.

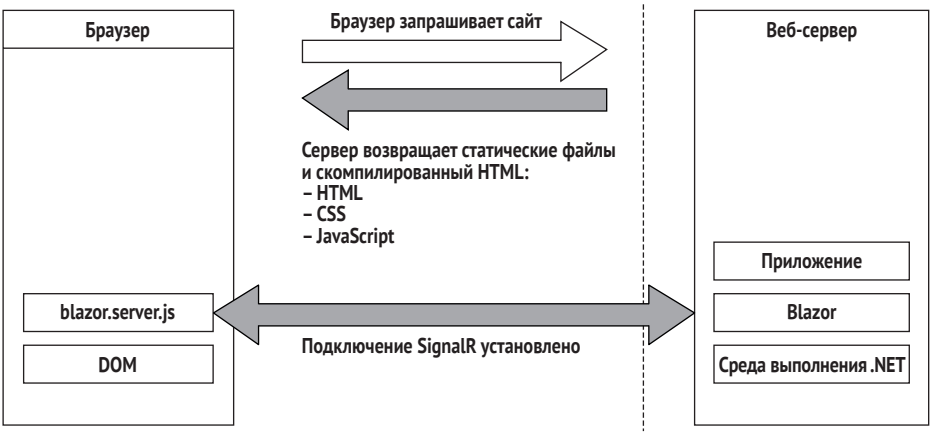

Рис. 1.7 Процесс загрузки приложения Blazor Server

Затем запрос обрабатывается приложением, и исходная полезная нагрузка отправляется обратно в браузер. Сюда входят статические ресурсы, такие как CSS и JavaScript файлы и изображения. Есть также начальный HTML, но он является компилируемым, а не статическим HTML, который мы видели в Blazor WebAssembly. Это происходит потому, что страница на хостинге для приложения Blazor Server – это Razor Page, а не статическая HTML-страница, как в модели WebAssembly. Преимущество данного подхода в том, что он позволяет приложениям Blazor Server использовать предварительный рендеринг на стороне сервера из коробки. Фактически эта функция включается по умолчанию при создании такого типа Blazor-приложения.

Как только первоначальная полезная нагрузка передается в браузер, файлы обрабатываются и создается DOM, а затем выполняется файл *blazor.server.js*. Задача этой среды выполнения – установить обратное подключение SignalR к приложению Blazor, работающему на сервере. На данном этапе приложение полностью загружено и готово к взаимодействию с пользователем.

SignalR

SignalR – это библиотека с открытым исходным кодом от компании Microsoft, позволяющая разработчикам добавлять в свои приложения функциональность реального времени. Клиенты подключаются к серверу через хаб, после чего сервер отправляет обновления клиентам в реальном времени с помощью веб-сокетов (или другой подходящей технологии). Распространенный пример использования SignalR – создание

приложения для чата. Хотя SignalR может быть использован отдельно, в Blazor он применяется для передачи событий и обновлений пользовательского интерфейса между клиентом и сервером, и в данном случае SignalR считается деталью реализации фреймворка, поэтому разработчику, работающему с Blazor Server, не требуется настраивать его или как-то взаимодействовать с ним. На странице https://dotnet.microsoft.com/apps/aspnet/signalr содержится подробная информация о SignalR.

Расчет обновлений пользовательского интерфейса

Что происходит при взаимодействии пользователя с приложением? Ранее мы уже разобрали, как в Blazor WebAssembly события обрабатываются прямо в браузере наряду с вычислением всех обновлений пользовательского интерфейса и применением их к модели DOM. Но здесь этого не может происходить, так как приложение работает на сервере.

Рассмотрим аналогичный пример, что и в случае с Blazor WebAssembly, – приложение Blazor Server с двумя страницами, содержащими только заголовки Home и Counter соответственно. Пользователь находится на домашней странице приложения и щелкает по ссылке, чтобы перейти на страницу счетчика. Рассмотрим процесс, который Blazor выполняет для обновления пользовательского интерфейса при переходе с домашней страницы на страницу счетчика (рис. 1.8).

Рис. 1.8 Процесс обновления пользовательского интерфейса в Blazor Server

Пользователь щелкает по ссылке в меню, и это событие перехватывается средой выполнения Blazor на стороне клиента. После этого среда выполнения обрабатывает событие, чтобы понять, что произошло. В этом случае есть два события: событие щелчка мышью и событие навигации, т. к. это была гиперссылка, по которой щелкнул пользователь. Затем эти два события объединяются и отправляются

обратно на сервер через подключение SignalR, которое устанавливается при запуске приложения.

На сервере сообщение, отправленное клиентом, распаковывается и обрабатывается. Далее фреймворк Blazor вызывает весь необходимый код приложения. В данном случае он создает экземпляр компонента страницы счетчика и выполняет соответствующие методы жизненного цикла.

После завершения этого процесса Blazor определит минимальное количество изменений, необходимых для преобразования текущей страницы в страницу счетчика, а затем отправит их обратно клиенту через подключение SignalR. Чтобы было понятнее: Blazor не будет отправлять клиенту совершенно новую страницу. Он отправит обратно только минимальное количество инструкций, необходимых для обновления текущей модели DOM в соответствии со страницей счетчика. В нашем случае разница только в заголовке. Blazor отправит обратно одну инструкцию, чтобы изменить текст в заголовке с Home на Counter. И больше ничего меняться не будет.

На стороне клиента изменения распаковываются, и необходимые изменения применяются к физической модели DOM. С точки зрения пользователя, он перешел на новую страницу в приложении, страницу счетчика, но фактически он все еще находится на той же странице; просто у нее другой заголовок.

Возможно, вы уже заметили, что весь этот процесс ничем не отличается от того, как работает Blazor WebAssembly; просто он немного растянут из-за соединения SignalR. Это такое же одностраничное приложение, как и в случае с Angular, Vue.js или Blazor WebAssembly. Просто Blazor Server запускает всю логику и рассчитывает обновления пользовательского интерфейса на стороне сервера, а не на стороне клиента. Я уверен, что если бы вам показали два одинаковых приложения, одно из которых написано на Blazor Server, а другое – на Blazor WebAssembly, то вы бы не заметили разницы, как пользователь.

Производительность

Прежде чем мы обсудим преимущества и недостатки этой модели, я хочу кратко упомянуть о производительности. Учитывая большой сетевой трафик в этой модели размещения, вы, возможно, задаетесь вопросом, будет ли она хорошо масштабироваться.

В 2019 г. команда ASP.NET Core провела тестирование, чтобы установить уровни производительности приложений Blazor Server. Они настроили приложение в Azure и протестировали его на виртуальных машинах разной мощности, проверяя количество активных пользователей, которое может поддерживать приложение. Вот результаты:

- *стандартный экземпляр D1 v2 (1 виртуальный ЦП и 3,5 ГБ памяти)* – более 5000 пользователей одновременно;

- *стандартный экземпляр D3 v2 (4 виртуальных ЦП и 14 ГБ памяти)* – более 20 000 пользователей одновременно.

Как видите, Blazor Server вовсе не так плох под нагрузкой. Команда обнаружила, что основным фактором, влияющим на количество поддерживаемых клиентов, является память. Это не лишено смысла, т. к. сервер должен отслеживать всех подключенных к нему клиентов: чем больше клиентов, тем больше информации нужно хранить в памяти.

Другим важным результатом тестирования стало влияние сетевой задержки на приложение. Поскольку все взаимодействия отправляются обратно на сервер для обработки, задержка может иметь большое влияние на удобство использования. Например, если время ответа сервера 250 миллисекунд (мс) для клиента, то каждое взаимодействие будет обрабатываться не менее 500 мс, так как оно должно отправляться на сервер (250 мс), затем обрабатываться и после снова возвращаться (250 мс).

Тестирование показало, что, когда задержка превышала 200 мс, пользовательский интерфейс казался медленным и менее отзывчивым. Поэтому лучше, чтобы клиенты находились на том же континенте, что и сервер. И если приложение предназначено для использования по всему миру, то для Blazor Server необходимо, чтобы серверы распределялись равномерно в пределах 200 мс от клиентов.

Преимущества и недостатки Blazor Server

Рассмотрим также преимущества и недостатки выбора приложения Blazor Server:

- *небольшая полезная нагрузка* – поскольку приложение работает на стороне сервера, а не на стороне клиента, то первоначальная загрузка значительно меньше. В зависимости от статических ресурсов, таких как CSS и изображения, размер приложения Blazor Server может составлять от 100 до 200 КБ;
- *быстрое время загрузки* – приложение загружается намного быстрее, поскольку полезной нагрузки гораздо меньше. Предварительная отрисовка на стороне сервера также помогает, и пользователь не видит сообщения о загрузке;
- *доступ к полнофункциональной среде исполнения* – код приложения выполняется на сервере на основе полной среды выполнения .NET. Это означает, что в такой среде есть полный доступ к файловой системе сервера без каких-либо ограничений по критериям безопасности;
- *защищенность кода* – если у вас есть проприетарный код и вы не хотите, чтобы его можно было скачать и посмотреть, то Blazor Server – подходящий вариант. Весь код приложения выполняется на сервере, а клиенту отправляются только обновления пользовательского интерфейса. Это означает, что у клиента нет доступа к вашему коду.

У Blazor Server есть неплохие преимущества, но каковы недостатки?

- *Высокая нагрузка на сервер* – в то время как Blazor WebAssembly позволяет использовать возможности клиента, у Blazor Server все наоборот. Почти всю работу теперь выполняет сервер. Это означает, что вам могут потребоваться большие инвестиции в инфраструктуру для поддержки приложений Blazor Server. В зависимости от размера приложения для правильного управления сеансами на основе SignalR, используемыми Blazor Sever, также может потребоваться балансировка нагрузки;
- *невозможность работы в автономном режиме* – там, где Blazor WebAssembly нормально работает в автономном режиме, Blazor Server такого себе позволить не может. Соединение SignalR является жизненно важным для приложения, и без него клиент вообще не может функционировать. По умолчанию это приводит к показу пользователю сообщения, в котором говорится, что клиент пытается восстановить соединение. Если это не удается, то пользователь должен обновить страницу, чтобы перезапустить приложение;
- *задержка* – по своей сути приложения Blazor Server чувствительны к проблемам с задержкой. Каждое взаимодействие пользователя с приложением должно отправляться обратно на сервер для обработки и ожидания любых обновлений, которые необходимо применить. Если в соединении между клиентом и сервером большая задержка, проявляется заметное отставание, и пользовательский интерфейс будет работать медленно. В реальных проектах задержка, превышающая 200 мс, будет вызывать проблемы;
- *необходимость наличия стабильного подключения* – учитывая необходимость низкой задержки и невозможность работы в автономном режиме, приложения Blazor Server должны иметь стабильное подключение к интернету. Если соединение прерывается, то пользователи будут постоянно видеть в приложении сообщение о повторном подключении, и это будет их смущать. Самый частый случай таких проблем – когда обращение к серверу происходит с мобильного устройства, с нестабильным подключением к интернету.

Таким образом, если вам нужно быстро загружаемое приложение и у вас есть пользователи с быстрым и стабильным сетевым подключением, то Blazor Server – отличный выбор. При выборе этой модели размещения вы также получаете безопасность кода.

1.3.4 Другие модели размещения

Прежде чем закончить эту главу, хочу познакомить вас еще с двумя моделями размещения – Blazor Hybrid и Blazor Mobile Bindings. Я не буду подробно останавливаться на них, т. к. они не являются цент-

ральной темой данной книги, но знание того, что они существуют, иллюстрирует возможности того, что можно создать с помощью Blazor.

Blazor Hybrid

Blazor Hybrid создан на основе технологии фреймворка .NET MAUI и позволяет разработчикам использовать Blazor для написания кросс-платформенных настольных приложений. Для написания компонентов используются C#, HTML и CSS, как и в Blazor WebAssembly и Blazor Server, а их отрисовка осуществляется с помощью элемента управления BlazorWebView. В следующем листинге показан пример компонента, работающего в приложении Blazor Hybrid.

Листинг 1.1 Компонент, работающий в Blazor Hybrid

```
<div>
    <p>Current count: @currentCount</p>
    <button @onclick="IncrementCount">Click me</button>
</div>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Существенным преимуществом Blazor Hybrid является возможность запускать универсальные компоненты, которые также могут работать в Blazor WebAssembly или Blazor Server. Код, показанный в листинге 1.1, может выполняться во всех трех моделях размещения без каких-либо изменений.

Mobile Blazor Bindings

Mobile Blazor Bindings – это экспериментальная модель размещения, использующая иной подход к созданию компонентов. Компоненты для этой модели должны быть написаны с использованием нативных элементов управления. Следующий листинг содержит тот же самый компонент, что и в листинге 1.1, но он был переписан для модели размещения Mobile Blazor Bindings.

Листинг 1.2 Компонент, работающий в Mobile Blazor Bindings

```
<StackLayout>
    <Label> Current count: @currentCount </Label>
    <Button OnClick="@IncrementCount">Click me</Button>
</StackLayout>
```

```
@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Как видите, модель программирования в обоих примерах одна и та же. Логика в блоке кода остается без изменений; в конце концов, это всего лишь C#. Единственная разница – в разметке, где веб-технологии были заменены на нативные мобильные элементы управления. Это означает, что мы не можем переиспользовать компоненты между моделями размещения для веб-приложений и нативными моделями. Однако как только мы освоим модель программирования Blazor, то сможем с легкостью использовать эти знания для создания других видов пользовательских интерфейсов.

Резюме

Данная глава представляет собой введение в фреймворк Blazor. Мы коснулись большого числа замечательных возможностей Blazor и представили довольно много понятий, которые, вероятно, не имеют большого смысла на данный момент. Не волнуйтесь; в последующих главах мы подробно изучим все это и многое другое. А пока краткое изложение того, что мы рассмотрели:

- Blazor позволяет разработчикам использовать мощь C# и .NET для создания многофункциональных интерактивных пользовательских интерфейсов без необходимости прибегать к JavaScript;
- Blazor – это фреймворк для разработки одностраничных приложений, который может работать полностью в браузере через открытый веб-стандарт WebAssembly, или на сервере, используя подключение SignalR для связи браузера с приложением;
- Blazor WebAssembly можно использовать с любой существующей серверной технологией. Тем не менее использование его в сочетании с серверной частью ASP.NET Core дает реальные преимущества, т. к. код можно легко использовать совместно с помощью библиотеки классов .NET;
- приложения пишутся с использованием компонентов. Они позволяют создавать автономные части пользовательского интерфейса, которые работают по отдельности или в сочетании друг с другом.