

ГЛАВА 2

Ваше первое приложение MVC

Лучший способ оценки инфраструктуры, предназначенной для разработки программного обеспечения, заключается в том, чтобы приступить непосредственно к ее использованию. В настоящей главе мы создадим простое приложение ввода данных с применением ASP.NET Core MVC. Мы будем решать эту задачу пошагово, чтобы вы поняли, каким образом строится приложение MVC. Для простоты мы пока опускаем некоторые технические подробности. Но не беспокойтесь — если вы только начинаете знакомство с MVC, то узнаете много интересного. Когда что-либо используется без пояснения, то приводится ссылка на главу, в которой находятся все необходимые детали.

Установка Visual Studio

Настоящая книга опирается на продукт Visual Studio 2017, который предоставляет среду разработки для проектов ASP.NET Core MVC. Мы будем применять бесплатную редакцию *Visual Studio 2017 Community*, доступную для загрузки на веб-сайте www.visualstudio.com. При установке Visual Studio 2017 вы должны выбрать рабочую нагрузку *.NET Core cross-platform development* (Межплатформенная разработка .NET Core), как показано на рис. 2.1.

На заметку! Версия Visual Studio 2017 предшествовала выпуску ASP.NET Core MVC 2. В случае установки Visual Studio для более ранних версий ASP.NET Core MVC понадобится применить последние обновления. Применить обновления можно, запустив программу установки Visual Studio и щелкнув на кнопке Update (Обновить).

Совет. Среда Visual Studio поддерживает только Windows. Создавать приложения ASP.NET Core MVC на других платформах можно с использованием Visual Studio Code. Продукт Visual Studio Code не обладает всеми возможностями Visual Studio, но он предлагает великолепный редактор и любые средства, требующиеся для разработки приложений MVC. Детали ищите в главе 13.

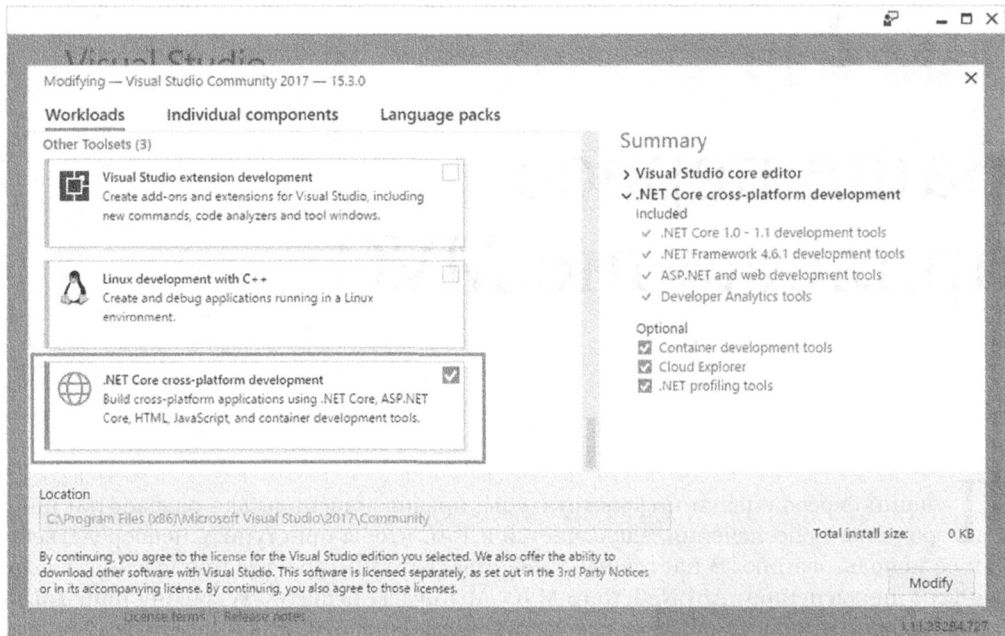


Рис. 2.1. Выбор рабочей нагрузки Visual Studio

Установка .NET Core 2.0 SDK

Установка Visual Studio содержит все средства, необходимые для разработки приложений ASP.NET Core MVC, но не включает комплект .NET Core 2.0 SDK, который должен быть загружен и установлен отдельно.

Перейдем по ссылке <https://www.microsoft.com/net/core>, загрузим программу установки .NET Core SDK для Windows и запустим ее. После завершения работы программы установки откроем окно командной строки или окно PowerShell и введем следующую команду для отображения установленной версии платформы .NET:

```
dotnet --version
```

Если установка прошла успешно, тогда результирующим выводом команды будет 2.0.0.

Создание нового проекта ASP.NET Core MVC

Мы собираемся начать с создания нового проекта ASP.NET Core MVC в среде Visual Studio. Выберем в меню File (Файл) пункт New⇒Project (Создать⇒Проект), чтобы открыть диалоговое окно New Project (Новый проект). Перейдя в раздел Templates⇒Visual C#⇒Web (Шаблоны⇒Visual C#⇒Веб) в панели слева, можно заметить шаблон проекта ASP.NET Core Web Application (Веб-приложение ASP.NET Core). Выберем этот тип проекта (рис. 2.2).

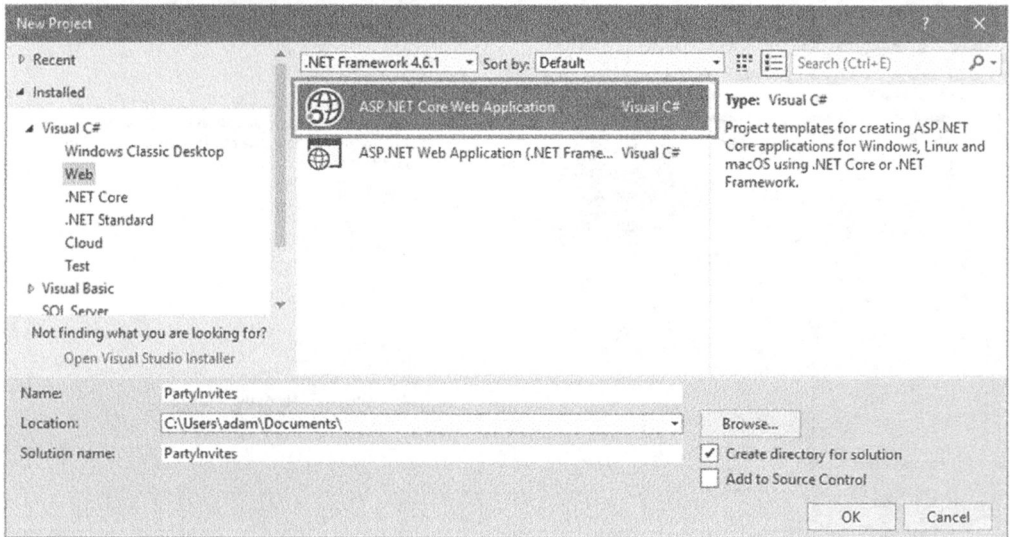


Рис. 2.2. Выбор шаблона проекта ASP.NET Core Web Application

Совет. При выборе шаблона проекта может возникнуть путаница из-за сильно похожих названий шаблонов. Шаблон ASP.NET Web Application (.NET Framework) (Веб-приложение ASP.NET (.NET Framework)) предназначен для создания проектов с использованием унаследованных версий ASP.NET и MVC Framework, которые предшествовали ASP.NET Core. Указанные два шаблона позволяют создавать приложения ASP.NET Core и отличаются применяемой исполняющей средой, предлагая на выбор .NET Framework или .NET Core. Разница между ними объясняется в главе 6, но в книге повсеместно используется вариант .NET Core, поэтому для получения идентичных результатов при работе с примерами приложений вы должны выбирать именно его.

В поле Name (Имя) для нового проекта введем PartyInvites. Для продолжения щелкнем на кнопке ОК. Откроется еще одно диалоговое окно, которое предложит установить начальное содержимое проекта. Удостоверимся, что в раскрывающихся списках слева сверху выбраны элементы .NET Core и ASP.NET Core 2.0 (рис. 2.3).

Для шаблона ASP.NET Core Web Application доступно несколько вариантов, каждый из которых приводит к созданию проекта с отличающимся начальным содержимым. Для целей данной главы выберем вариант Web Application (Model-View-Controller) (Веб-приложение (модель-представление-контроллер)), который создаст приложение MVC с заранее определенным содержимым, чтобы немедленно приступить к разработке.

На заметку! Шаблон проекта Web Application (Model-View-Controller) применяется только в настоящей главе. Мне не нравится пользоваться заранее определенными шаблонами, поскольку они потворствуют интерпретации ряда важных средств наподобие аутентификации как черных ящиков. Моя цель здесь — предоставить вам достаточный объем знаний для понимания и управления каждым аспектом приложения MVC, так что в оставшихся материалах книги будет применяться шаблон Empty (Пустой). Текущая глава посвящена быстрому началу процесса разработки, для чего хорошо подходит шаблон Web Application (Model-View-Controller).

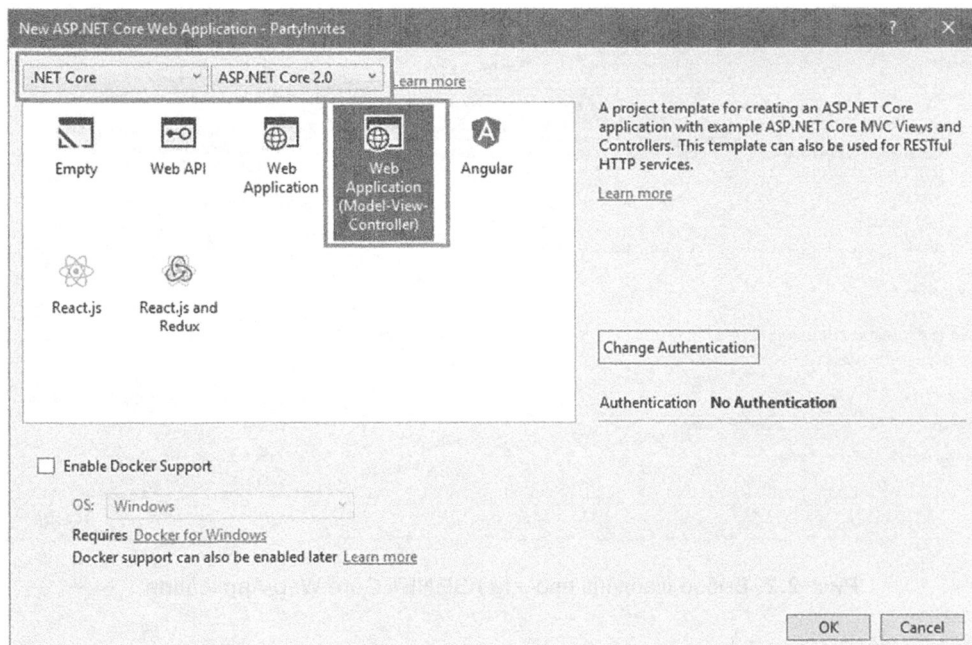


Рис. 2.3. Выбор начальной конфигурации проекта

Щелчком на кнопке Change Authentication (Изменить аутентификацию) и в открывшемся диалоговом окне Change Authentication (Изменение аутентификации) проверим, что выбран переключатель No Authentication (Аутентификация отсутствует), как показано на рис. 2.4. Данный проект не требует какой-либо аутентификации, а в главах 28–30 будет объясняться, как защищать приложения ASP.NET.

Щелчком на кнопке ОК, чтобы закрыть диалоговое окно Change Authentication. Удостоверимся в том, что флажок Enable Docker Support (Включить поддержку Docker) не отмечен, и затем щелчком на кнопке ОК для создания проекта PartyInvites.

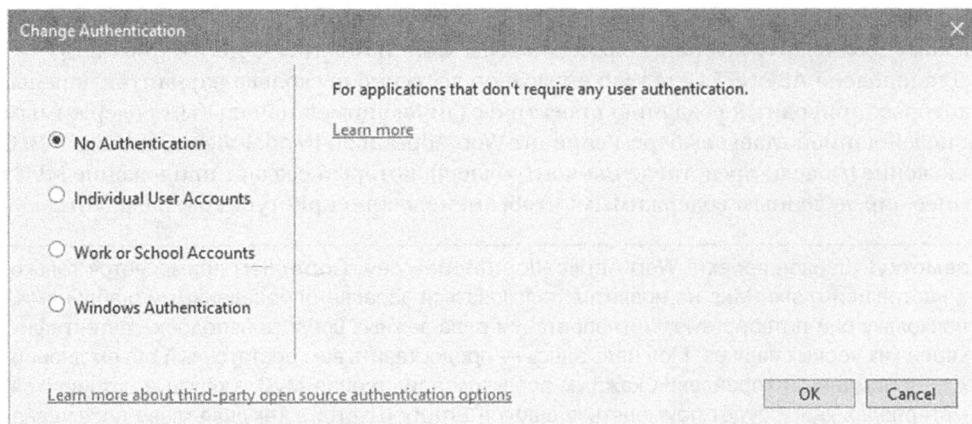


Рис. 2.4. Выбор настроек аутентификации

После того как среда Visual Studio создала проект, в окне Solution Explorer (Проводник решения) появится множество файлов и папок (рис. 2.5). Они представляют собой стандартную структуру для проекта MVC, созданного с использованием шаблона Web Application (Model-View-Controller), и вскоре вы узнаете назначение каждого файла и папки, которые были созданы Visual Studio.

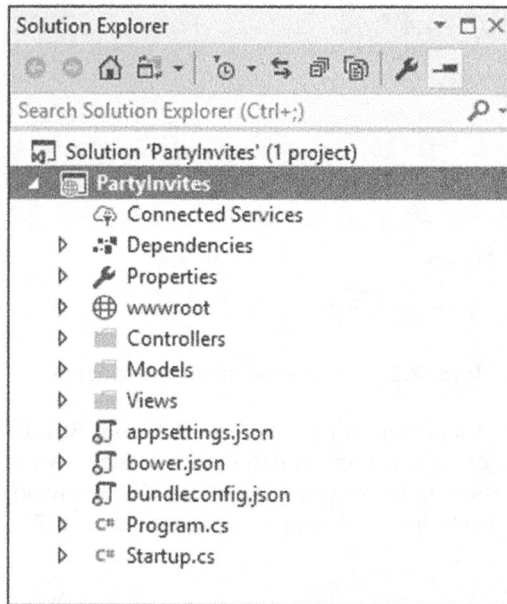


Рис. 2.5. Начальная структура файлов и папок проекта ASP.NET Core MVC

Совет. Если вместо папок `Controllers`, `Models` и `Views` вы видите папку `Pages`, то это означает, что вы выбрали шаблон `Web Application (Веб-приложение)`, а не шаблон `Web Application (Model-View-Controller)`. Я не имею ни малейшего понятия о том, почему в Microsoft решили, что настолько похожие названия шаблонов будут удачной идеей, но в подобной ситуации понадобится удалить созданный проект и начать все заново.

Теперь приложение можно запустить, выбрав в меню `Debug (Отладка)` пункт `Start Debugging (Запустить отладку)`; если появится запрос на включение отладки, тогда нужно просто щелкнуть на кнопке `OK`. Среда Visual Studio скомпилирует приложение, с помощью сервера приложений IIS Express запустит его и откроет окно веб-браузера для запроса содержимого приложения. При запуске проекта в первый раз среде Visual Studio потребуется некоторое время; по завершении процесса будет получен результат, показанный на рис. 2.6.

Когда среда Visual Studio создает проект с применением шаблона `Web Application (Model-View-Controller)`, она добавляет базовый код и содержимое, которое вы наблюдаете после запуска приложения. Далее в главе мы заменим такое содержимое, чтобы создать простое приложение MVC.

По завершении понадобится остановить отладку, для чего закрыть окно браузера или вернуться в Visual Studio и выбрать в меню `Debug` пункт `Stop Debugging (Остановить отладку)`.

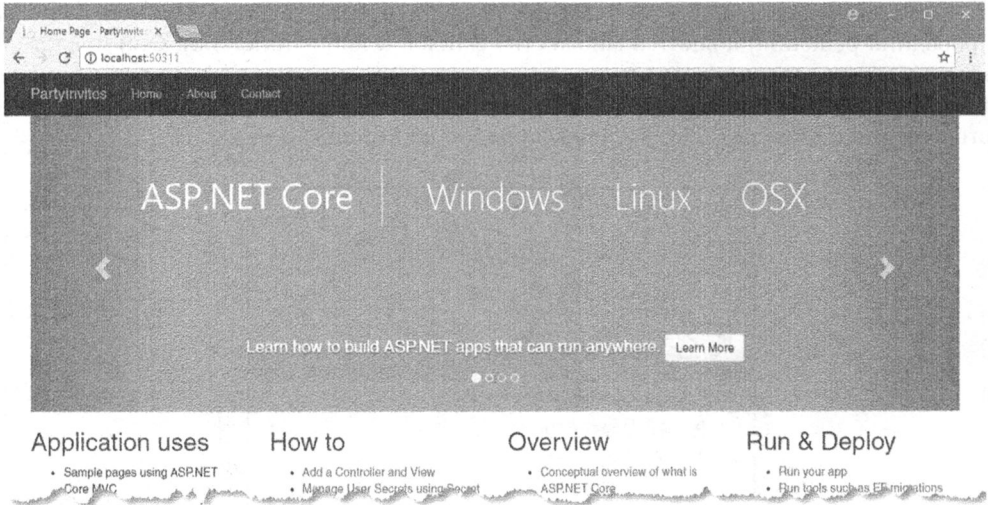


Рис. 2.6. Выполнение примера проекта

Вы видели, что для отображения проекта среда Visual Studio открывает окно браузера. Можно указать любой установленный браузер, щелкнув на кнопке со стрелкой правее кнопки IIS Express в панели инструментов и выбрав нужный вариант из списка в меню Web Browser (Веб-браузер), как показано на рис. 2.7.

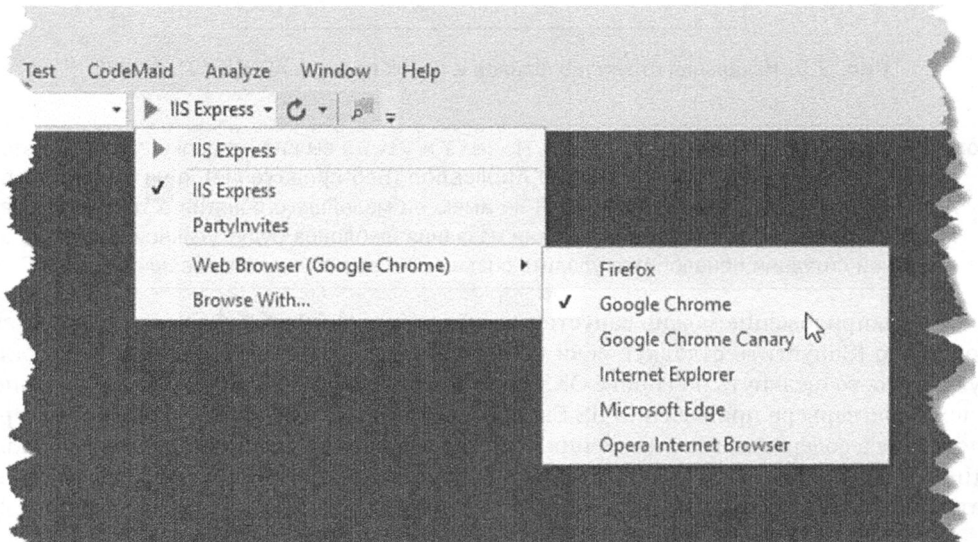


Рис. 2.7. Выбор браузера

В дальнейшем во всех примерах будет использоваться браузер Google Chrome или Google Chrome Canary, но вы можете применять любой современный браузер, включая Microsoft Edge.

Добавление контроллера

В рамках паттерна MVC входящие запросы обрабатываются *контроллерами*. В ASP.NET Core MVC контроллеры — это просто классы C# (обычно унаследованные от класса `Microsoft.AspNetCore.Mvc.Controller`, который является встроенным базовым классом контроллера MVC). Каждый открытый метод в контроллере называется *методом действия*, что означает возможность его вызова из веб-среды через некоторый URL для выполнения какого-то действия. В соответствии с соглашением MVC контроллеры помещаются в папку `Controllers`, автоматически создаваемую Visual Studio при настройке проекта.

Совет. Вы вовсе не обязаны соблюдать указанное или большинство других соглашений MVC, но рекомендуется его придерживаться — и не в последнюю очередь потому, что это поможет уяснить примеры, приведенные в настоящей книге.

Среда Visual Studio добавляет в проект класс стандартного контроллера, который можно увидеть, раскрыв папку `Controllers` в окне `Solution Explorer`. Файл называется `HomeController.cs`. Файлы классов контроллеров имеют имена, завершающиеся словом `Controller`, т.е. в файле `HomeController.cs` содержится код контроллера по имени `Home` — стандартного контроллера, используемого в приложениях MVC. Щелкнем на имени файла `HomeController.cs` в окне `Solution Explorer`, чтобы среда Visual Studio открыла его для редактирования. Отобразится код C#, приведенный в листинге 2.1.

Листинг 2.1. Первоначальное содержимое файла `HomeController.cs` из папки `Controllers`

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
            return View();
        }
        public IActionResult About() {
            ViewData["Message"] = "Your application description page.";
            return View();
        }
        public IActionResult Contact() {
            ViewData["Message"] = "Your contact page.";
            return View();
        }
        public IActionResult Error() {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id
                ?? HttpContext.TraceIdentifier });
        }
    }
}
```

Заменим код в файле `HomeController.cs` кодом, показанным в листинге 2.2. Здесь были удалены все методы кроме одного, у которого изменен возвращаемый тип и его реализация, а также удалены операторы `using` для неиспользуемых пространств имен.

Листинг 2.2. Изменение содержимого файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        public string Index() {
            return "Hello World";
        }
    }
}
```

Изменения не приводят к особо впечатляющим результатам, но их вполне достаточно для хорошей демонстрации. Метод по имени `Index()` изменен так, что теперь он возвращает строку "Hello World". Снова запустим проект, выбрав пункт `Start Debugging` из меню `Debug` в `Visual Studio`.

Совет. Если вы оставили в функционирующем состоянии приложение из предыдущего раздела, тогда выберите в меню `Debug` пункт `Restart` (Перезапустить) или при желании пункт `Stop Debugging` и затем `Start Debugging`.

Браузер сделает HTTP-запрос серверу. Стандартная конфигурация MVC предусматривает, что данный запрос будет обрабатываться с применением метода `Index()`, называемого *методом действия* или просто *действием*, а результат, полученный из этого метода, будет отправлен обратно браузеру (рис. 2.8).

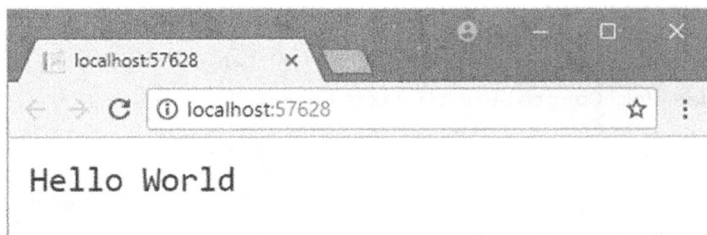


Рис. 2.8. Вывод из метода действия

Совет. Обратите внимание, что среда `Visual Studio` направляет браузер на порт 57628. Внутри URL, который будет запрашивать ваш браузер, почти наверняка будет присутствовать другой номер порта, т.к. `Visual Studio` при создании проекта выделяет произвольный порт. Если вы заглянете в область уведомлений панели задач `Windows`, то найдете там значок для `IIS Express`. Данный значок представляет усеченную версию полного сервера приложений `IIS`, которая входит в состав `Visual Studio` и используется для доставки содержимого и служб `ASP.NET` во время разработки. Развертывание проекта MVC в производственной среде будет описано в главе 12.

Понятие маршрутов

В дополнение к моделям, представлениям и контроллерам в приложениях MVC применяется *система маршрутизации* ASP.NET, которая определяет, как URL отображаются на контроллеры и действия. Маршрут — это правило, которое используется для решения о том, как обрабатывать запрос. Когда среда Visual Studio создает проект MVC, она добавляет ряд стандартных маршрутов, выступающих в качестве начальных. Можно запрашивать любой из следующих URL, и они будут направлены на действие `Index` класса `HomeController`:

- /
- /Home
- /Home/Index

Таким образом, когда браузер запрашивает `http://ваш-сайт/` или `http://ваш_сайт/Home`, он получает вывод из метода `Index()` класса `HomeController`. Можете проверить сказанное самостоятельно, изменив URL в браузере. В настоящий момент он будет выглядеть как `http://localhost:57628/`, но представляющая порт часть может быть другой. Если вы добавите к URL порцию `/Home` или `/Home/Index` и нажмете клавишу `<Enter>`, то получите от приложения MVC тот же самый результат — строку "Hello World".

Это хороший пример получения выгоды от соблюдения соглашений, поддерживаемых ASP.NET Core MVC. В данном случае соглашение заключается в том, что существует класс контроллера по имени `HomeController`, который будет служить стартовой точкой приложения MVC. Стандартная конфигурация, создаваемая средой Visual Studio для нового проекта, предполагает, что мы будем следовать такому соглашению. И поскольку мы *действительно* соблюдаем соглашение, то автоматически получаем поддержку всех URL из приведенного выше списка. Если *не* следовать соглашению, тогда конфигурацию пришлось бы модифицировать для указания на контроллер, созданный взамен стандартного. В рассматриваемом простом примере стандартной конфигурации вполне достаточно.

Визуализация веб-страниц

Выводом предыдущего примера была не HTML-разметка, а просто строка "Hello World". Чтобы сгенерировать HTML-ответ на запрос браузера, понадобится создать *представление*, которое сообщает MVC, каким образом генерировать ответ на запрос, поступивший из браузера.

Создание и визуализация представления

Прежде всего, необходимо модифицировать метод действия `Index()`, как показано в листинге 2.3. Для простоты восприятия в этом и во всех будущих листингах изменения выделяются полужирным.

Листинг 2.3. Визуализация представления в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
namespace PartyInvites.Controllers {
```

```

public class HomeController : Controller {
    public IActionResult Index() {
        return View("MyView");
    }
}
}

```

Возвращая из метода действия объект `ViewResult`, мы инструктируем MVC о визуализации представления. Объект `ViewResult` создается посредством вызова метода `View()` с указанием имени представления, которое должно применяться, т.е. `MyView`. Запустив приложение, можно заметить, что инфраструктура MVC пытается найти представление, как отражено в сообщении об ошибке на рис. 2.9.

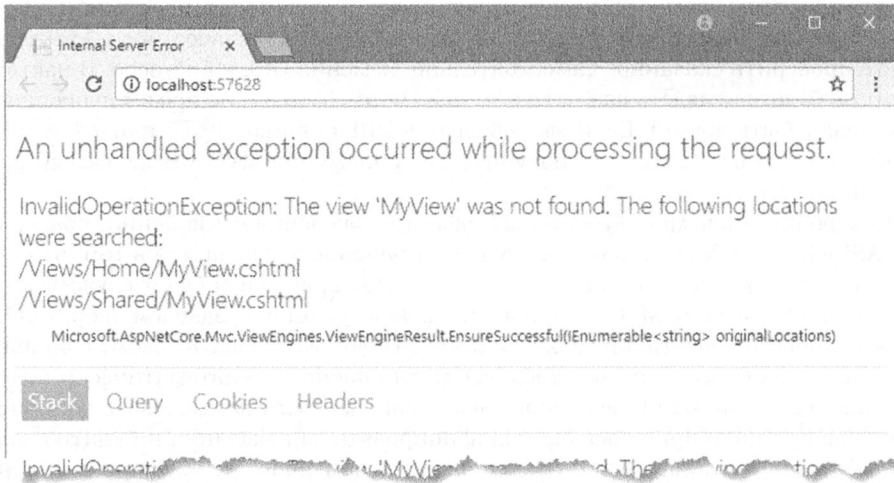


Рис. 2.9. Инфраструктура MVC пытается найти представление

Сообщение об ошибке очень полезно. Оно объясняет, что инфраструктура MVC не смогла найти представление, указанное для метода действия, и показывает, где производился поиск. Представления хранятся в подпапках внутри папки `Views`. Например, представления, которые связаны с контроллером `Home`, содержатся в папке по имени `Views/Home`. Представления, не являющиеся специфическими для отдельного контроллера, хранятся в папке под названием `Views/Shared`. Среда Visual Studio создает папки `Home` и `Shared` автоматически, когда используется шаблон `Web Application (Model-View-Controller)`, и в рамках начальной подготовки проекта помещает в них несколько представлений-заполнителей.

Чтобы создать представление, необходимое для текущего примера, раскроем папку `Views` в окне `Solution Explorer`, щелкнем правой кнопкой мыши на папке `Home` и выберем в контекстном меню пункт `Add ⇒ New Item (Добавить ⇒ Новый элемент)`. Среда Visual Studio предложит список шаблонов элементов. Добравшись до категории `ASP.NET Core ⇒ Web ⇒ ASP.NET` в панели слева, выберем элемент `MVC View Page (Страница представления MVC)` в центральной панели (рис. 2.10). (Не следует применять шаблон `Razor Page (Страница Razor)`, который не относится к MVC Framework.)

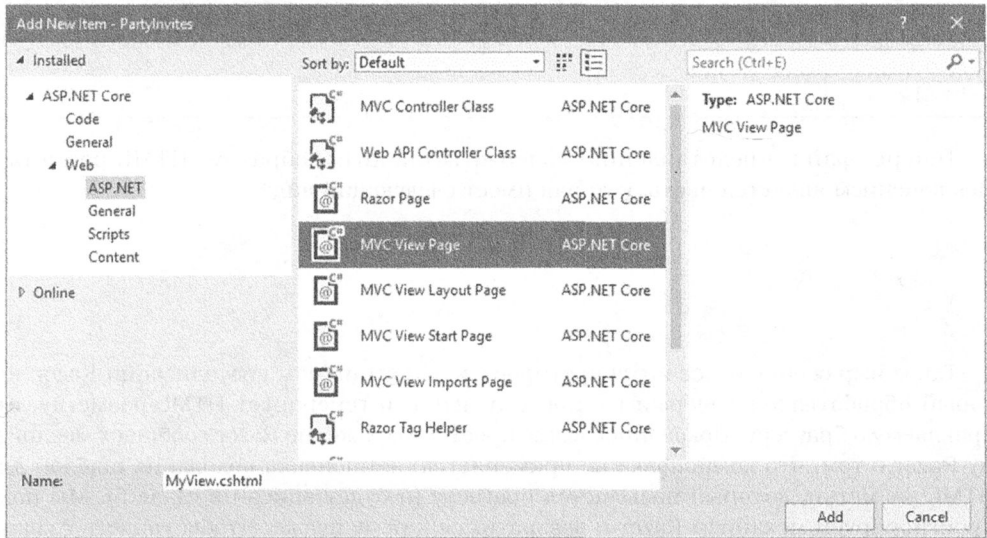


Рис. 2.10. Создание представления

Совет. В папке `Views` уже есть несколько файлов, которые были добавлены Visual Studio с целью предоставления начального содержимого, показанного на рис. 2.6. Такие файлы можно спокойно проигнорировать.

Введем в поле `Name` (Имя) имя `MyView.cshtml` и щелкнем на кнопке `Add` (Добавить) для создания представления. Среда Visual Studio создаст файл `Views/Home/MyView.cshtml` и откроет его для редактирования. Начальное содержимое файла представления — это просто ряд комментариев и заполнитель. Заменяем его содержимым, приведенным в листинге 2.4.

Совет. Довольно легко создать файл представления не в той папке. Если в итоге вы не получили файл по имени `MyView.cshtml` в папке `Views/Home`, тогда удалите созданный файл и попробуйте создать заново.

Листинг 2.4. Замена содержимого файла `MyView.cshtml` из папки `Views/Home`

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
```

```

    Hello World (from the view)
  </div>
</body>
</html>

```

Теперь файл представления содержит главным образом HTML-разметку. Исключением является часть, которая имеет следующий вид:

```

...
@{
    Layout = null;
}
...

```

Такое выражение будет интерпретироваться механизмом визуализации Razor, который обрабатывает содержимое представлений и генерирует HTML-разметку, отправляемую браузеру. Показанное выше простое выражение Razor сообщает механизму Razor о том, что компоновка не применяется; компоновка похожа на шаблон для HTML-разметки, который посылается браузеру (и будет описан в главе 5). Мы пока проигнорируем механизм Razor и возвратимся к нему позже. Чтобы увидеть созданное представление, выберем в меню Debug пункт Start Debugging для запуска приложения. Должен получиться результат, приведенный на рис. 2.11.

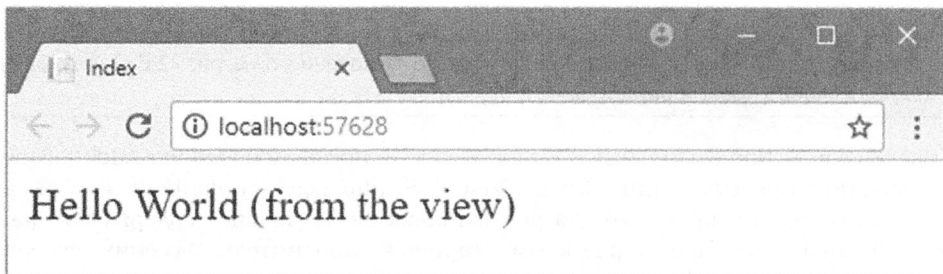


Рис. 2.11. Тестирование представления

При первом редактировании метод действия `Index()` возвращал строковое значение, так что инфраструктура MVC всего лишь передавала браузеру строковое значение в том виде, как есть. Теперь, когда метод `Index()` возвращает объект `ViewResult`, инфраструктура MVC визуализирует представление и возвращает сгенерированную HTML-разметку. Мы сообщили инфраструктуре MVC о том, какое представление должно использоваться, поэтому с помощью соглашения об именовании она автоматически выполнила его поиск. Соглашение предполагает, что имя файла представления совпадает с именем метода действия, а файл представления хранится в папке, названной по имени контроллера: `/Views/Home/MyView.cshtml`.

Кроме строк и объектов `ViewResult` методы действий могут возвращать другие результаты. Например, если мы возвращаем объект `RedirectResult`, то браузер будет перенаправлен на другой URL. Если мы возвращаем объект `HttpUnauthorizedResult`, то предлагаем пользователю войти в систему. Все вместе такие объекты называются *результатами действий*. Система результатов действий позволяет инкапсулировать и повторно использовать часто встречающиеся ответы в действиях. В главе 17 мы рассмотрим их подробнее и продемонстрируем разные способы их применения.

Добавление динамического вывода

Весь смысл платформы для разработки веб-приложений состоит в конструировании и отображении *динамического* вывода. В рамках MVC работа контроллера заключается в подготовке данных и передаче их представлению, которое отвечает за их визуализацию в виде HTML-разметки.

Один из способов передачи данных из контроллера в представление предусматривает использование объекта `ViewBag`, который является членом базового класса `Controller`. По существу `ViewBag` — это динамический объект, в котором можно устанавливать произвольные свойства, делая их значения доступными в любом визуализируемом далее представлении. В листинге 2.5 демонстрируется передача таким способом простых динамических данных в файле `HomeController.cs`.

Листинг 2.5. Установка данных представления в `HomeController.cs` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;
namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View("MyView");
        }
    }
}
```

Данные для представления указываются во время присваивания значения свойству `ViewBag.Greeting`. Свойство `Greeting` не существует вплоть до момента, когда ему присваивается значение, что позволяет передавать данные из контроллера в представление в свободной и гибкой манере, без необходимости в предварительном определении классов. Чтобы получить значение данных, необходимо еще раз сослаться на свойство `ViewBag.Greeting`, но уже в представлении, как показано в листинге 2.6, содержащем изменение, которое было внесено в файл `MyView.cshtml`.

Листинг 2.6. Извлечение значения данных `ViewBag` в файле `MyView.cshtml` из папки `Views/Home`

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
```

```
@ViewBag.Greeting World (from the view)
</div>
</body>
</html>
```

Добавленный в листинге фрагмент — это выражение Razor, которое оценивается, когда MVC применяет представление для генерации ответа. Вызов метода `View()` в методе `Index()` контроллера приводит к тому, что MVC находит файл представления `MyView.cshtml` и запрашивает у механизма визуализации Razor синтаксический анализ содержимого данного файла. Механизм Razor ищет выражения, подобные добавленному в листинге 2.6, и обрабатывает их. В рассматриваемом примере обработка выражения означает вставку в представление значения, которое было присвоено свойству `ViewBag.Greeting` внутри метода действия.

Выбор для свойства имени `Greeting` не диктуется какими-то особыми соображениями. Его можно было бы заменить любым другим именем, и все работало бы точно так же при условии, что имя, используемое в контроллере, совпадает с именем, которое применяется в представлении. Присваивая значения более чем одному свойству, можно передавать из контроллера в представление множество значений данных. После запуска проекта будет виден результат внесенных изменений (рис. 2.12).

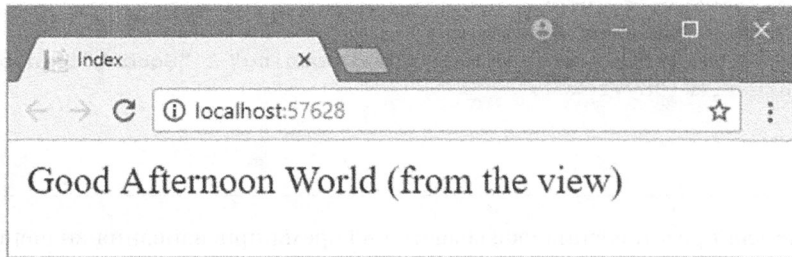


Рис. 2.12. Динамический ответ, сгенерированный MVC

Создание простого приложения для ввода данных

В оставшихся разделах главы будут исследованы другие базовые функциональные средства MVC за счет построения простого приложения для ввода данных. В этом разделе мы собираемся несколько увеличить темп изложения. Целью является демонстрация инфраструктуры MVC в действии, а потому некоторые объяснения того, что происходит “за кулисами”, будут пропущены. Однако не беспокойтесь — мы вернемся к подробному обсуждению данных тем в последующих главах.

Предварительная настройка

Представьте себе, что ваша подруга решила организовать вечеринку в канун нового года и попросила создать веб-приложение, которое позволяет приглашенным ответить на приглашение по электронной почте. Она высказала пожелание относительно четырех основных средств, которые перечислены ниже:

- домашняя страница, отображающая информацию о вечеринке;
- форма, которая может использоваться для ответа на приглашение (*répondez s'il vous plaît* — RSVP);

- проверка достоверности для формы RSVP, которая будет отображать страницу с выражением благодарности за внимание;
- итоговая страница, которая показывает, кто собирается прийти на вечеринку.

В последующих разделах мы достроим проект MVC, созданный в начале главы, и добавим в него перечисленные выше средства. Первый пункт можно убрать из списка, применив то, что было показано ранее — добавить HTML-разметку с подробной информацией о вечеринке в существующее представление. В листинге 2.7 приведено содержимое файла `Views/Home/MyView.cshtml` с внесенными дополнениями.

Листинг 2.7. Отображение подробностей о вечеринке в файле `MyView.cshtml` из папки `Views/Home`

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @ViewBag.Greeting World (from the view)
        <p>We're going to have an exciting party.<br />
        (To do: sell it better. Add pictures or something.)
    </p>
    </div>
</body>
</html>
```

Мы движемся в верном направлении. Если запустить приложение, выбрав в меню Debug пункт Start Debugging, то отобразятся подробности о вечеринке — точнее, заполнитель для подробностей, но сама идея должна быть понятной (рис. 2.13).

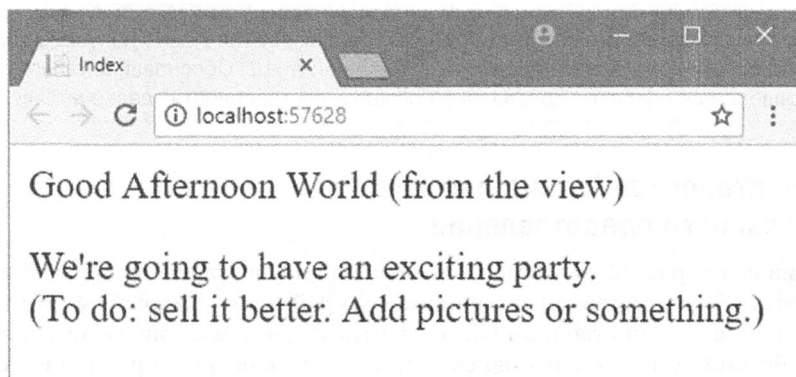


Рис. 2.13. Добавление информации о вечеринке в HTML-разметку представления

Проектирование модели данных

Буква “М” в аббревиатуре MVC обозначает *model* (модель), которая является самой важной частью приложения. Модель — это представление реальных объектов, процессов и правил, которые определяют сферу приложения, известную как *предметная область*. Модель, которую часто называют *моделью предметной области*, содержит объекты C# (или *объекты предметной области*), образующие “вселенную” приложения, и методы, позволяющие манипулировать ими. Представления и контроллеры открывают доступ клиентам к предметной области в согласованной манере, и любое корректно разработанное приложение MVC начинается с хорошо спроектированной модели, которая затем служит центральным узлом при добавлении контроллеров и представлений.

Для проекта `PartyInvites` сложная модель не требуется, поскольку приложение совсем простое, и нужно создать только один класс предметной области, который получит имя `GuestResponse`. Такой объект будет отвечать за хранение, проверку достоверности и подтверждение ответа на приглашение (RSVP).

По соглашению MVC классы, которые образуют модель, помещаются в папку по имени `Models`, которую среда Visual Studio создает автоматически в случае выбора шаблона `Web Application (Model-View-Controller)`.

Чтобы создать файл класса, щелкнем правой кнопкой мыши на папке `Models` в окне `Solution Explorer` и выберем в контекстном меню пункт `Add ⇒ Class (Добавить ⇒ Класс)`. Назначим новому классу имя `GuestResponse.cs` и щелкнем на кнопке `Add (Добавить)`. Приведем содержимое нового файла класса к виду, показанному в листинге 2.8.

Листинг 2.8. Содержимое файла `GuestResponse.cs` из папки `Models`

```
namespace PartyInvites.Models {
    public class GuestResponse {
        public string Name { get; set; }
        public string Email { get; set; }
        public string Phone { get; set; }
        public bool? WillAttend { get; set; }
    }
}
```

Совет. Вы могли заметить, что свойство `WillAttend` имеет тип `bool`, допускающий `null`, т.е. оно может принимать значение `true`, `false` или `null`. Обоснование такого решения будет приведено в разделе “Добавление проверки достоверности” далее в главе.

Создание второго действия и строго типизированного представления

Одной из целей разрабатываемого приложения является включение формы RSVP, что означает необходимость определения метода действия, который сможет получать запросы к форме. В единственном классе контроллера можно определять множество методов действий, а по соглашению связанные действия группируются вместе в одном контроллере. В листинге 2.9 иллюстрируется добавление нового метода действия к контроллеру `Home`.

Листинг 2.9. Добавление метода действия в файле HomeController.cshtml из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View("MyView");
        }

        public IActionResult RsvpForm() {
            return View();
        }
    }
}
```

Метод действия `RsvpForm()` вызывает метод `View()` без аргументов, что сообщает инфраструктуре MVC о необходимости визуализации стандартного представления, связанного с этим методом действия, которым будет представление с таким же именем, как у метода действия (`RsvpForm.cshtml` в данном случае).

Щелчком правой кнопкой мыши на папке `Views/Home` и выберем в контекстном меню пункт `Add ⇒ New Item` (Добавить ⇒ Новый элемент). Выберем шаблон MVC View Page (Страница представления MVC), укажем `RsvpForm.cshtml` в качестве имени нового файла и щелкнем на кнопке `Add` (Добавить), чтобы создать файл. Приведем содержимое нового файла в соответствии с листингом 2.10.

Листинг 2.10. Содержимое файла RsvpForm.cshtml из папки Views/Home

```
@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <div>
        This is the RsvpForm.cshtml View
    </div>
</body>
</html>
```

Содержимое состоит в основном из HTML-разметки, но с добавлением Razor-выражения `@model`, которое используется для создания строго типизированного представления. Строго типизированное представление предназначено для визуализации

зации специфического типа модели, и если указан желаемый тип (в данном случае класс `GuestResponse` из пространства имен `PartyInvites.Models`), то MVC может создать ряд удобных сокращений, чтобы сделать его проще. Вскоре мы задействуем преимущество характеристики строгой типизации.

Чтобы протестировать новый метод действия и его представление, запустим приложение, выбрав в меню `Debug` пункт `Start Debugging`, и с помощью браузера перейдем на URL вида `/Home/RsvpForm`.

Инфраструктура MVC применит описанное ранее соглашение об именовании для направления запроса методу действия `RsvpForm()`, определенному в контроллере `Home`. Данный метод действия указывает MVC о том, что должно визуализироваться стандартное представление, которое посредством еще одного применения того же соглашения об именовании визуализирует `RsvpForm.cshtml` из папки `Views/Home`. Результат показан на рис. 2.14.

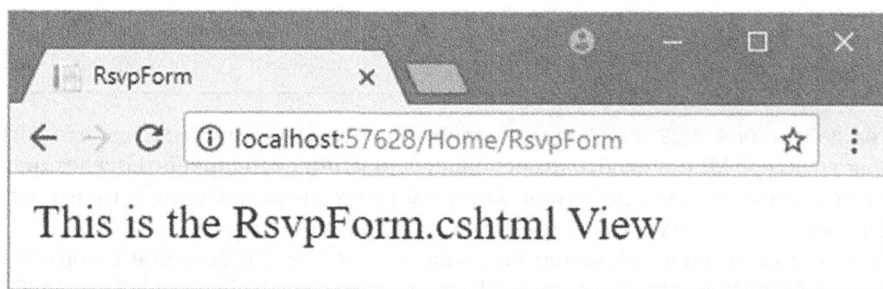


Рис. 2.14. Визуализация второго представления

Ссылка на методы действий

Нам необходимо создать в представлении `MyView` ссылку, чтобы гости могли видеть представление `RsvpForm` без обязательного знания URL, который указывает на специфический метод действия (листинг 2.11).

Листинг 2.11. Добавление ссылки на форму RSVP в файле `MyView.cshtml` из папки `Views/Home`

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @ViewBag.Greeting World (from the view)
        <p>We're going to have an exciting party.<br />
        (To do: sell it better. Add pictures or something.)
    </p>
    </div>
</body>
</html>
```

```
<a asp-action="RsvpForm">RSVP Now</a>
</div>
</body>
</html>
```

В листинге 2.11 добавлен элемент `a`, который имеет атрибут `asp-action`. Данный атрибут является примером атрибута *вспомогательной функции дескриптора*, т.е. инструкцией Razor, которая будет выполнена, когда представление визуализируется. Атрибут `asp-action` — это инструкция по добавлению к элементу `a` атрибута `href`, содержащего URL для метода действия. Работа вспомогательных функций дескрипторов объясняется в главах 24–26, а пока достаточно знать, что `asp-action` — простейший вид атрибута вспомогательной функции дескриптора для элементов `a`. Он указывает Razor на необходимость вставки URL для метода действия, определенного в том же контроллере, для которого визуализируется текущее представление. Запустив проект, можно увидеть ссылку, которую создала вспомогательная функция дескриптора (рис. 2.15).

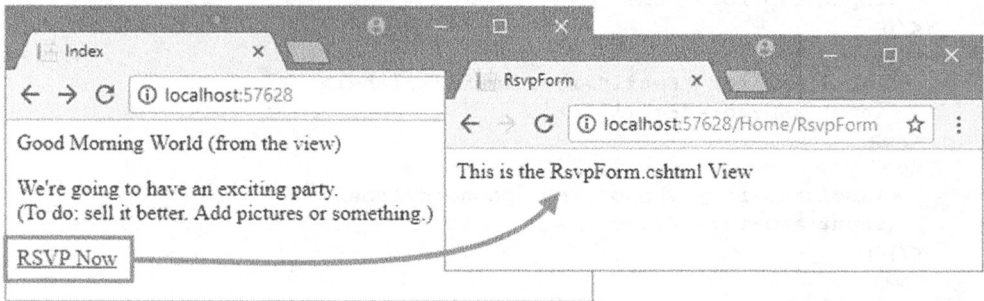


Рис. 2.15. Ссылка на метод действия

Если после запуска приложения навести курсор мыши на ссылку `RSVP Now` (Ответить на приглашение) в окне браузера, то можно заметить, что ссылка указывает на следующий URL (возможно, вашему проекту Visual Studio назначит другой номер порта):

```
http://localhost:57628/Home/RsvpForm
```

Здесь требуется соблюдать один важный принцип: вы должны использовать средства, предлагаемые MVC для генерации URL, а не жестко кодировать их в своих представлениях. Когда вспомогательная функция дескриптора создает атрибут `href` для элемента `a`, она инспектирует конфигурацию приложения, чтобы выяснить, каким должен быть URL. В итоге появляется возможность изменять конфигурацию приложения с целью поддержки разных форматов URL без необходимости в обновлении каких-либо представлений. Особенности работы такого механизма рассматриваются в главе 15.

Построение формы

Теперь, когда строго типизированное представление создано и достижимо из представления `Index`, займемся подгонкой содержимого файла `RsvpForm.cshtml`, чтобы превратить его в HTML-форму для редактирования объектов `GuestResponse` (листинг 2.12).

Листинг 2.12. Создание представления в виде формы в файле RsvpForm.cshtml из папки Views/Home

```

@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <form asp-action="RsvpForm" method="post">
        <p>
            <label asp-for="Name">Your name:</label>
            <input asp-for="Name" />
        </p>
        <p>
            <label asp-for="Email">Your email:</label>
            <input asp-for="Email" />
        </p>
        <p>
            <label asp-for="Phone">Your phone:</label>
            <input asp-for="Phone" />
        </p>
        <p>
            <label>Will you attend?</label>
            <select asp-for="WillAttend">
                <option value="">Choose an option</option>
                <option value="true">Yes, I'll be there</option>
                <option value="false">No, I can't come</option>
            </select>
        </p>
        <button type="submit">Submit RSVP</button>
    </form>
</body>
</html>

```

Для каждого свойства класса модели `GuestResponse` определены элементы `label` и `input` (или элемент `select` в случае свойства `WillAttend`). Каждый элемент ассоциирован со свойством модели с применением еще одного атрибута вспомогательной функции дескриптора — `asp-for`. Атрибуты вспомогательных функций дескрипторов конфигурируют элементы, чтобы привязать их к объекту модели. Вот пример HTML-разметки, которую генерируют вспомогательные функции дескрипторов для отправки браузеру:

```

<p>
    <label for="Name">Your name:</label>
    <input type="text" id="Name" name="Name" value="">
</p>

```

Атрибут `asp-for` в элементе `label` устанавливает значение атрибута `for`. Атрибут `asp-for` в элементе `input` устанавливает атрибуты `id` и `name`. В данный момент это не выглядит особенно полезным, но по мере определения прикладной функциональности вы увидите, что ассоциирование элементов со свойством модели предлагает дополнительные преимущества.

Более непосредственный результат дает атрибут `asp-action` в элементе `form`, который использует конфигурацию маршрутизации URL приложения для установки атрибута `action` в URL, нацеленный на специфический метод действия, например:

```
<form method="post" action="/Home/RsvpForm">
```

Как и в случае атрибута вспомогательной функции дескриптора, примененного к элементу `a`, преимущество такого подхода заключается в том, что вы можете изменять систему URL, используемую приложением, и содержимое, которое генерируется вспомогательными функциями дескрипторов, автоматически отразит изменения.

Форму можно увидеть, запустив приложение и щелкнув на ссылке `RSVP Now` (рис. 2.16).

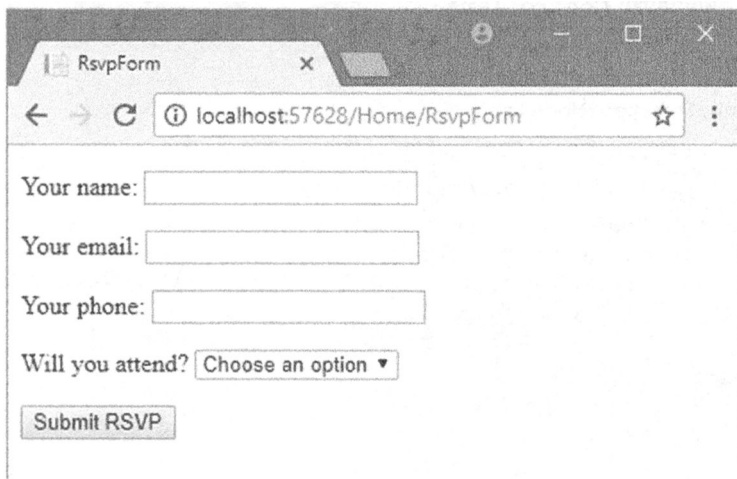


Рис. 2.16. Добавление HTML-формы к приложению

Получение данных формы

Мы пока еще не указали инфраструктуре MVC, что должно быть сделано, когда форма отправляется серверу. При нынешнем состоянии приложения щелчок на кнопке `Submit RSVP` (Отправить RSVP) лишь очищает любые значения, введенные внутри формы. Причина в том, что форма осуществляет обратную отправку методу действия `RsvpForm()` контроллера `Home`, который только сообщает MVC о необходимости повторной визуализации представления.

Чтобы получить и обработать отправленные данные формы, мы собираемся воспользоваться основной возможностью контроллера. Мы добавим второй метод действия `RsvpForm()`, чтобы получить в свое распоряжение следующие возможности.

- *Метод, который отвечает на HTTP-запросы GET.* Каждый раз, когда кто-то щелкает на ссылке, браузер обычно выдает запрос GET. Эта версия действия будет отвечать за отображение изначально пустой формы, когда кто-нибудь впервые посещает /Home/RsvpForm.
- *Метод, который отвечает на HTTP-запросы POST.* По умолчанию формы, визуализированные с помощью `Html.BeginForm()`, отправляются браузером как запросы POST. Эта версия действия будет отвечать за получение отправленных данных и принятие решения о том, что с ними делать.

Обработка запросов GET и POST в отдельных методах C# способствует обеспечению аккуратности кода контроллера, т.к. описанные выше два метода имеют разные обязанности. Оба метода действий вызываются через тот же самый URL, но в зависимости от вида запроса — GET или POST — инфраструктура MVC вызывает подходящий метод. В листинге 2.13 показаны изменения, которые необходимо внести в класс `HomeController`.

Листинг 2.13. Добавление метода в файле `HomeController.cs` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View("MyView");
        }
        [HttpGet]
        public IActionResult RsvpForm() {
            return View();
        }
        [HttpPost]
        public IActionResult RsvpForm(GuestResponse guestResponse) {
            // Что сделать: сохранить ответ от гостя
            return View();
        }
    }
}
```

Существующий метод действия `RsvpForm()` был снабжен атрибутом `HttpGet`, который указывает MVC на то, что данный метод должен применяться только для запросов GET. Затем была добавлена перегруженная версия метода `RsvpForm()`, принимающая объект `GuestResponse`. К ней был применен атрибут `HttpPost`, который сообщает MVC о том, что новый метод будет иметь дело только с запросами POST. Произведенные добавления объясняются в последующих разделах. Кроме того, было импортировано пространство имен `PartyInvites.Models`. Это сделано для того, чтобы на тип модели `GuestResponse` можно было ссылаться без необходимости в указании полностью определенного имени класса.

Использование привязки модели

Первая перегруженная версия метода действия `RsvpForm()` визуализирует то же самое представление, что и ранее (файл `RsvpForm.cshtml`), для генерации формы, показанной на рис. 2.16. Вторая перегруженная версия более интересна из-за наличия параметра. Но с учетом того, что данный метод действия будет вызываться в ответ на HTTP-запрос `POST`, а тип `GuestResponse` является классом `C#`, каким образом они соединяются между собой?

Секрет кроется в привязке модели — удобной функциональной возможности MVC, посредством которой производится разбор входящих данных и применение пар “ключ/значение” в HTTP-запросе для заполнения свойств в типах моделей предметной области.

Привязка модели — мощное и настраиваемое средство, которое избавляет от кропотливого и тяжелого труда по взаимодействию с HTTP-запросами напрямую и позволяет работать с объектами `C#`, а не иметь дело с индивидуальными значениями данных, управляемыми браузером. Объект `GuestResponse`, который передается методу действия `RsvpForm()` в качестве параметра, автоматически заполняется данными из полей формы. Привязка модели, включая ее настройку, подробно рассматривается в главе 26.

Одной из целей приложения является предоставление итоговой страницы с деталями о том, кто придет на вечеринку, что означает необходимость сохранения получаемых ответов. Мы собираемся делать это за счет создания в памяти коллекции объектов. В реальном приложении такой подход не подойдет, т.к. данные ответов будут утрачиваться в результате останова или перезапуска приложения, но он позволяет сосредоточить внимание на MVC и создать приложение, которое может быть легко сброшено в свое начальное состояние.

Совет. В главе 8 будет продемонстрировано использование MVC для постоянного хранения и доступа к данным как часть более реалистичного примера приложения под названием `SportsStore`.

Мы добавили в проект новый файл, щелкнув правой кнопкой мыши на папке `Models` и выбрав в контекстном меню пункт `Add⇒Class` (Добавить⇒Класс). Файл имеет имя `Repository.cs` и содержимое, показанное в листинге 2.14.

Листинг 2.14. Содержимое файла `Repository.cs` из папки `Models`

```
using System.Collections.Generic;
namespace PartyInvites.Models {
    public static class Repository {
        private static List<GuestResponse> responses =
            new List<GuestResponse>();
        public static IEnumerable<GuestResponse> Responses {
            get {
                return responses;
            }
        }
        public static void AddResponse(GuestResponse response) {
            responses.Add(response);
        }
    }
}
```

Класс `Repository` и его члены объявлены статическими, чтобы облегчить сохранение и извлечение данных в разных местах приложения. Инфраструктура MVC предлагает более сложный подход к определению общей функциональности, называемый *внедрением зависимостей*, который будет описан в главе 18, но для простого приложения вроде рассматриваемого вполне достаточно и статического класса.

Сохранение ответов

Теперь, когда есть куда сохранять данные, можно обновить метод действия, который получает HTTP-запросы POST (листинг 2.15).

Листинг 2.15. Обновление метода действия в файле `HomeController.cs` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View("MyView");
        }

        [HttpGet]
        public IActionResult RsvpForm() {
            return View();
        }

        [HttpPost]
        public IActionResult RsvpForm(GuestResponse guestResponse) {
            Repository.AddResponse(guestResponse);
            return View("Thanks", guestResponse);
        }
    }
}
```

Все, что необходимо сделать с данными формы, отправленными в запросе — передать методу `Repository.AddResponse()` в качестве аргумента объект `GuestResponse`, который был передан методу действия, чтобы ответ мог быть сохранен.

Почему привязка модели не похожа на инфраструктуру Web Forms?

В главе 1 упоминалось, что одним из недостатков традиционной инфраструктуры ASP.NET Web Forms было сокрытие деталей HTTP и HTML от разработчиков. Вас может интересовать, делает ли то же самое привязка модели MVC, которая применялась для создания объекта `GuestResponse` из HTTP-запроса POST в листинге 2.15.

Нет, не делает. Привязка модели освобождает нас от решения утомительной и подверженной ошибкам задачи по инспектированию HTTP-запроса и извлечению всех требующихся значений данных, но (что самое важное) при желании мы могли бы обрабатывать запрос

вручную, поскольку MVC обеспечивает легкий доступ ко всем данным запроса. Ничто не скрыто от разработчика, но есть несколько удобных средств, которые упрощают работу с HTTP и HTML; тем не менее, использовать их вовсе не обязательно.

Это может показаться едва заметной разницей, но по мере углубления знаний инфраструктуры MVC вы увидите, что практика разработки в ней полностью отличается от традиционной инфраструктуры Web Forms. Вы всегда будете осведомлены относительно того, как обрабатываются получаемые приложением запросы.

Вызов метода `View()` внутри метода действия `RsvpForm()` сообщает MVC о том, что нужно визуализировать представление по имени `Thanks` и передать ему объект `GuestResponse`. Для создания упомянутого представления щелкнем правой кнопкой мыши на папке `Views/Home` в окне `Solution Explorer` и выберем в контекстном меню пункт `Add⇒New Item (Добавить⇒Новый элемент)`. Укажем шаблон MVC `View Page` (Страница представления MVC) из категории `ASP.NET`, назовем его `Thanks.cshtml` и щелкнем на кнопке `Add (Добавить)`. Среда `Visual Studio` создаст файл `Views/Home/Thanks.cshtml` и откроет его для редактирования. Поместим в файл содержимое, приведенное в листинге 2.16.

Листинг 2.16. Содержимое файла `Thanks.cshtml` из папки `Views/Home`

```
@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
</head>
<body>
    <p>
        <h1>Thank you, @Model.Name!</h1>
        @if (Model.WillAttend == true) {
            @:It's great that you're coming. The drinks are already in the fridge!
        } else {
            @:Sorry to hear that you can't make it, but thanks for letting us know.
        }
    </p>
    <p>Click <a asp-action="ListResponses">here</a> to see who is coming.</p>
</body>
</html>
```

Представление `Thanks.cshtml` применяет механизм визуализации `Razor` для отображения содержимого на основе значения свойства `GuestResponse`, которое передается методу `View()` внутри `RsvpForm()`. Выражение `@model` синтаксиса `Razor` указывает тип модели предметной области, с помощью которого представление строго типизировано.

Для доступа к значению свойства в объекте предметной области используется конструкция `Model.ИмяСвойства`. Например, чтобы получить значение свойства

Name, применяется Model.Name. Не беспокойтесь, если синтаксис Razor пока не понятен — он более подробно объясняется в главе 5.

Теперь, когда создано представление Thanks, мы получили базовый пример обработки формы посредством MVC. Запустим приложение в Visual Studio, выбрав в меню Debug пункт Start Debugging, щелкнем на ссылке RSVP Now, введем на форме какие-нибудь данные и щелкнем на кнопке Submit RSVP. Появится результат, показанный на рис. 2.17 (он может отличаться, если введено другое имя либо указано о невозможности посетить вечеринку).



Рис. 2.17. Представление Thanks

Отображение ответов

В конце представления Thanks.cshtml мы добавили элемент а для создания ссылки, которая позволяет отобразить список людей, собирающихся посетить вечеринку. С применением атрибута вспомогательной функции дескриптора asp-action создается URL, который нацелен на метод действия по имени ListResponses():

```
...
<p>Click <a asp-action="ListResponses">here</a> to see who is coming.
</p>
...
```

Наведя курсор мыши на ссылку, которую отображает браузер, легко заметить, что она указывает на URL вида /Home/ListResponses. Это не соответствует ни одному методу действия в контроллере Home, и если щелкнуть на ссылке, то появится страница ошибки "404 — Not Found" (не найдено).

Мы устраним проблему путем создания в контроллере Home метода действия, на который нацелен URL (листинг 2.17).

Листинг 2.17. Добавление метода действия в файле HomeController.cs

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {
```

```
public ActionResult Index() {
    int hour = DateTime.Now.Hour;
    ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
    return View("MyView");
}

[HttpGet]
public ActionResult RsvpForm() {
    return View();
}

[HttpPost]
public ActionResult RsvpForm(GuestResponse guestResponse) {
    Repository.AddResponse(guestResponse);
    return View("Thanks", guestResponse);
}

public ActionResult ListResponses() {
    return View(Repository.Responses.Where(r => r.WillAttend == true));
}
}
```

Новый метод действия называется `ListResponses()`; он вызывает метод `View()`, используя свойство `Repository.Responses` в качестве аргумента. Именно так метод действия предоставляет данные строго типизированному представлению. Коллекция объектов `GuestResponse` фильтруется с применением LINQ, так что используются только ответы с положительным решением об участии в вечеринке.

Метод действия `ListResponses()` не указывает имя представления, которое должно применяться для отображения коллекции объектов `GuestResponse`, поэтому будет задействовано соглашение об именовании и MVC иницирует поиск представления по имени `ListResponses.cshtml` в папках `Views/Home` и `Views/Shared`. Чтобы создать представление, щелкнем правой кнопкой мыши на папке `Views/Home` в окне `Solution Explorer` и выберем в контекстном меню пункт `Add⇒New Item (Добавить⇒Новый элемент)`. Выберем шаблон MVC View Page (Страница представления MVC) из категории ASP.NET, назначим ему имя `ListResponses.cshtml` и щелкнем на кнопке `Add (Добавить)`. Приведем содержимое нового файла представления в соответствии с листингом 2.18.

Листинг 2.18. Отображение принятых приглашений в файле `ListResponses.cshtml` из папки `Views/Home`

```
@model IEnumerable<PartyInvites.Models.GuestResponse>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Responses</title>
</head>
```

```

<body>
  <h2>Here is the list of people attending the party</h2>
  <table>
    <thead>
      <tr>
        <th>Name</th>
        <th>Email</th>
        <th>Phone</th>
      </tr>
    </thead>
    <tbody>
      @foreach (PartyInvites.Models.GuestResponse r in Model) {
        <tr>
          <td>@r.Name</td>
          <td>@r.Email</td>
          <td>@r.Phone</td>
        </tr>
      }
    </tbody>
  </table>
</body>
</html>

```

Файлы представлений Razor имеют расширение `.cshtml`, потому что содержат смесь кода C# и элементов HTML. Это можно заметить в листинге 2.18, где используется цикл `foreach` для обработки всех объектов `GuestResponse`, которые метод действия передает представлению с применением метода `View()`. В отличие от нормального цикла `foreach` языка C# тело цикла `foreach` из Razor содержит элементы HTML, добавляемые к ответу, который будет отправлен обратно браузеру. В данном представлении для каждого объекта `GuestResponse` генерируется элемент `tr`, который содержит элементы `td`, заполненные значениями свойств объекта.

Чтобы увидеть список в работе, запустим приложение, выбрав в меню Debug пункт Start Debugging, отправим какие-то данные формы и затем щелкнем на ссылке для просмотра списка ответов. Отобразится сводка по данным, введенным вами с момента запуска приложения (рис. 2.18). Представление не оформляет данные привлекательным образом, но пока этого вполне достаточно, а стилизацией мы займемся позже в главе.

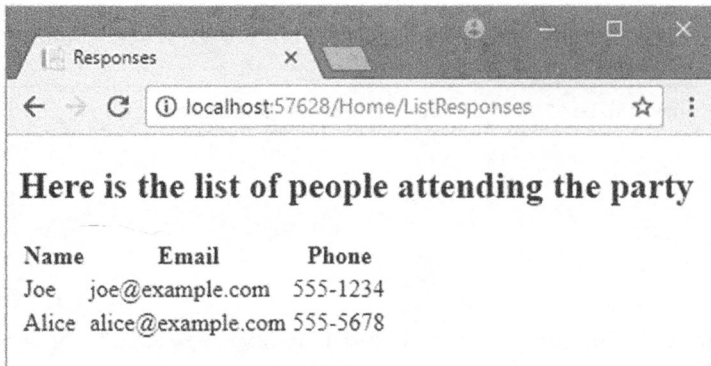


Рис. 2.18. Отображение списка участников вечеринки

Добавление проверки достоверности

Теперь мы готовы добавить в приложение проверку достоверности вводимых данных. В отсутствие проверки достоверности пользователи смогут вводить бессмысленные данные или даже отправлять пустую форму. В приложении MVC проверка достоверности обычно применяется к модели предметной области, а не производится в пользовательском интерфейсе. Это значит, что проверка достоверности определяется в одном месте, но оказывает воздействие в приложении везде, где используется класс модели. Инфраструктура MVC поддерживает *декларативные правила проверки достоверности*, определяемые с помощью атрибутов из пространства имен `System.ComponentModel.DataAnnotations`, т.е. ограничения проверки достоверности выражаются посредством стандартных атрибутов C#. В листинге 2.19 показано, как применить такие атрибуты к классу модели `GuestResponse`.

Листинг 2.19. Применение проверки достоверности в файле `GuestResponse.cs` из папки `Models`

```
using System.ComponentModel.DataAnnotations;

namespace PartyInvites.Models {
    public class GuestResponse {
        [Required(ErrorMessage = "Please enter your name")]
        // Пожалуйста, введите свое имя
        public string Name { get; set; }
        [Required(ErrorMessage = "Please enter your email address")]
        [RegularExpression(".+\\@.+\\.+",
            ErrorMessage = "Please enter a valid email address")]
        // Пожалуйста, введите свой адрес электронной почты
        public string Email { get; set; }
        [Required(ErrorMessage = "Please enter your phone number")]
        // Пожалуйста, введите свой номер телефона
        public string Phone { get; set; }
        [Required(ErrorMessage = "Please specify whether you'll attend")]
        // Пожалуйста, укажите, примете ли участие
        public bool? WillAttend { get; set; }
    }
}
```

Инфраструктура MVC автоматически обнаруживает атрибуты проверки достоверности и использует их для проверки данных во время процесса привязки модели. Мы импортировали пространство имен, которое содержит атрибуты проверки достоверности, так что к ним можно обращаться, не указывая полные имена.

Совет. Как отмечалось ранее, для свойства `WillAttend` был выбран булевский тип, допускающий `null`. Мы поступили так для того, чтобы появилась возможность применять атрибут проверки достоверности `Required`. Если бы использовался обычный булевский тип, то значением, получаемым посредством привязки модели, могло быть только `true` или `false`, и отсутствовала бы возможность определить, выбрал ли пользователь значение. Булевский тип, допускающий `null`, имеет три разрешенных значения: `true`, `false` и `null`. Браузер отправляет `null`, если пользователь не выбрал значение, и тогда атрибут `Required` сообщит об ошибке проверки достоверности. Это хороший пример того, насколько элегантно инфраструктура MVC сочетает средства C# с HTML и HTTP.

Проверку на наличие проблемы с достоверностью данных можно выполнить с применением свойства `ModelState.IsValid` в классе контроллера. В листинге 2.20 показано, как она реализована в методе действия `RsvpForm()`, поддерживающем запросы POST, внутри класса контроллера `Home`.

Листинг 2.20. Проверка на наличие ошибок проверки достоверности в файле `HomeController.cs` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View("MyView");
        }

        [HttpGet]
        public IActionResult RsvpForm() {
            return View();
        }

        [HttpPost]
        public IActionResult RsvpForm(GuestResponse guestResponse) {
            if (ModelState.IsValid) {
                Repository.AddResponse(guestResponse);
                return View("Thanks", guestResponse);
            } else {
                // Обнаружена ошибка проверки достоверности.
                return View();
            }
        }

        public IActionResult ListResponses() {
            return View(Repository.Responses.Where(r => r.WillAttend == true));
        }
    }
}
```

Базовый класс `Controller` предоставляет свойство по имени `ModelState`, которое сообщает информацию о преобразовании данных HTTP-запроса в объекты C#. Если свойство `ModelState.IsValid` возвращает `true`, то известно, что инфраструктура MVC сумела удовлетворить ограничения проверки достоверности, указанные через атрибуты для класса `GuestResponse`. Когда такое происходит, визуализируется представление `Thanks`, как делалось до того.

Если свойство `ModelState.IsValid` возвращает `false`, то известно, что есть ошибки проверки достоверности. Возвращаемый свойством `ModelState` объект предоставляет подробные сведения о каждой возникшей проблеме, но нам нет нужды погружаться на такой уровень деталей, поскольку мы можем полагаться на удобное средство, которое автоматизирует процесс указания пользователю на необходимость решения любых проблем, вызывая метод `View()` без параметров.

Когда MVC визуализирует представление, механизм Razor имеет доступ к деталям любых ошибок проверки достоверности, связанных с запросом, и вспомогательные функции дескрипторов могут обращаться к этим деталям, чтобы отображать сообщения об ошибках пользователю. В листинге 2.21 приведено содержимое файла представления `RsvpForm.cshtml` с добавленными атрибутами вспомогательных функций дескрипторов для проверки достоверности.

Листинг 2.21. Добавление сводки по проверке достоверности в файле `RsvpForm.cshtml` из папки `Views/Home`

```
@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <form asp-action="RsvpForm" method="post">
        <div asp-validation-summary="All"></div>
        <p>
            <label asp-for="Name">Your name:</label>
            <input asp-for="Name" />
        </p>
        <p>
            <label asp-for="Email">Your email:</label>
            <input asp-for="Email" />
        </p>
        <p>
            <label asp-for="Phone">Your phone:</label>
            <input asp-for="Phone" />
        </p>
        <p>
            <label>Will you attend?</label>
            <select asp-for="WillAttend">
                <option value="">Choose an option</option>
                <option value="true">Yes, I'll be there</option>
                <option value="false">No, I can't come</option>
            </select>
        </p>
        <button type="submit">Submit RSVP</button>
    </form>
</body>
</html>
```

Атрибут `asp-validation-summary` применяется к элементу `div` и отображает список ошибок проверки достоверности при визуализации представления. Значение для атрибута `asp-validation-summary` берется из перечисления по име-

ни `ValidationSummary`, которое указывает типы ошибок проверки достоверности, помещаемые в сводку. Мы указали `All`, что является хорошей отправной точкой для большинства приложений, а в главе 27 будут описаны другие значения.

Чтобы взглянуть, как работает сводка по проверке достоверности, запустим приложение, заполним поле `Name` и отправим форму, не вводя другие данные. Появится сводка с ошибками (рис. 2.19).

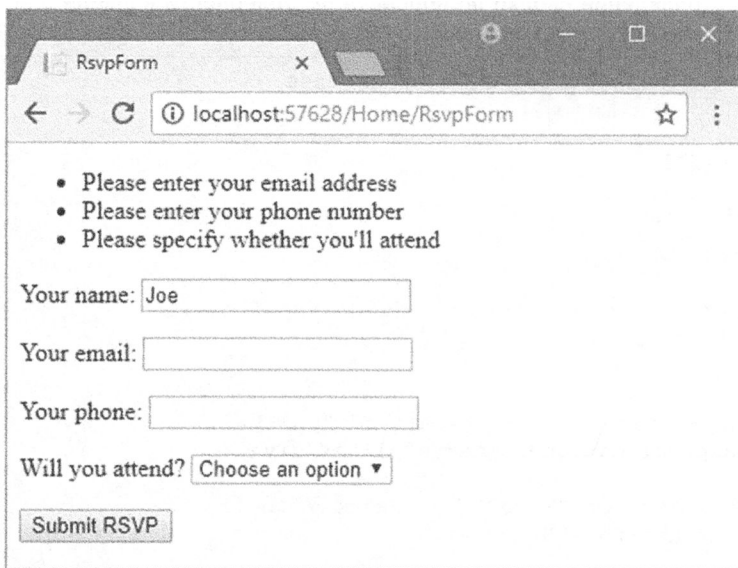


Рис. 2.19. Отображение ошибок проверки достоверности

Метод действия `RsvpForm()` не визуализирует представление `Thanks` до тех пор, пока не будут удовлетворены все ограничения проверки достоверности, примененные к классу `GuestResponse`. Обратите внимание, что введенные в поле `Name` данные были сохранены и отображены снова при визуализации механизмом `Razor` представления со сводкой проверки достоверности. Это еще одно преимущество привязки модели, и оно упрощает работу с данными формы.

На заметку! Если вы работали с ASP.NET Web Forms, то знаете, что в Web Forms поддерживается концепция *серверных элементов управления*, которые сохраняют состояние, сериализуя значения в скрытое поле формы по имени `__VIEWSTATE`. Привязка модели MVC не имеет никакого отношения к концепциям серверных элементов управления, обратным отправкам или состоянию представления, принятым в Web Forms. Инфраструктура MVC не внедряет скрытое поле `__VIEWSTATE` в визуализированные HTML-страницы. Взамен она включает данные, устанавливая атрибуты `value` элементов управления `input`.

Подсветка полей с недопустимыми значениями

Атрибуты вспомогательных функций дескрипторов, которые ассоциируют свойства модели с элементами, обладают удобным средством, которое можно использовать

в сочетании с привязкой модели. Когда свойство класса модели не проходит проверку достоверности, атрибуты вспомогательных функций дескрипторов будут генерировать несколько отличающуюся HTML-разметку. Вот элемент `input`, который генерируется для поля `Phone` при отсутствии ошибок проверки достоверности:

```
<input type="text" data-val="true"
      data-val-required="Please enter your phone number"
      id="Phone" name="Phone" value="">
```

Для сравнения ниже показан тот же HTML-элемент после того, как пользователь отправил форму, не введя данные в текстовое поле (что является ошибкой проверки достоверности, поскольку мы применили к свойству `Phone` класса `GuestResponse` атрибут проверки `Required`):

```
<input type="text" class="input-validation-error" data-val="true"
      data-val-required="Please enter your phone number" id="Phone"
      name="Phone" value="">
```

Отличие выделено полужирным: атрибут вспомогательной функции дескриптора `asp-for` добавил к элементу `input` класс по имени `input-validation-error`. Мы можем воспользоваться такой возможностью, создав таблицу стилей, которая содержит стили CSS для этого класса и другие стили, применяемые различными атрибутами вспомогательных функций дескрипторов.

По принятому в проектах MVC соглашению статическое содержимое, доставляемое клиентам, помещается в папку `wwwroot`, подпапки которой организованы по типу содержимого, так что таблицы стилей CSS находятся в папке `wwwroot/css`, файлы JavaScript — в папке `wwwroot/js` и т.д.

Для создания таблицы стилей щелкнем правой кнопкой мыши на папке `wwwroot/css` в окне `Solution Explorer` и выберем в контекстном меню пункт `Add ⇒ New Item` (Добавить ⇒ Новый элемент). В открывшемся диалоговом окне `Add New Item` (Добавление нового элемента) перейдем в раздел `ASP.NET Core ⇒ Web ⇒ Content` (`ASP.NET Core ⇒ Web ⇒ Содержимое`) и выберем шаблон `Style Sheet` (Таблица стилей) из списка шаблонов (рис. 2.20).

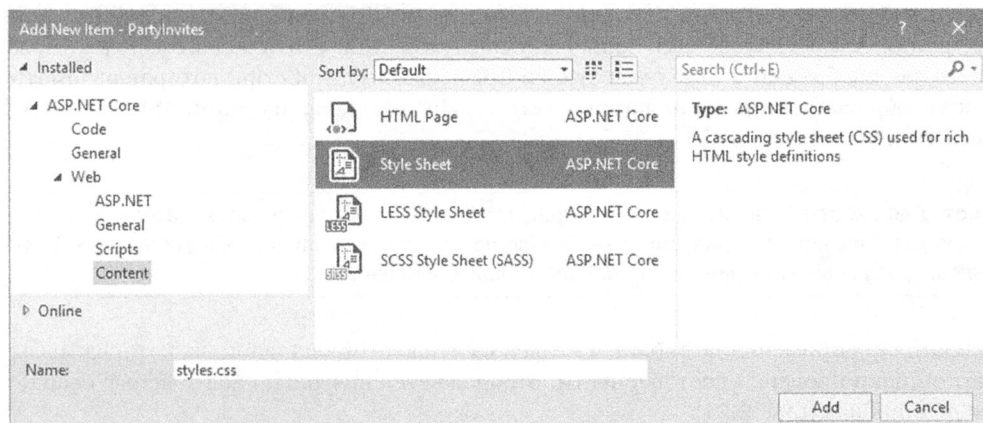


Рис. 2.20. Создание таблицы стилей CSS

Совет. Когда для проекта используется шаблон Web Application (Model-View-Controller), среда Visual Studio создает файл `site.css` в папке `wwwroot/css`. В текущей главе данный файл не задействован.

Назначим файлу имя `styles.css`, щелкнем на кнопке Add (Добавить), чтобы создать файл таблицы стилей, и приведем его содержимое к виду, показанному в листинге 2.22.

Листинг 2.22. Содержимое файла `styles.css` из папки `wwwroot/css`

```
.field-validation-error { color: #f00; }
.field-validation-valid { display: none; }
.input-validation-error { border: 1px solid #f00;
                          background-color: #fee; }
.validation-summary-errors { font-weight: bold; color: #f00; }
.validation-summary-valid { display: none; }
```

Чтобы применить эту таблицу стилей, мы добавили элемент `link` в раздел `head` представления `RsvpForm` (листинг 2.23).

Листинг 2.23. Применение таблицы стилей в файле `RsvpForm.cshtml`

```
...
<head>
  <meta name="viewport" content="width=device-width" />
  <title>RsvpForm</title>
  <link rel="stylesheet" href="/css/styles.css" />
</head>
...
```

Элемент `link` использует атрибут `href` для указания местоположения таблицы стилей. Обратите внимание, что папка `wwwroot` в URL опущена. Стандартная конфигурация ASP.NET включает поддержку обслуживания статического содержимого, такого как изображения, таблицы стилей CSS и файлы JavaScript, которая автоматически отображает запросы на папку `wwwroot`. Процесс конфигурации ASP.NET и MVC рассматривается в главе 14.

Совет. Для работы с таблицами стилей предусмотрена специальная вспомогательная функция дескриптора, которая может оказаться полезной при наличии многочисленных файлов, требующих управления. Подробности ищите в главе 25.

Благодаря применению таблицы стилей в случае отправки данных, которые вызывают ошибки проверки достоверности, отображается визуально более ясное сообщение об ошибках (рис. 2.21).

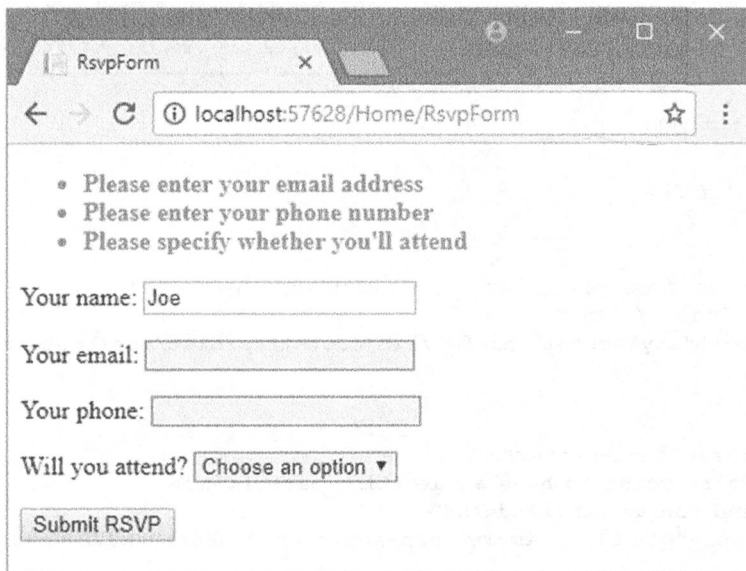


Рис. 2.21. Автоматическая подсветка полей с ошибками проверки достоверности

Стилизация содержимого

Все цели приложения, касающиеся функциональности, достигнуты, но его общий вид оставляет желать лучшего. Когда вы создаете проект с использованием шаблона Web Application (Model-View-Controller), как в рассматриваемом примере, среда Visual Studio устанавливает несколько распространенных пакетов для разработки на стороне клиента. Хотя я не являюсь сторонником применения шаблонов проектов, мне нравятся выбранные Microsoft библиотеки клиентской стороны. Одна из них называется Bootstrap и представляет собой удобную инфраструктуру CSS, первоначально разработанную в Twitter, которая постепенно превратилась в крупный проект с открытым кодом и стала главной опорой разработки веб-приложений.

На заметку! На момент написания книги текущей версией была Bootstrap 3, но версия Bootstrap 4 находилась на стадии разработки. В Microsoft могут принять решение обновить версию Bootstrap, используемую шаблоном Web Application, в последующих выпусках Visual Studio, что может привести к отображению содержимого по-другому. В других главах книги это не должно стать проблемой, т.к. там будет показано, каким образом явно указывать версию пакета, чтобы получить ожидаемые результаты.

Стилизация начального представления

Базовые средства Bootstrap работают за счет применения классов к элементам, которые соответствуют селекторам CSS, определенным внутри добавленных в папку `wwwroot/lib/bootstrap` файлов. Подробную информацию о классах, определенных в библиотеке Bootstrap, можно получить на веб-сайте <http://getbootstrap.com>, а в листинге 2.24 демонстрируется использование нескольких базовых стилей в представлении `MyView.cshtml`.

Листинг 2.24. Добавление классов Bootstrap в файле MyView.cshtml из папки Views/Home

```

@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css"
/>
</head>
<body>
    <div class="text-center">
        <h3>We're going to have an exciting party!</h3>
        <h4>And you are invited</h4>
        <a class="btn btn-primary" asp-action="RsvpForm">RSVP Now</a>
    </div>
</body>
</html>

```

В разметку был добавлен элемент `link`, атрибут `href` которого загружает файл `bootstrap.css` из папки `wwwroot/lib/bootstrap/dist/css`. По соглашению пакеты CSS и JavaScript от независимых поставщиков устанавливаются в папку `wwwroot/lib`, и в главе 6 будет описан инструмент, предназначенный для управления такими пакетами.

После импортирования таблиц стилей Bootstrap осталось стилизовать элементы. Рассматриваемый пример прост, поэтому требует использования лишь небольшого числа классов CSS из Bootstrap: `text-center`, `btn` и `btn-primary`.

Класс `text-center` центрирует содержимое элемента и его дочерних элементов. Класс `btn` стилизует элемент `button`, `input` или `a` в виде симпатичной кнопки, а класс `btn-primary` указывает диапазон цветов для кнопки. Запустив приложение, можно увидеть результат, показанный на рис. 2.22.

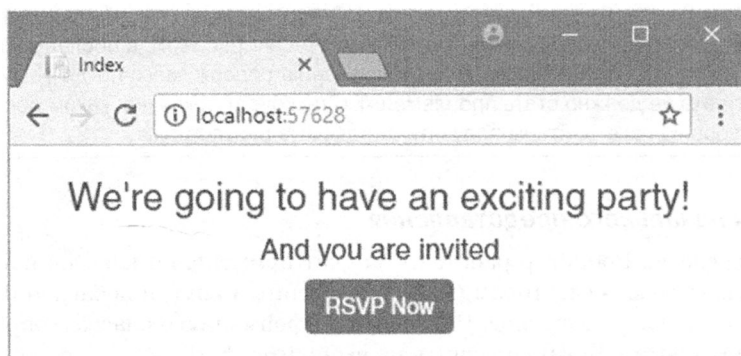


Рис. 2.22. Стилизация представления

Совершенно очевидно, что я — не веб-дизайнер. На самом деле, будучи еще ребенком, я был освобожден от уроков рисования по причине полного отсутствия таланта. Это произвело благоприятный эффект в виде того, что я стал уделять больше времени урокам математики, но вместе с тем мои художественные навыки не развивались примерно с десятилетнего возраста. Для реальных проектов я обратился бы к помощи профессионального дизайнера содержимого, но в настоящем примере я собираюсь делать все самостоятельно и применять Bootstrap с максимально возможной сдержанностью и согласованностью, на какую только способен.

Стилизация представления *RsvpForm*

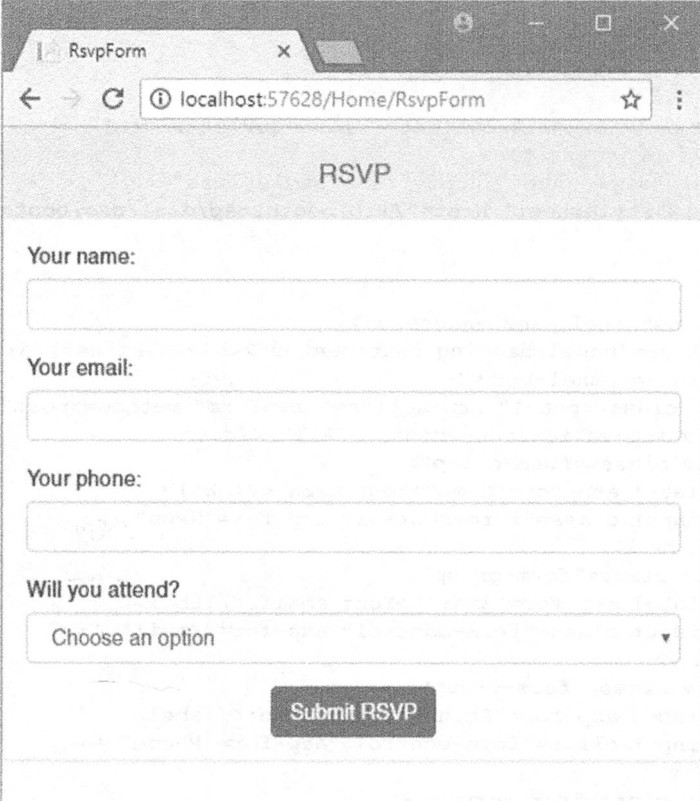
В библиотеке Bootstrap определены классы, которые могут использоваться для стилизации форм. Я не планирую вдаваться в особые детали, но в листинге 2.25 показано, как были применены эти классы.

Листинг 2.25. Добавление классов Bootstrap в файле *RsvpForm.cshtml* из папки *Views/Home*

```
@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
    <link rel="stylesheet" href="/css/styles.css" />
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
    <div class="panel panel-success">
        <div class="panel-heading text-center"><h4>RSVP</h4></div>
        <div class="panel-body">
            <form class="p-a-1" asp-action="RsvpForm" method="post">
                <div asp-validation-summary="All"></div>
                <div class="form-group">
                    <label asp-for="Name">Your name:</label>
                    <input class="form-control" asp-for="Name" />
                </div>
                <div class="form-group">
                    <label asp-for="Email">Your email:</label>
                    <input class="form-control" asp-for="Email" />
                </div>
                <div class="form-group">
                    <label asp-for="Phone">Your phone:</label>
                    <input class="form-control" asp-for="Phone" />
                </div>
                <div class="form-group">
                    <label>Will you attend?</label>
                    <select class="form-control" asp-for="WillAttend">
```

```
        <option value="">Choose an option</option>
        <option value="true">Yes, I'll be there</option>
        <option value="false">No, I can't come</option>
    </select>
</div>
<div class="text-center">
    <button class="btn btn-primary" type="submit">
        Submit RSVP
    </button>
</div>
</form>
</div>
</div>
</body>
</html>
```

Классы Bootstrap в данном примере создают заголовок, просто чтобы придать компоновке структурированность. Для стилизации формы используется класс `form-group`, который стилизует элемент, содержащий `label` и связанный элемент `input` или `select`. Результаты стилизации можно видеть на рис. 2.23.



The screenshot shows a web browser window with the title "RsvpForm" and the address bar displaying "localhost:57628/Home/RsvpForm". The page content is a form titled "RSVP" with the following elements:

- A header section with the text "RSVP".
- A label "Your name:" followed by a text input field.
- A label "Your email:" followed by a text input field.
- A label "Your phone:" followed by a text input field.
- A label "Will you attend?" followed by a dropdown menu with the text "Choose an option".
- A "Submit RSVP" button at the bottom.

Рис. 2.23. Стилизация представления RsvpForm

Стилизация представления *Thanks*

Следующим стилизуемым представлением является `Thanks.cshtml`; в листинге 2.26 показано, как это делается с применением классов CSS, подобных тем, которые использовались для других представлений. Чтобы облегчить управление приложением, имеет смысл везде, где только возможно, избегать дублирования кода и разметки. Инфраструктура MVC предлагает несколько средств, помогающих сократить дублирование, которые рассматриваются в последующих главах. К таким средствам относятся компоновки Razor (глава 5), частичные представления (глава 21) и компоненты представлений (глава 22).

Листинг 2.26. Применение классов Bootstrap в файле `Thanks.cshtml` из папки `Views/Home`

```
@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body class="text-center">
    <p>
        <h1>Thank you, @Model.Name!</h1>
        @if (Model.WillAttend == true) {
            @:It's great that you're coming. The drinks are already in the fridge!
        } else {
            @:Sorry to hear that you can't make it, but thanks for letting us know.
        }
    </p>
    Click <a class="nav-link" asp-action="ListResponses">here</a>
    to see who is coming.
</body>
</html>
```

На рис. 2.24 показан результат стилизации.

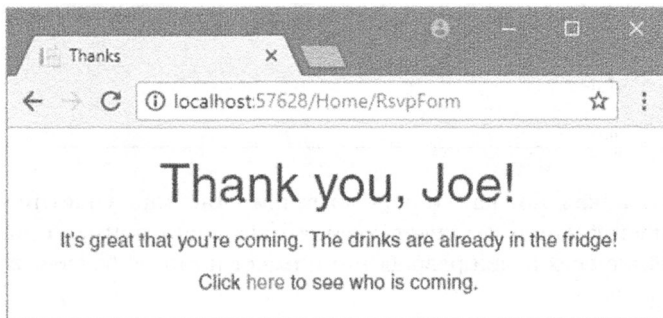


Рис. 2.24. Стилизация представления `Thanks`

Стилизация представления *ListResponses*

Последним мы стилизуем представление `ListResponses`, которое отображает список участников вечеринки. Стилизация содержимого следует тому же базовому подходу, который применялся в отношении всех стилей Bootstrap (листинг 2.27).

Листинг 2.27. Добавление классов Bootstrap в файле `ListResponses.cshtml` из папки `Views/Home`

```
@model IEnumerable<PartyInvites.Models.GuestResponse>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
    <title>Responses</title>
</head>
<body>
    <div class="panel-body">
        <h2>Here is the list of people attending the party</h2>
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Email</th>
                    <th>Phone</th>
                </tr>
            </thead>
            <tbody>
                @foreach (PartyInvites.Models.GuestResponse r in Model) {
                    <tr>
                        <td>@r.Name</td>
                        <td>@r.Email</td>
                        <td>@r.Phone</td>
                    </tr>
                }
            </tbody>
        </table>
    </div>
</body>
</html>
```

На рис. 2.25 показано, как теперь выглядит таблица участников вечеринки. Добавление стилей к данному представлению завершает пример приложения, которое теперь достигло всех целей разработки и имеет намного более совершенный внешний вид.

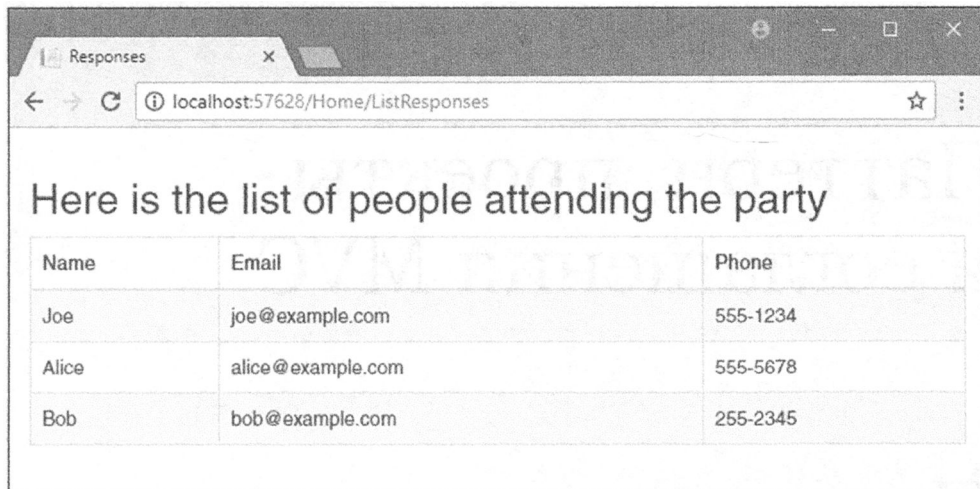


Рис. 2.25. Стилизация представления `ListResponses`

Резюме

В главе был создан новый проект MVC, который использовался для построения простого приложения ввода данных MVC, что позволило получить первое представление об архитектуре ASP.NET Core MVC и применяемом подходе. Некоторые основные средства (включая синтаксис Razor, маршрутизацию и тестирование) не рассматривались, но мы вернемся к данным темам в последующих главах. В следующей главе будет описаны паттерны проектирования MVC, которые формируют основу эффективной разработки с помощью ASP.NET Core MVC.