Проверка достоверности моделей

Реальность же такова, что пользователи часто вводят данные, которые не являются допустимыми и не могут использоваться, а потому темой настоящей главы будет проверка достоверности моделей.

Проверка достоверности моделей представляет собой процесс, который обеспечивает пригодность данных, получаемых приложением, для привязки моделей. Если данные не подходят, то пользователям предоставляется полезная информация, которая помогает устранить проблему.

Первая часть процесса — проверка полученных данных — является одним из основных способов предохранить целостность модели предметной области. Отклонение данных, которые не имеют смысла в контексте предметной области, может предотвратить возникновение странных и нежелательных состояний в приложении. Вторая часть — помощь пользователям в корректировке проблемы — важна в равной степени. Без такой информации и обратной связи, необходимой для взаимодействия с приложением, пользователи будут недовольны и озадачены. В общедоступных приложениях это означает, что пользователи просто прекратят ими пользоваться. В корпоративных приложениях это означает нарушение рабочего потока пользователей. Ни тот, ни другой исход не является желательным, но к счастью инфраструктура MVC предоставляет широкую поддержку проверки достоверности моделей.

В табл. 27.1 приведена сводка, позволяющая поместить проверку достоверности моделей в контекст.

В табл. 27.2 приведена сводка по главе.

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `ModelValidation` с применением шаблона ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core. Чтобы включить инфраструктуру MVC и другие компоненты промежуточного ПО, внесем в класс `Startup` изменения, показанные в листинге 27.1.

Таблица 27.1. Помещение проверки достоверности моделей в контекст

Вопрос	Ответ
Что это такое?	Проверка достоверности моделей — это процесс выяснения, что данные, предоставленные в запросе, пригодны для применения в приложении
Чем она полезна?	Пользователи не всегда вводят допустимые данные. Использование таких данных в приложении может приводить к неожиданным и нежелательным последствиям
Как она используются?	Контроллеры исследуют результат процесса проверки достоверности, а посредством вспомогательных функций дескрипторов в представления, отображаемые пользователям, включается отклик проверки. Проверка достоверности автоматически выполняется во время процесса привязки моделей. Она обычно дополняется специальной проверкой в классе контроллера или за счет применения атрибутов проверки достоверности
Существуют ли какие-то скрытые ловушки или ограничения?	Важно протестировать эффективность кода проверки достоверности, убедившись в том, что он способен справляться с полным диапазоном значений, которые может получать приложение
Существуют ли альтернативы?	Нет, проверка достоверности моделей тесно интегрирована в ASP.NET Core MVC

Таблица 27.2. Сводка по главе

Задача	Решение	Листинг
Явная проверка достоверности модели	Используйте объект `ModelState` для регистрации ошибок проверки достоверности	27.9, 27.10
Генерация сводки по ошибкам проверки достоверности	Примените атрибут `asp-validation-summary` к элементу `div`	27.11
Изменение стандартных сообщений привязки моделей	Переопределите функции сообщений в поставщике сообщений привязки моделей	27.12
Генерация ошибок проверки достоверности на уровне свойств	Примените атрибут `asp-validation-for` к элементу `span`	27.13
Генерация ошибок проверки достоверности на уровне модели	Используйте объект `ModelState` для регистрации ошибок проверки достоверности, которые не ассоциированы со специфическими свойствами, и укажите значение `ModelOnly` для атрибута `asp-validation-summary` в элементе `div`	27.14, 27.15
Определение самопроверяемой модели	Примените атрибуты проверки достоверности данных к свойствам модели	27.16, 27.17
Создание специального атрибута проверки достоверности	Реализуйте интерфейс `IModelValidator`	27.18, 27.19
Выполнение проверки достоверности на стороне клиента	Используйте пакеты ненавязчивой проверки достоверности jQuery	27.20, 27.21
Выполнение удаленной проверки достоверности	Определите метод действия для выполнения проверки достоверности и примените атрибут `Remote` к свойству модели	27.22, 27.23

Листинг 27.1. Содержимое файла `Startup.cs` из папки `ModelValidation`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ModelValidation {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Создание модели

Создадим папку `Models` и добавим в нее файл класса по имени `Appointment.cs` с определением, приведенным в листинге 27.2.

Листинг 27.2. Содержимое файла `Appointment.cs` из папки `Models`

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models {
  public class Appointment {

    public string ClientName { get; set; }

    [UIHint("Date")]
    public DateTime Date { get; set; }

    public bool TermsAccepted { get; set; }
  }
}
```

В классе `Appointment` определены три свойства и с помощью атрибута `UIHint` указано, что свойство `Date` должно выражаться как дата без компонента времени.

Создание контроллера

Создадим папку `Controllers`, добавим файл класса по имени `HomeController.cs` и поместим в него определение контроллера из листинга 27.3, который оперирует с классом модели `Appointment`.

Листинг 27.3. Содержимое файла HomeController.cs из папки Controllers

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

namespace ModelValidation.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() =>
      View("MakeBooking", new Appointment { Date = DateTime.Now });

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) =>
      View("Completed", appt);
  }
}
```

Действие `Index` визуализирует представление `MakeBooking` с новым объектом `Appointment` в качестве модели представления. Метод действия `MakeBooking()` более интересен, т.к. в нем будет выполняться проверка достоверности модели.

На заметку! Пример приложения настолько прост, что в нем даже не определено хранилище и не добавлен код для сохранения объектов `Appointment`, которые выпускаются процессом привязки моделей. Тем не менее, важно иметь в виду, что главной причиной проверки достоверности модели является предотвращение попадания в хранилище неправильных или бессмысленных данных с последующим возникновением проблем (либо при попытке сохранения данных, либо при попытке их обработки в более позднее время).

Создание компоновки и представлений

Для ряда примеров, приводимых в главе, требуется простая компоновка. Создадим папку `Views/Shared` и добавим в нее файл `_Layout.cshtml`, содержимое которого показано в листинге 27.4.

Листинг 27.4. Содержимое файла _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>Model Validation</title>
  <link asp-href-include=
    "/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
  @RenderSection("scripts", false)
</head>
<body class="m-1 p-1">
  @RenderBody()
</body>
</html>
```

Чтобы снабдить методы действий представлениями, создадим папку `Views/Home` и добавим в нее файл по имени `MakeBooking.cshtml` с разметкой из листинга 27.5.

Листинг 27.5. Содержимое файла `MakeBooking.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</h2>
</div>

<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Когда форма, содержащаяся в файле `Index.cshtml`, отправляется обратно приложению, метод действия `MakeBooking()` отображает детали созданной пользователем встречи с применением представления `Completed.cshtml` из папки `Views/Home` (листинг 27.6).

Листинг 27.6. Содержимое файла `Completed.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

<div class="bg-success m-1 p-1 text-white"><h2>Your Appointment</h2>
  </div>

<table class="table table-bordered">
  <tr>
    <th>Your name is:</th>
    <td>@Model.ClientName</td>
  </tr>
  <tr>
    <th>Your appontment date is:</th>
    <td>@Model.Date.ToString("d")</td>
  </tr>
</table>
<a class="btn btn-success" asp-action="Index">Make Another Appointment</a>
```

При стилизации HTML-элементов представления полагаются на CSS-пакет Bootstrap. Создадим в корневой папке проекта файл `bower.json` с использованием шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел `dependencies` (листинг 27.7). Добавим также пакет jQuery, который понадобится позже в главе.

Листинг 27.7. Добавление пакета Bootstrap в файле `bower.json` из папки `ModelValidation`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

Последний подготовительный шаг связан с созданием файла `_ViewImports.cshtml` в папке `Views`, в котором настраиваются встроенные вспомогательные функции дескрипторов для применения в представлениях Razor и импортируется пространство имен модели (листинг 27.8).

Листинг 27.8. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@using ModelValidation.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Как вы уже наверняка догадались, пример основан на создании встреч. Взглянуть на него в работе можно, запустив приложение и запросив стандартный URL. Ввод в элементах формы информации о встрече и щелчок на кнопке Make Booking (Создать встречу) приведет к отправке данных серверу, который выполнит процесс привязки моделей для создания объекта `Appointment`, детали которого затем визуализируются с использованием представления `Completed.cshtml` (рис. 27.1).

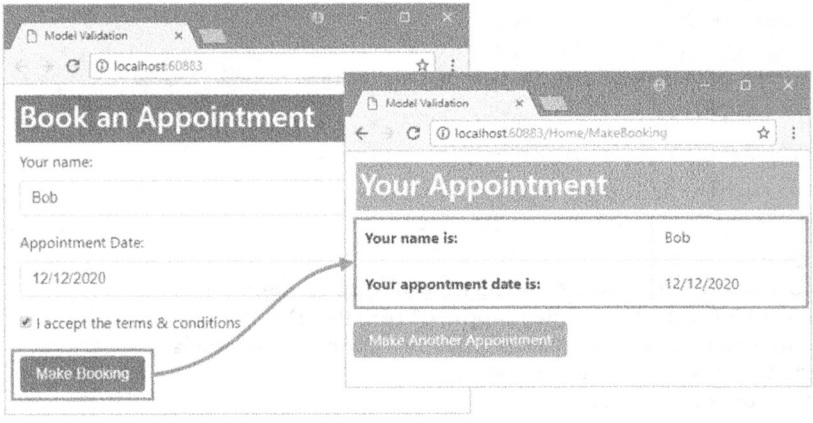

Рис. 27.1. Выполнение примера приложения

Необходимость в проверке достоверности модели

Проверка достоверности модели — это процесс принудительного применения требований, которые приложение налагает на данные, получаемые от клиентов. Без проверки достоверности приложение будет пытаться оперировать с любыми полученными данными, что может привести к генерации исключений и непредвиденному поведению, которое отразит немедленные или долговременные проблемы, постепенно накапливаемые по мере того, как хранилище наполняется неправильными, незавершенными или злонамеренными данными.

В настоящее время пример приложения будет принимать любые данные, которые отправляет пользователь. Чтобы предохранить целостность приложения и модели предметной области, должны быть удовлетворены три перечисленных ниже условия, прежде чем станет известно, что пользователь предоставляет приемлемый объект `Appointment`:

- пользователь должен предоставить имя;
- пользователь должен предоставить дату, относящуюся к будущему;
- пользователь должен отметить флажок для принятия условий.

В последующих разделах демонстрируется использование проверки достоверности моделей для навязывания указанных требований за счет контроля получаемых приложением данных и предоставления пользователям обратной связи, когда приложение не может работать с данными, которые они отправили.

Явная проверка достоверности модели

Самый прямой способ проверки достоверности модели предусматривает ее выполнение в методе действия. В листинге 27.9 показан новый метод действия `MakeBooking()` с добавленными явными проверками для каждого свойства, определяемого классом `Appointment`.

Листинг 27.9. Явная проверка достоверности модели в файле `HomeController.cs` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() =>
      View("MakeBooking", new Appointment { Date = DateTime.Now });

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) {
      if (string.IsNullOrEmpty(appt.ClientName)) {
        ModelState.AddModelError(nameof(appt.ClientName),
          "Please enter your name");
        // Введите свое имя
      }
```

```
    if (ModelState.GetValidationState("Date")
       == ModelValidationState.Valid && DateTime.Now > appt.Date) {
      ModelState.AddModelError(nameof(appt.Date),
        "Please enter a date in the future");
       // Введите дату, относящуюся к будущему
    }
    if (!appt.TermsAccepted) {
      ModelState.AddModelError(nameof(appt.TermsAccepted),
        "You must accept the terms");
       // Вы должны принять условия
    }
    if (ModelState.IsValid) {
      return View("Completed", appt);
    } else {
      return View();
    }
  }
}
```

В коде проверяются значения, которые связыватель модели присвоил свойствам объекта параметра. Любые обнаруженные ошибки регистрируются с применением объекта `ModelStateDictionary`, который возвращается свойством `ModelState`, унаследованным из базового класса `Controller`.

Как подсказывает его имя, класс `ModelStateDictionary` — это словарь, который используется для отслеживания деталей состояния объекта модели, с акцентом на ошибках проверки достоверности. В табл. 27.3 описаны наиболее важные члены класса `ModelStateDictionary`.

Таблица 27.3. Избранные члены класса `ModelStateDictionary`

Имя	Описание
`AddModelError(property, message)`	Этот метод применяется для регистрации ошибки проверки достоверности модели, связанной с указанным свойством
`GetValidationState(property)`	Этот метод используется при выяснении, есть ли ошибки проверки достоверности модели для специфического свойства; результат выражается как значение перечисления `ModelValidationState`
`IsValid`	Это свойство возвращает `true`, если все свойства модели допустимы, и `false` в противном случае

В качестве примера применения `ModelStateDictionary` ниже демонстрируется выполнение проверки достоверности свойства `ClientName`:

```
...
if (string.IsNullOrEmpty(appt.ClientName)) {
  ModelState.AddModelError(nameof(appt.ClientName),
  "Please enter your name");
}
...
```

Одна из целей приведенного примера связана с гарантированием того, что пользователь предоставляет значение для данного свойства, поэтому здесь используется статический метод `string.IsNullOrEmpty()` для проверки значения свойства, которое процесс привязки моделей извлек из запроса. Если свойство `ClientName` равно `null` или пустой строке, тогда известно, что цель проверки достоверности не была достигнута. В таком случае посредством метода `ModelState.AddModelError()` регистрируется ошибка проверки достоверности с указанием имени свойства (`ClientName`) и сообщения, которое будет отображаться пользователю для объяснения природы проблемы (`Please enter your name` (Введите свое имя)).

Система привязки моделей также применяет объект `ModelStateDictionary` для регистрации любых проблем с нахождением и присваиванием значений свойствам модели. Метод `GetValidationState()` используется с целью выяснения, зарегистрированы ли для свойства модели какие-то ошибки процессом привязки моделей или вызовом `AddModelError()` во время явной проверки достоверности в методе действия. Метод `GetValidationState()` возвращает значение перечисления `ModelValidationState`, которое описано в табл. 27.4.

Таблица 27.4. Значения перечисления `ModelValidationState`

Имя	Описание
`Unvalidated`	Указывает, что в отношении свойства модели проверка достоверности не выполнялась, обычно из-за отсутствия в запросе значения, которое бы соответствовало имени свойства
`Valid`	Указывает, что значение в запросе, ассоциированное со свойством, является допустимым
`Invalid`	Указывает, что значение в запросе, ассоциированное со свойством, является недопустимым и применяться не должно
`Skipped`	Указывает, что свойство модели не было обработано. Обычно это говорит о наличии настолько большого количества ошибок проверки достоверности, что продолжать проверку не имеет смысла

Для свойства `Date` выполняется проверка, сообщил ли процесс привязки моделей о проблеме во время преобразования отправленного браузером значения в объект `DateTime`:

```
...
if (ModelState.GetValidationState("Date") == ModelValidationState.Valid
    && DateTime.Now > appt.Date) {
  ModelState.AddModelError(nameof(appt.Date),
  "Please enter a date in the future");
}
...
```

Цель проверки достоверности для свойства `Date` — обеспечить предоставление пользователем допустимой даты в будущем. Метод `GetValidationState()` используется для выяснения, смог ли процесс привязки моделей преобразовать значение из запроса в объект `DateTime`, за счет проверки на предмет равенства `ModelValidationState.Valid`. В случае допустимой даты выполняется проверка, относится ли она к будущему, и если нет, то посредством метода `AddModelError()` регистрируется проблема проверки достоверности.

После того, как все свойства объекта модели проверены, с помощью свойства ModelState.IsValid выясняется, возникали ли ошибки. Данное свойство возвращает true, если во время проверок вызывался метод Model.State.AddModelError() или у связывателя модели возникали проблемы с созданием объекта Appointment:

```
...
if (ModelState.IsValid) {
  return View("Completed", appt);
} else {
  return View();
}
...
```

Объект Appointment является допустимым, если свойство IsValid возвращает true, в случае чего метод действия визуализирует представление Completed.cshtml. Если же свойство IsValue возвращает false, тогда есть проблема проверки достоверности, которая обрабатывается путем вызова метода View() для визуализации стандартного представления.

Отображение пользователю ошибок проверки достоверности

Иметь дело с ошибкой проверки достоверности, вызывая метода View(), может показаться странным подходом, но данные контекста, которыми инфраструктура MVC снабжает представление, содержат детали ошибок, возникших при проверке достоверности модели. Такие детали автоматически обнаруживаются и применяются вспомогательной функцией дескриптора, которая используется для трансформации элементов input.

Чтобы посмотреть, как все работает, запустим приложение и щелкнем на кнопке Make Booking, не заполняя форму данными о встрече. Визуальных изменений в окне браузера не будет, но если просмотреть HTML-разметку, которую MVC возвращает из запроса POST, то можно заметить, что изменился атрибут class элемента формы. Вот как выглядел элемент ClientName до отправки формы:

```
<input class="form-control" type="text" id="ClientName"
       name="ClientName" value="">
```

А после отправки пустой формы элемент ClientName принимает следующий вид:

```
<input class="form-control input-validation-error"
       type="text" id="ClientName" name="ClientName" value="">
```

Вспомогательная функция дескриптора добавляет элементы, чьи значения не прошли проверку достоверности, в класс input-validation-error, который затем можно стилизовать с целью выделения проблемы.

Делать это можно за счет определения специальных стилей CSS в таблице стилей, но если нужно задействовать встроенные стили для проверки, предоставляемые CSS-библиотеками вроде Bootstrap, то придется проделать небольшую дополнительную работу. Поскольку имя класса, добавляемое к элементам формы, изменять нельзя, потребуется код JavaScript для сопоставления имени, применяемого инфраструктурой MVC, и классами ошибок CSS, предлагаемыми библиотекой Bootstrap.

Совет. Использовать код JavaScript подобного рода может быть неудобно, из-за чего возникает соблазн применять специальные стили CSS даже при работе с CSS-библиотеками типа Bootstrap. Однако цвета, используемые классами проверки достоверности в Bootstrap, можно переопределять посредством тем или путем настройки пакета и определения собственных стилей, т.е. вам придется обеспечить соответствие между любыми изменениями в теме и изменениями в любых специальных стилях, которые вы определили. В идеальном случае разработчики из Microsoft сделают имена классов проверки достоверности конфигурируемыми в будущем выпуске ASP.NET Core MVC, но до тех пор написание кода JavaScript для применения стилей Bootstrap является более надежным подходом, чем создание специальных таблиц стилей.

В листинге 27.10 к представлению `MakeBooking` добавляется код jQuery для поиска элементов в классе `input-validation-error`, нахождения ближайшего родительского элемента, который был назначен классу `form-group`, и добавления этого элемента в класс `has-danger` (используемый библиотекой Bootstrap для установки цвета ошибки в элементах формы).

Листинг 27.10. Назначение элементов классам проверки достоверности в файле `MakeBooking.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

@section scripts {
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script type="text/javascript">
    $(document).ready(function () {
      $("input.input-validation-error")
        .closest(".form-group").addClass("has-danger");
    });
  </script>
}

<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</h2>
</div>

<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Добавленный код jQuery выполняется, когда браузер завершает разбор всех элементов в HTML-документе, и результатом будет выделение элементов input, которые назначены классу input-validaton-error. Запустив приложение и отправив форму без заполнения всех полей, можно получить результат, изображенный на рис. 27.2.

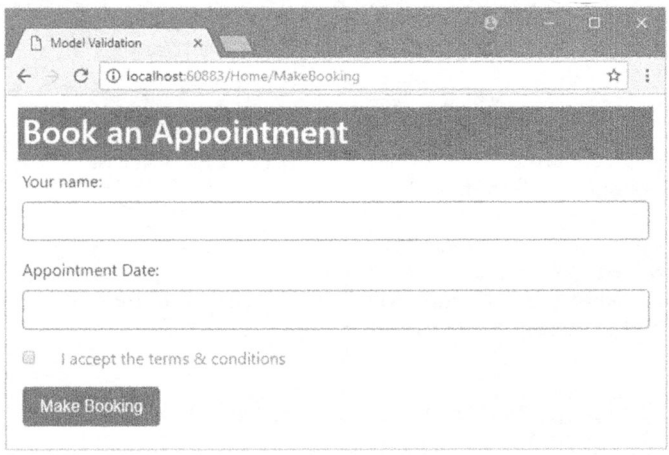

Рис. 27.2. Выделение ошибок проверки достоверности

Если форма отправляется без ввода каких-либо данных, то все три свойства выделяются как содержащие ошибки. Пользователь не увидит представление Completed.cshtml до тех пор, пока форма не будет отправлена с данными, которые могут быть разобраны связывателем модели и успешно проходят явные проверки достоверности в методе MakeBooking(). Пока подобное не произойдет, отправка формы будет приводить к визуализации представления MakeBooking.cshtml с текущими ошибками проверки достоверности.

Отображение сообщений об ошибках проверки достоверности

Классы CSS, которые вспомогательные функции дескрипторов применяют к элементам input, указывают на наличие проблемы с полем формы, но не сообщают пользователю, в чем конкретно состоит проблема. Предоставление пользователю дополнительной информации требует использования другой вспомогательной функции дескриптора, которая добавляет в представление сводку по проблемам (листинг 27.11).

Листинг 27.11. Отображение сводки по проверке достоверности в файле MakeBooking.cshtml из папки Views/Home

```
@model Appointment
@{ Layout = "_Layout"; }
@section scripts {
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script type="text/javascript">
    $(document).ready(function () {
      $("input.input-validation-error")
        .closest(".form-group").addClass("has-danger");
    });
  </script>
}
```

```
<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</h2>
</div>
<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div asp-validation-summary="All" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Класс `ValidationSummaryTagHelper` обнаруживает атрибут `asp-validation-summary` в элементах `div` и реагирует добавлением сообщений, которые описывают любые ошибки проверки достоверности, выявленные методом действия. В атрибуте `asp-validation-summary` указывается одно из значений перечисления `ValidationSummary`, которые описаны в табл. 27.5 и вскоре будут продемонстрированы в работе.

Таблица 27.5. Значения перечисления `ValidationSummary`

Имя	Описание
`All`	Применяется для отображения всех ошибок проверки достоверности, которые были зарегистрированы
`ModelOnly`	Используется для отображения только ошибок проверки достоверности, относящихся ко всей модели, за исключением тех, которые были зарегистрированы для индивидуальных свойств, как описано в разделе "Отображение сообщений об ошибках проверки достоверности на уровне модели" далее в главе
`None`	Применяется для отключения вспомогательной функции дескриптора, так что она не будет трансформировать HTML-элемент

Если запустить приложение и отправить форму, не внося какие-либо изменения, то появится сводка, которую сгенерировала вспомогательная функция дескриптора. Цвет текста в рассматриваемом примере определяется классом `text-danger` из Bootstrap, который гарантирует, что цвет текста соответствует цвету, используемому для выделения текстовых полей (рис. 27.3).

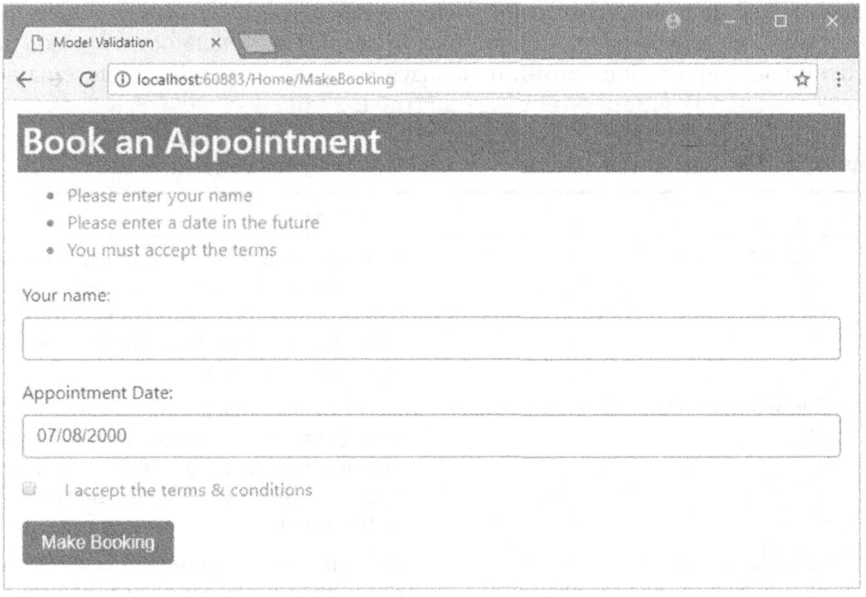

Рис. 27.3. Отображение пользователю сводки по проверке достоверности

Заглянув в HTML-разметку, которая была получена браузером, легко заметить, что сообщения проверки достоверности отправлялись в виде списка:

```
<div class="text-danger validation-summary-errors"
     data-valmsg-summary="true">
  <ul>
    <li>Please enter your name</li>
    <li>Please enter a date in the future</li>
    <li>You must accept the terms</li>
  </ul>
</div>
```

Конфигурирование стандартных сообщений об ошибках проверки достоверности

Процесс привязки моделей, описанный в главе 26, выполняет собственную проверку достоверности, когда он пытается предоставить значения данных, требующиеся для вызова метода действия. Чтобы увидеть, как это работает, запустим приложение, очистим содержимое поля Appointment Date (Дата встречи) и отправим форму. Вы обнаружите, что одно из отображаемых сообщений проверки достоверности изменилось, и нового сообщения нет среди строк, передаваемых методу `AddModelError()` внутри метода действия:

```
The value '' is invalid
```

Показанное сообщение добавляется в `ModelStateDictionary` процессом привязки моделей, когда он не может найти значение для свойства или находит значение, но не в состоянии его преобразовать. В данном случае ошибка возникла из-за отправки в данных формы пустой строки, которая не может быть преобразована в объект `DateTime` для свойства `Date` класса `Appointment`.

Связыватель модели располагает набором предопределенных сообщений, которые он применяет для обозначения ошибок проверки достоверности. Их можно заменить специальными сообщениями, используя методы, которые определены в классе `DefaultModelBindingMessageProvider` (табл. 27.6).

Таблица 27.6. Методы класса `DefaultModelBindingMessageProvider`

Имя	Описание
`SetValueMustNotBeNullAccessor()`	Функция, передаваемая этому методу, используется для генерации сообщения об ошибке проверки достоверности, когда свойству модели, не допускающему `null`, поступает значение `null`
`SetMissingBindRequiredValueAccessor()`	Функция, передаваемая этому методу, применяется для генерации сообщения об ошибке проверки достоверности, когда запрос не содержит значение для обязательного свойства
`SetMissingKeyOrValueAccessor()`	Функция, передаваемая этому методу, используется для генерации сообщения об ошибке проверки достоверности, когда данные, требующиеся объекту словаря, содержат ключи или значения, равные `null`
`SetAttemptedValueIsInvalidAccessor()`	Функция, передаваемая этому методу, применяется для генерации сообщения об ошибке проверки достоверности, когда системе привязки моделей не удается преобразовать значение данных в требуемый тип C#
`SetUnknownValueIsInvalidAccessor()`	Функция, передаваемая этому методу, используется для генерации сообщения об ошибке проверки достоверности, когда системе привязки моделей не удается преобразовать значение данных в требуемый тип C#
`SetValueMustBeANumberAccessor()`	Функция, передаваемая этому методу, применяется для генерации сообщения об ошибке проверки достоверности, когда значение данных не может быть преобразовано в числовой тип C#
`SetValueIsInvalidAccessor()`	Функция, передаваемая этому методу, используется для генерации запасного сообщения об ошибке проверки достоверности, которое применяется в качестве последнего средства

Каждый метод, описанный в табл. 27.6, получает функцию, которая вызывается для получения сообщения об ошибке, подлежащего отображению. Эти методы используются в классе `Startup` для конфигурирования приложения, как показано в листинге 27.12, где производится замена стандартного сообщения, которое отображается в случае значения `null`.

Листинг 27.12. Замена функции генерации сообщений при привязке моделей в файле `Startup.cs` из папки `ModelValidation`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ModelValidation {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc().AddMvcOptions(opts =>
        opts.ModelBindingMessageProvider
            .SetValueMustNotBeNullAccessor(value => "Please enter a value")
      );
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Указанная функция получает значение, предоставленное пользователем, хотя это не особенно удобно, когда приходится иметь дело со значениями `null`. Чтобы увидеть специальное сообщение, понадобится снова запустить приложение и отправить форму, очистив поле Appointment Date (рис. 27.4).

Рис. 27.4. Изменение сообщений об ошибках привязки моделей

Отображение сообщений об ошибках проверки достоверности на уровне свойств

Несмотря на то что специальное сообщение об ошибке более выразительно, чем стандартное сообщение, польза от него по-прежнему невелика, т.к. оно не указывает ясно пользователю на проблему. Для ошибок такого рода отображать сообщения об ошибках проверки достоверности более практично рядом с HTML-элементами, которые содержат проблемные данные, что можно делать с применением класса вспомогательной функции дескриптора `ValidationMessageTag`. Он ищет элементы `span` с атрибутом `asp-validation-for`, используемым для указания свойства модели, к которому должны относиться отображаемые сообщения об ошибках.

В листинге 27.13 для каждого элемента `input` внутри формы добавляются элементы сообщений об ошибках проверки достоверности на уровне свойств. Кроме того, удален раздел `scripts`, поскольку индивидуальные сообщения об ошибках проверки достоверности обеспечат достаточное выделение, чтобы можно было понять, какие элементы содержат ошибки.

Листинг 27.13. Добавление сообщений об ошибках проверки достоверности на уровне свойств в файле `MakeBooking.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

...

<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</h2>
</div>

<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div asp-validation-summary="All" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <div><span asp-validation-for="ClientName" class="text-danger"></span>
    </div>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <div><span asp-validation-for="Date" class="text-danger"></span></div>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <span asp-validation-for="TermsAccepted" class="text-danger"></span>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Так как элементы span отображаются внутри, нужно позаботиться о том, чтобы было вполне очевидно, к каким элементам относятся сообщения об ошибках проверки достоверности. Запустив приложение и отправив форму без ввода каких-нибудь данных, можно увидеть новые сообщения об ошибках проверки достоверности (рис. 27.5).

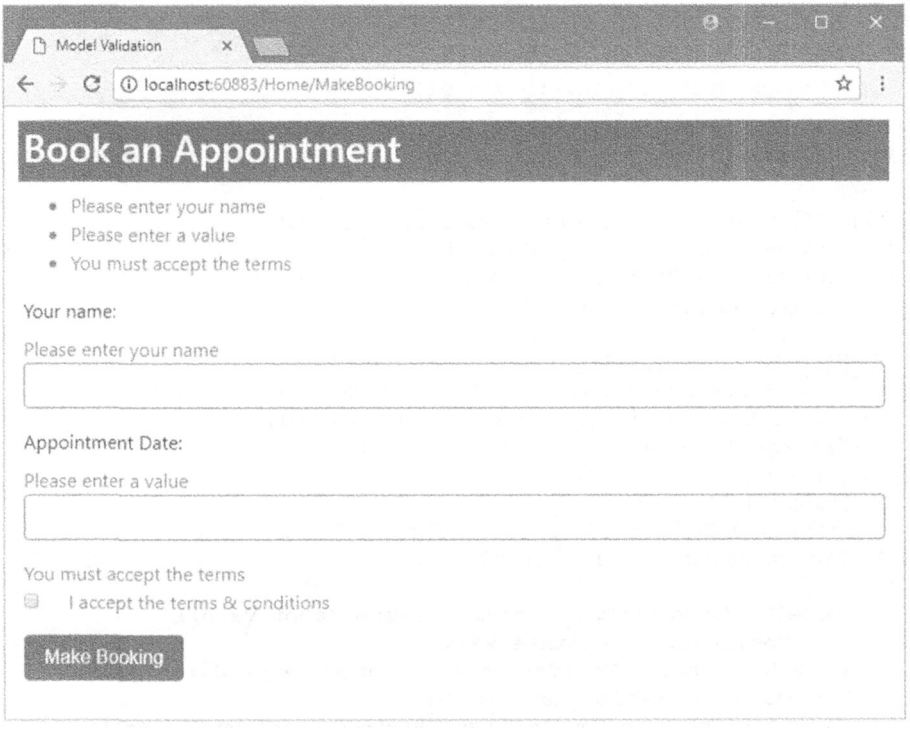

Рис. 27.5. Применение сообщений об ошибках проверки достоверности на уровне свойств

Отображение сообщений об ошибках проверки достоверности на уровне модели

Может показаться, что сводка по проверке достоверности в приложении избыточна, т.к. она просто дублирует сообщения уровня свойств, которые в целом более полезны пользователю, поскольку они находятся рядом с элементами формы, где должны быть устранены проблемы. Но сводка позволяет предпринять удобный трюк, который заключается в возможности отображать сообщения, применимые ко всей модели, а не только к отдельным свойствам. Другими словами, можно сообщать об ошибках, которые возникают в комбинации индивидуальных свойств, например, когда заданная дата допустима только в сочетании со специфическим именем.

В листинге 27.14 добавлена проверка достоверности, которая предотвращает назначение встречи пользователем по имени Джо (Joe) по понедельникам (Monday).

Листинг 27.14. Выполнение проверки на уровне модели в файле HomeController.cs из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() =>
      View("MakeBooking", new Appointment() { Date = DateTime.Now });

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) {
      if (string.IsNullOrEmpty(appt.ClientName)) {
        ModelState.AddModelError(nameof(appt.ClientName),
          "Please enter your name");
      }
      if (ModelState.GetValidationState("Date")
          == ModelValidationState.Valid && DateTime.Now > appt.Date) {
        ModelState.AddModelError(nameof(appt.Date),
          "Please enter a date in the future");
      }
      if (!appt.TermsAccepted) {
        ModelState.AddModelError(nameof(appt.TermsAccepted),
          "You must accept the terms");
      }
      if (ModelState.GetValidationState(nameof(appt.Date))
            == ModelValidationState.Valid
          && ModelState.GetValidationState(nameof(appt.ClientName))
            == ModelValidationState.Valid
          && appt.ClientName.Equals("Joe", StringComparison.OrdinalIgnoreCase)
          && appt.Date.DayOfWeek == DayOfWeek.Monday) {
        ModelState.AddModelError("",
          "Joe cannot book appointments on Mondays");
      }
      if (ModelState.IsValid) {
        return View("Completed", appt);
      } else {
        return View();
      }
    }
  }
}
```

Код выглядит более запутанным, чем есть на самом деле, что отражает саму природу проверки достоверности данных. Допустимость значений `ClientName` и `Date` проверяется путем инспектирования состояния модели перед выяснением, выпадает ли указанная дата на понедельник и содержит ли свойство `ClientName` строку `Joe`. Если Джо пытается назначить встречу на понедельник, тогда вызывается метод `AddModelError()` с передачей в первом аргументе пустой строки (`""`), которая указывает на то, что ошибка касается всей модели, а не отдельного свойства.

В листинге 27.15 значение атрибута `asp-validation-summary` установлено в `ModelOnly`, что приводит к исключению ошибок уровня свойств, поэтому сводка будет отображать только сообщения об ошибках, которые применимы к модели целиком.

Листинг 27.15. Отображение сообщений об ошибках проверки достоверности на уровне модели в файле `MakeBooking.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

@section scripts {
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script type="text/javascript">
    $(document).ready(function () {
      $("input.input-validation-error")
        .closest(".form-group").addClass("has-danger");
    });
  </script>
}

<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</h2>
</div>

<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div asp-validation-summary="ModelOnly" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <div><span asp-validation-for="ClientName" class="text-danger">
         </span>
    </div>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <div><span asp-validation-for="Date" class="text-danger"></span>
    </div>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <span asp-validation-for="TermsAccepted" class="text-danger"></span>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Запустим приложение, введем `Joe` в поле `ClientName` и выберем дату, выпадающую на понедельник, такую как 18 января 2027 года (`01/18/2027`). После отправки формы будет получен ответ, показанный на рис. 27.6.

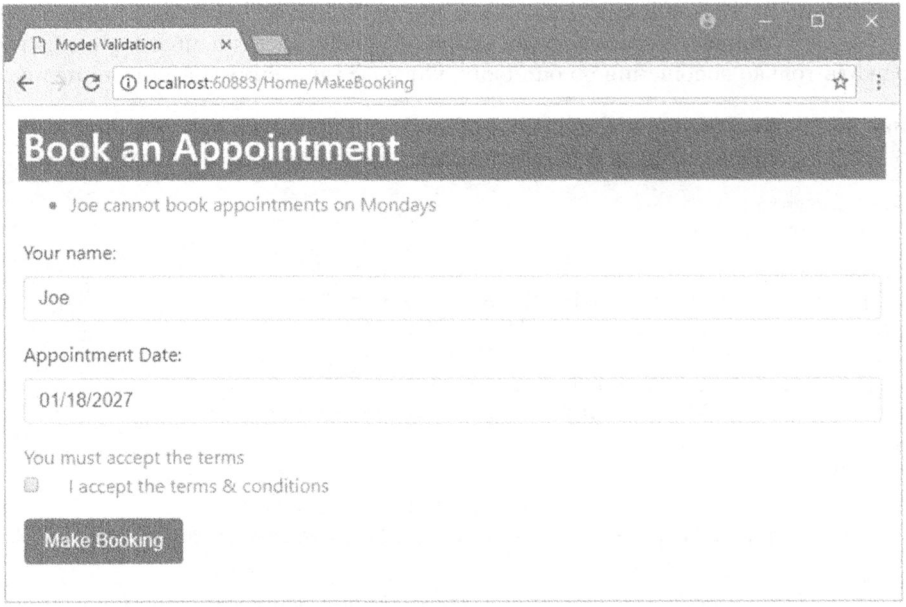

Рис. 27.6. Использование сообщений об ошибках проверки достоверности на уровнях модели и свойств

Указание правил проверки достоверности с помощью метаданных

Одна из проблем с помещением логики проверки достоверности внутрь метода действия связана с тем, что в итоге она дублируется в каждом методе действия, который получает данные от пользователя. Чтобы помочь сократить дублирование, процесс проверки достоверности поддерживает использование атрибутов. Атрибуты позволяют выражать правила проверки достоверности моделей прямо в классе модели, гарантируя применение одного и того же набора правил независимо от метода, который используется для обработки запроса.

В листинге 27.16 к классу `Appointment` применяются атрибуты для навязывания того же набора правил проверки достоверности на уровне свойств, который использовался в предыдущем разделе.

Листинг 27.16. Применение атрибутов проверки достоверности в файле `Appointment.cs` из папки `Models`

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models {

  public class Appointment {

    [Required]
    [Display(Name = "name")]
    public string ClientName { get; set; }
```

```
    [UIHint("Date")]
    [Required(ErrorMessage = "Please enter a date")]
    public DateTime Date { get; set; }

    [Range(typeof(bool), "true", "true",
      ErrorMessage = "You must accept the terms")]
    public bool TermsAccepted { get; set; }
  }
}
```

Здесь используются два атрибута проверки достоверности — `Required` и `Range`. Атрибут `Required` указывает, что ошибка проверки достоверности возникает, если пользователь не отправил значение для свойства. Атрибут `Range` задает подмножество приемлемых значений. В табл. 27.7 приведен набор встроенных атрибутов проверки достоверности, доступных в приложении MVC.

Таблица 27.7. Встроенные атрибуты проверки достоверности

Атрибут	Пример	Описание
`Compare`	`[Compare ("`*ДругоеСвойство*`")]`	Данный атрибут гарантирует, что свойства имеют одно и то же значение. Это полезно, когда вы предлагаете пользователю два раза предоставить ту же самую информацию, такую как адрес электронной почты или пароль
`Range`	`[Range(10, 20)]`	Этот атрибут гарантирует, что числовое значение (или значение свойства любого типа, реализующего интерфейс `IComparable`) не находится за рамками заданных минимального и максимального значений. Чтобы указать границу только с одной стороны, применяйте константу `MinValue` или `MaxValue` (например, `[Range(int.MinValue, 50)]`)
`RegularExpression`	`[RegularExpression ("`*шаблон*`")]`	Данный атрибут гарантирует, что строковое значение соответствует указанному шаблону регулярного выражения. Обратите внимание, что шаблон должен соответствовать *всему* предоставленному пользователем значению, а не только подстроке внутри него. По умолчанию при сопоставлении учитывается регистр символов, но с помощью модификатора `(?i)` его можно сделать нечувствительным к регистру, например, `[RegularExpression("(?i)`*шаблон*`")]`

Атрибут	Пример	Описание
Required	[Required]	Этот атрибут гарантирует, что значение не является пустой строкой или строкой, состоящей только из пробелов. Чтобы трактовать пробельные символы как допустимые, необходимо использовать [Required(AllowEmptyStrings=true)]
StringLength	[StringLength(10)]	Данный атрибут гарантирует, что строковое значение не превышает указанную максимальную длину. Можно также задавать минимальную длину: [StringLength(10, MinimumLength=2)]

Все атрибуты проверки достоверности позволяют указывать специальное сообщение об ошибке за счет установки значения для свойства ErrorMessage, например:

```
...
[UIHint("Date")]
[Required(ErrorMessage = "Please enter a date")]
public DateTime Date { get; set; }
...
```

Если специальное сообщение об ошибке не указано, тогда будут применяться стандартные сообщения, но они склонны раскрывать детали класса модели, которые не имеют смысла для пользователя, если только также не используется атрибут Display, как делалось в отношении свойства ClientName:

```
...
[Required]
[Display(Name = "name")]
public string ClientName { get; set; }
...
```

Стандартное сообщение, генерируемое атрибутом Required, отражает имя, указанное с помощью атрибута Display, поэтому оно не раскрывает пользователю имя самого свойства.

Обеспечение согласованной работы такой проверки достоверности требует определенного внимания. В качестве примера вот атрибут проверки достоверности, примененный к свойству TermsAccepted:

```
...
[Range(typeof(bool), "true", "true",
ErrorMessage="You must accept the terms")]
public bool TermsAccepted { get; set; }
...
```

В данном случае необходимо удостовериться в том, что пользователь отметил флажок для принятия условий. Использовать атрибут Required нельзя, т.к. браузер отправит для данного свойства значение false, если пользователь не отметил флажок. Проблема решается за счет возможности атрибута Range предоставлять объект Type

и указывать верхнюю и нижнюю границы в виде строковых значений. Установка обеих границ в `true` создает эквивалент атрибута `Required` для свойств типа `bool`, которые редактируются с применением флажков. Гарантирование того, что атрибуты проверки достоверности и данные, отправляемые браузером, успешно работают вместе, может потребовать некоторого экспериментирования.

Использование атрибутов проверки достоверности в классе модели означает, что метод действия в контроллере можно упростить, как показано в листинге 27.17.

Листинг 27.17. Удаление проверки достоверности на уровне свойств в файле `HomeController.cs` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {
  public class HomeController : Controller {
    public IActionResult Index() =>
      View("MakeBooking", new Appointment() { Date = DateTime.Now });
    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) {
      if (ModelState.GetValidationState(nameof(appt.Date))
            == ModelValidationState.Valid
          && ModelState.GetValidationState(nameof(appt.ClientName))
            == ModelValidationState.Valid
          && appt.ClientName.Equals("Joe", StringComparison.OrdinalIgnoreCase)
          && appt.Date.DayOfWeek == DayOfWeek.Monday) {
        ModelState.AddModelError("",
          "Joe cannot book appointments on Mondays");
      }
      if (ModelState.IsValid) {
        return View("Completed", appt);
      } else {
        return View();
      }
    }
  }
}
```

Атрибуты проверки достоверности применяются перед вызовом метода действия, т.е. при выполнении проверки достоверности на уровне модели для выяснения, допустимы ли индивидуальные свойства, можно по-прежнему полагаться на состояние модели. Чтобы взглянуть на атрибуты проверки достоверности в действии, запустим приложение и отправим форму, не вводя какие-либо данные (рис. 27.7).

Создание специального атрибута проверки достоверности для свойства

Процесс проверки достоверности может быть расширен за счет создания атрибута, который реализует интерфейс `IModelValidator`. Создадим папку `Infrastructure` и добавим в нее файл класса по имени `MustBeTrueAttribute.cs` с определением, приведенным в листинге 27.18.

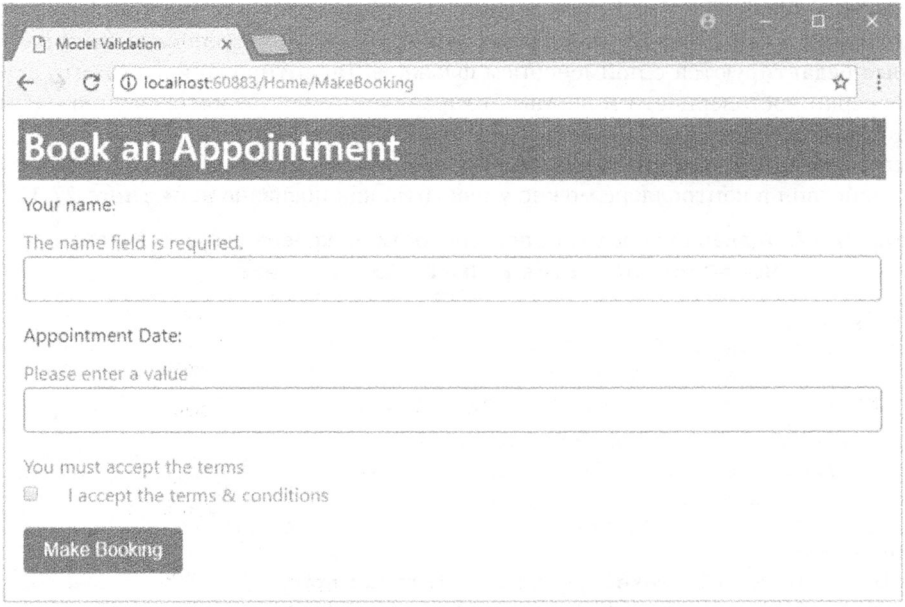

Рис. 27.7. Использование атрибутов проверки достоверности

Листинг 27.18. Содержимое файла `MustBeTrueAttribute.cs` из папки `Infrastructure`

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;

namespace ModelValidation.Infrastructure {
  public class MustBeTrueAttribute : Attribute, IModelValidator {

    public bool IsRequired => true;

    public string ErrorMessage { get; set; } = "This value must be true";

    public IEnumerable<ModelValidationResult> Validate(
            ModelValidationContext context) {
      bool? value = context.Model as bool?;
      if (!value.HasValue || value.Value == false) {
        return new List<ModelValidationResult> {
          new ModelValidationResult("", ErrorMessage)
        };
      } else {
        return Enumerable.Empty<ModelValidationResult>();
      }
    }
  }
}
```

В интерфейсе IModelValidator определено свойство IsRequired, которое применяется для указания, требуется ли проверка достоверности с помощью этого класса (что немного вводит в заблуждение, т.к. значение, возвращаемое данным свойством, просто используется для упорядочения атрибутов проверки достоверности, чтобы обязательные атрибуты выполнялись первыми). Метод Validate() применяется для выполнения проверки достоверности и получает информацию через экземпляр класса ModelValidationContext, полезные свойства которого описаны в табл. 27.8.

Таблица 27.8. Полезные свойства класса ModelValidationContext

Имя	Описание
Model	Возвращает значение свойства, подлежащего проверке достоверности, которым в рассматриваемом примере будет значение TermsAccepted
Container	Возвращает содержащий свойство объект, которым в рассматриваемом примере будет объект Appointment
ActionContext	Возвращает объект ActionContext, предоставляющий данные контекста и описывающий метод действия, который будет обрабатывать запрос
ModelMetadata	Возвращает объект ModelMetadata, который детально описывает класс модели, подвергающийся проверке достоверности

Метод Validate() возвращает последовательность объектов ModelValidation Result, каждый из которых описывает одиночную ошибку проверки достоверности. В примере атрибута объект ModelValidationResult создается, если значение context.Model не равно true.

В первом аргументе конструктору ModelValidationResult передается имя свойства, с которым ассоциирована ошибка, что указывается как пустая строка при проверке достоверности индивидуальных свойств. Во втором аргументе задается сообщении об ошибке, которое будет отображаться пользователю. В листинге 27.19 атрибут Range заменяется специальным атрибутом.

Листинг 27.19. Применение специального атрибута в файле Appointment.cs из папки Models

```
using System;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Infrastructure;

namespace ModelValidation.Models {

  public class Appointment {

    [Required]
    [Display(Name = "name")]
    public string ClientName { get; set; }

    [UIHint("Date")]
    [Required(ErrorMessage = "Please enter a date")]
    public DateTime Date { get; set; }

    [MustBeTrue(ErrorMessage = "You must accept the terms")]
    public bool TermsAccepted { get; set; }
  }
}
```

Результат использования специального атрибута проверки достоверности будет точно таким же, как и атрибута `Range`, но при чтении кода цель специального атрибута более очевидна.

Выполнение проверки достоверности на стороне клиента

Все продемонстрированные до сих пор приемы проверки были примерами *проверки достоверности на стороне сервера*. Это означает, что пользователь посылает свои данные серверу, сервер проверяет данные и отправляет обратно результаты проверки (либо признак успешности, либо список ошибок, подлежащих исправлению).

В веб-приложениях пользователи обычно ожидают немедленного отклика проверки достоверности без необходимости отправки чего-либо серверу. Такая возможность известна как *проверка достоверности на стороне клиента* и реализуется с применением JavaScript. Вводимые пользователем данные проверяются на предмет достоверности перед их отправкой серверу, снабжая пользователя немедленным откликом и шансом скорректировать любые проблемы.

Инфраструктура MVC поддерживает *ненавязчивую проверку достоверности на стороне клиента*. Термин "ненавязчивая" означает, что правила проверки достоверности выражаются с использованием атрибутов, которые добавляются к HTML-элементам, генерируемым представлениями. Атрибуты интерпретируются библиотекой JavaScript, входящей в состав инфраструктуры MVC, которая в свою очередь конфигурирует библиотеку jQuery Validation, выполняющую действительную работу по проверке достоверности. В последующих разделах будет показано, как работает встроенная поддержка проверки достоверности, и продемонстрированы способы расширения ее функциональности для обеспечения проверки достоверности на стороне клиента.

Совет. Проверка достоверности на стороне клиента сосредоточена на проверке достоверности индивидуальных свойств. В действительности настроить проверку достоверности клиентской стороны на уровне модели с помощью встроенной в MVC поддержки нелегко. С этой целью в большинстве приложений MVC проверка на стороне клиента применяется для свойств, а проверка на стороне сервера — для всей модели.

Прежде всего, понадобится добавить в приложение новые пакеты JavaScript, используя Bower (листинг 27.20).

Листинг 27.20. Добавление пакетов в файле `bower.json` из папки `ModelValidation`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1",
    "jquery-validation": "1.17.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

Применение проверки достоверности на стороне клиента означает добавление к представлению трех файлов JavaScript: библиотеки jQuery, библиотеки проверки достоверности jQuery и библиотеки ненавязчивой проверки достоверности от Microsoft; все они задействованы в листинге 27.21.

Совет. Инструмент Bower не всегда корректно выполняет установку пакетов проверки достоверности. Если вы обнаружите, что папка `wwwroot/lib` не содержит необходимых файлов, тогда удалите ее вместе с содержимым. Откройте окно PowerShell, перейдите в папку проекта и запустите по очереди команды `bower cache clean` и `bower install`, чтобы загрузить актуальные копии пакетов проверки достоверности.

Листинг 27.21. Добавление элементов проверки достоверности JavaScript в файле `MakeBooking.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

@section scripts {
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script src="/lib/jquery-validation/dist/jquery.validate.min.js">
  </script>
  <script
    src=
"/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js">
  </script>
}

<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</
h2></div>

<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div asp-validation-summary="ModelOnly" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <div><span asp-validation-for="ClientName" class="text-danger"></span>
    </div>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <div><span asp-validation-for="Date" class="text-danger"></span></div>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <span asp-validation-for="TermsAccepted" class="text-danger"></span>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Файлы должны добавляться в показанном порядке. Когда вспомогательные функции дескрипторов трансформируют элементы `input`, они инспектируют атрибуты проверки достоверности, примененные к свойствам класса модели, и добавляют атрибуты к выходным элементам. Запустив приложение и просмотрев HTML-разметку, отправленную браузеру, можно заметить элемент следующего вида:

```
<input class="form-control" type="text" data-val="true"
       data-val-required="The name field is required." id="ClientName"
       name="ClientName" value="" />
```

Код JavaScript ищет элементы с атрибутом `data-val` и выполняет локальную проверку достоверности в браузере, когда пользователь отправляет форму, не посылая HTTP-запрос серверу. Чтобы увидеть эффект, запустим приложение и отправим форму, одновременно используя инструменты <F12> для наблюдения за тем, что сообщения об ошибках проверки достоверности отображаются, хотя никакие HTTP-запросы серверу не отправлялись.

Избегание конфликтов с проверкой достоверности, встроенной в браузеры

Ряд браузеров, поддерживающих HTML5, обеспечивают простую проверку достоверности на стороне клиента, которая основана на атрибутах, применяемых к элементам `input`. Общая идея в том, что элемент `input`, к которому применен, скажем, атрибут `required`, вызовет отображение браузером сообщения об ошибке проверки достоверности, если пользователь попытается отправить форму, не предоставив значение для этого элемента.

В случае генерации элементов форм из моделей, как делалось ранее в главе, никаких проблем со встроенной в браузеры проверкой достоверности возникать не будет, поскольку для указания правил проверки инфраструктура MVC генерирует и использует атрибуты данных. (Таким образом, например, элемент `input`, который обязан иметь значение, помечается атрибутом `data-val-required`, не распознаваемым браузерами.)

Тем не менее, вы можете столкнуться с проблемами, если не в состоянии полностью контролировать разметку в приложении, что часто происходит при поступлении содержимого, сгенерированного в другом месте. В результате с формой может работать и проверка достоверности jQuery, и проверка достоверности, встроенная в браузер, что всего лишь сбивает с толку пользователя. Чтобы избежать такой проблемы, к элементу `form` можно добавить атрибут `novalidate`.

Одна из замечательных характеристик проверки достоверности на стороне клиента MVC связана с тем, что те же самые атрибуты, используемые для указания правил проверки достоверности, применяются внутри клиента *и* на сервере. Это означает, что данные, поступающие из браузеров, которые не поддерживают JavaScript, подвергаются такой же проверке достоверности, что и данные из браузеров, поддерживающих JavaScript, безо всяких дополнительных усилий. Однако это также означает, что специальные атрибуты проверки достоверности при проверке на стороне клиента не поддерживаются, потому что код JavaScript не имеет возможности реализовать специальную логику внутри клиента. Иными словами, при желании использовать проверку достоверности на стороне клиента необходимо придерживаться встроенных атрибутов, описанных в табл. 27.7.

Сравнение проверки достоверности на стороне клиента в MVC и проверки достоверности с помощью библиотеки jQuery Validation

Функциональность проверки достоверности на стороне клиента в MVC построена на основе библиотеки jQuery Validation. Если хотите, то можете работать с библиотекой jQuery Validation напрямую, игнорируя средства MVC. Библиотека jQuery Validation обладает высокой гибкостью и большими возможностями. Ее имеет смысл исследовать хотя бы для того, чтобы понять, каким образом настраивать средства MVC с целью извлечения максимальной пользы из доступных вариантов проверки. Библиотека jQuery Validation подробно описана в книге *jQuery 2.0 для профессионалов* (ИД "Вильямс").

Выполнение удаленной проверки достоверности

В завершение главы мы рассмотрим *удаленную (дистанционную) проверку достоверности*. Это прием проверки достоверности на стороне клиента, при котором для выполнения проверки вызывается метод действия на сервере.

Распространенный пример удаленной проверки достоверности предусматривает выяснение доступности имени пользователя в приложениях, где оно должно быть уникальным, когда пользователь посылает данные и производится проверка достоверности на стороне клиента. В качестве части такого процесса серверу отправляется запрос Ajax для проверки имени пользователя. Если имя пользователя уже было выдано, тогда отображается сообщение об ошибке проверки и пользователю предоставляется возможность ввести другое имя.

Процесс может выглядеть похожим на обычную проверку достоверности на стороне сервера, но такой подход обладает рядом преимуществ. Во-первых, удаленно проверяться будут только некоторые свойства; ко всем остальным значениям данных, которые ввел пользователь, по-прежнему будет применяться проверка достоверности на стороне клиента. Во-вторых, запрос является относительно легковесным и сосредоточенным на проверке достоверности, а не на обработке целого объекта модели.

Третье отличие связано с тем, что удаленная проверка достоверности выполняется в фоновом режиме. Пользователю не приходится щелкать на кнопке отправки и ожидать визуализации нового представления. В итоге пользователи получают более отзывчивый интерфейс, что особенно важно в случае медленного сетевого соединения между браузером и сервером.

Тем не менее, удаленная проверка достоверности сопряжена с компромиссом. Она соблюдает баланс между проверками на стороне клиента и на стороне сервера, но требует отправки запросов серверу приложений, поэтому не будет настолько быстрой, как обычная проверка достоверности на стороне клиента.

Первый шаг в сторону использования удаленной проверки достоверности предполагает создание метода действия, который может проверить одно из свойств модели. Мы будем проверять свойство `Date` модели `Appointment`, чтобы обеспечить нахождение запрошенной даты встречи в будущем. (Это одно из исходных правил проверки достоверности, применяемых в начале главы, но его невозможно реализовать с помощью стандартных средств проверки на стороне клиента.)

В листинге 27.22 приведен код метода действия `ValidateDate()`, добавленного в контроллер `Home`.

Листинг 27.22. Добавление метода действия для проверки достоверности в файле HomeController.cs из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() =>
      View("MakeBooking", new Appointment() { Date = DateTime.Now });

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) {
      if (ModelState.GetValidationState(nameof(appt.Date))
            == ModelValidationState.Valid
          && ModelState.GetValidationState(nameof(appt.ClientName))
            == ModelValidationState.Valid
          && appt.ClientName.Equals("Joe", StringComparison.OrdinalIgnoreCase)
          && appt.Date.DayOfWeek == DayOfWeek.Monday) {
        ModelState.AddModelError("",
          "Joe cannot book appointments on Mondays");
      }

      if (ModelState.IsValid) {
        return View("Completed", appt);
      } else {
        return View();
      }
    }

    public JsonResult ValidateDate(string Date) {
      DateTime parsedDate;

      if (!DateTime.TryParse(Date, out parsedDate)) {
        return Json("Please enter a valid date (mm/dd/yyyy)");
      } else if (DateTime.Now > parsedDate) {
        return Json("Please enter a date in the future");
      } else {
        return Json(true);
      }
    }
  }
}
```

Методы действий, поддерживающие удаленную проверку достоверности, должны возвращать объект типа `JsonResult`, который сообщает инфраструктуре MVC о том, что работа производится с данными JSON, как объяснялось в главе 20. В дополнение к возвращению такого результата методы действий для проверки достоверности обязаны определять параметр, который имеет то же самое имя, что и проверяемое поле данных; в рассматриваемом примере это `Date`. Внутри метода действий проверка достоверности выполняется путем преобразования значения в объект `DateTime` и выяснения, относится ли дата к будущему.

Совет. Можно было бы воспользоваться привязкой моделей, так что параметром метода действия стал бы объект `DateTime`, но тогда метод проверки достоверности не вызывался бы в ситуации, когда пользователь ввел бессмысленное значение вроде `apple`. Причина в том, что связывателю модели не удается создать объект `DateTime` из `apple` и генерируется исключение. Средство удаленной проверки достоверности не способно обработать такое исключение, поэтому исключение молча отбрасывается. В итоге возникает нежелательный эффект: поле данных *не* подсвечивается, создавая впечатление, что введенное пользователем значение является допустимым. Как правило, наилучший подход к удаленной проверке достоверности предусматривает получение методом действия параметра `string` и явное выполнение любого преобразования типа, разбора или привязки модели.

Результаты проверки достоверности выражаются с применением метода `Json()`, создающего результат в формате JSON, который может разобрать и обработать сценарий удаленной проверки достоверности на стороне клиента. Если значение допустимо, тогда в качестве параметра методу `Json()` передается `true`:

```
...
return Json(true);
...
```

При наличии проблемы в параметре передается сообщение об ошибке проверки, которое должен увидеть пользователь:

```
...
return Json("Please enter a date in the future");
...
```

Чтобы использовать метод удаленной проверки достоверности, к нужному свойству класса модели применяется атрибут `Remote` (листинг 27.23).

Листинг 27.23. Использование атрибута `Remote` в файле `Appointment.cs` из папки `Models`

```
using System;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Infrastructure;
using Microsoft.AspNetCore.Mvc;

namespace ModelValidation.Models {

  public class Appointment {

    [Required]
    [Display(Name = "name")]
    public string ClientName { get; set; }

    [UIHint("Date")]
    [Required(ErrorMessage = "Please enter a date")]
    [Remote("ValidateDate", "Home")]
    public DateTime Date { get; set; }

    [MustBeTrue(ErrorMessage = "You must accept the terms")]
    public bool TermsAccepted { get; set; }
  }
}
```

В качестве аргументов атрибуту `Remote` указываются имя действия и контроллер, предназначенные для генерации URL, по которому библиотека проверки достоверности JavaScript будет обращаться с целью выполнения проверки — в рассматриваемом случае это действие `ValidateDate` из контроллера `Home`.

Чтобы посмотреть на работу удаленной проверки достоверности, запустим приложение, перейдем на URL вида `/Home` и введем дату, относящуюся к прошлому. После того как фокус ввода переместится на другой элемент, появится сообщение об ошибке проверки достоверности (рис. 27.8).

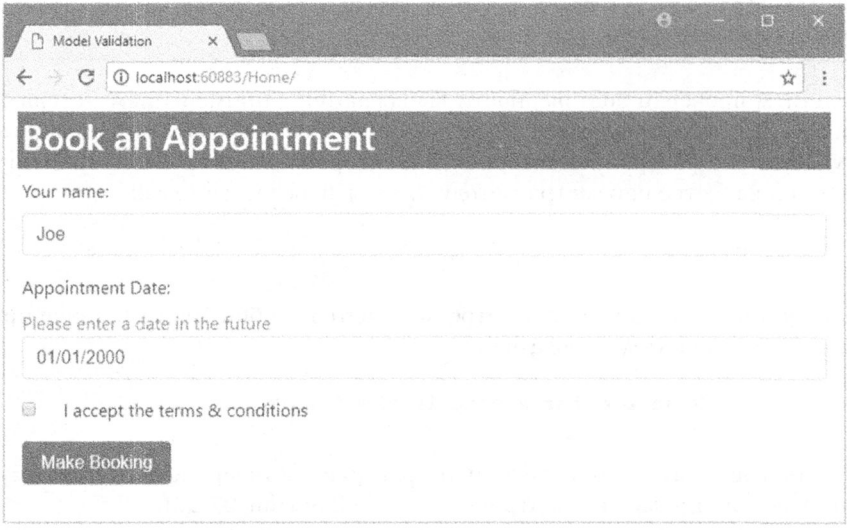

Рис. 27.8. Выполнение удаленной проверки достоверности

Внимание! Метод действия для проверки достоверности будет вызываться, когда пользователь впервые отправляет форму, и затем каждый раз, когда данные редактируются. Для элементов ввода текста любое нажатие клавиши будет иметь следствием обращение к серверу. В некоторых приложениях это может вылиться в значительное количество запросов, что должно быть учтено при описании требований к производительности сервера и ширине полосы пропускания в производственной среде. Кроме того, может быть принято решение *отказаться* от удаленной проверки достоверности для свойств, проверка которых сопряжена с высокими затратами (например, если выяснение уникальности имени пользователя требует обращения к медленному серверу).

Резюме

В настоящей главе был представлен широкий спектр приемов, доступных для выполнения проверки достоверности моделей, которая гарантирует, что предоставляемые пользователем данные удовлетворяют ограничениям, налагаемым на модель данных. Проверка достоверности моделей является важной темой, и наличие в приложении подходящих средств проверки жизненно важно для обеспечения пользователям комфортных условий работы.