ГЛАВА 3

Работа с данными

Редко когда удается встретить приложение, которое бы каким-либо образом не имело дело с данными, поэтому не должен вызывать удивление тот факт, что ASP.NET MVC предоставляет великолепную поддержку работы с данными на всех уровнях инфраструктуры. В этой главе мы рассмотрим инструменты, обеспечивающие такую поддержку, и покажем, как использовать их в сценариях, управляемых данными, за счет добавления этой функциональности к образцовому приложению EBuy.

Поскольку EBuy является сайтом онлайновых аукционов, наиболее важным его сценарием является предоставление пользователям возможности создавать списки аукционных товаров, содержащие детальные сведения по каждому элементу, который они желают продать. Итак, давайте посмотрим, как ASP.NET MVC может помочь в поддержке этого важного сценария.

Построение формы

Концепция HTML-формы столь же стара, как сама веб-сеть. Хотя в настоящее время браузеры стали более функциональными, а HTML-форму можно стилизовать и снабдить поведением с помощью JavaScript таким способом, который еще пять лет назад казался невозможным, в ее основе по-прежнему лежит набор старых добрых полей, готовых к заполнению и обратной отправке серверу.

Несмотря на то что ASP.NET MVC приветствует написание большей части HTML-разметки "вручную", эта инфраструктура предлагает набор вспомогательных методов HTML, позволяющих генерировать разметку для HTML-форм, среди которых `Html.TextBox`, `Html.Password` и `Html.HiddenField`. Кроме того, в ASP.NET MVC имеется несколько "более интеллектуальных" вспомогательных методов, таких как `Html.LabelFor` и `Html.EditorFor`, которые динамически определяют подходящую HTML-разметку на основе имени и типа переданного свойства модели.

Именно эти вспомогательные методы будут использоваться в примере веб-сайта EBuy для построения HTML-формы, которая позволит пользователям выполнять отправку в адрес действия `AuctionsController.Create` для создания новых аукционных товаров. Чтобы увидеть данные вспомогательные методы в действии, добавьте новое представление по имени `Create.cshtml` и заполните его следующей разметкой:

```
<h2>Create Auction</h2>
@using (Html.BeginForm()) {
   <p>
      @Html.LabelFor(model => model.Title)
      @Html.EditorFor(model => model.Title)
   </p>
```

```
    <p>
      @Html.LabelFor(model => model.Description)
      @Html.EditorFor(model => model.Description)
    </p>
    <p>
      @Html.LabelFor(model => model.StartPrice)
      @Html.EditorFor(model => model.StartPrice)
    </p>
    <p>
      @Html.LabelFor(model => model.EndTime)
      @Html.EditorFor(model => model.EndTime)
    </p>
    <p>
      <input type="submit" value="Create" />
    </p>
}
```

Затем добавьте в контроллер приведенное ниже действие для визуализации этого представления:

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

Показанное выше представление визуализируется в следующую HTML-разметку, предназначенную для браузера:

```
<h2>Create Auction</h2>
<form action="/auction/create" method="post">
  <p>
    <label for="Title">Title</label>
    <input id="Title" name="Title" type="text" value="">
  </p>
  <p>
    <label for="Description">Description</label>
    <input id="Description" name="Description" type="text" value="">
  </p>
  <p>
    <label for="StartPrice">StartPrice</label>
    <input id="StartPrice" name="StartPrice" type="text" value="">
  </p>
  <p>
    <label for="EndTime">EndTime</label>
    <input id="EndTime" name="EndTime" type="text" value="">
  </p>
  <p>
    <input type="submit" value="Create">
  </p>
</form>
```

Пользователь затем заполняет эту форму значениями и отправляет ее действию `/auctions/create`. Хотя с точки зрения браузера это выглядит как отправка формы самой себе (URL для визуализации формы изначально установлен также в `/auctions/create`), именно здесь в игру вступает второе действие контроллера `Create` с атрибутом `HttpPostAttribute`, сообщающее ASP.NET MVC о том, что это перегруженная версия, которая обрабатывает действие `POST` отправки формы.

А теперь самое время действительно *сделать* что-нибудь со значениями отправленной формы — но что?

Обработка отправок формы

Перед тем как можно будет работать со значениями, отправленными контроллеру, понадобится извлечь их из запроса. Как было показано в разделе "Параметры действий" главы 1, простейший способ предусматривает использование модели в качестве параметра действия и, к счастью, модель уже создана: вспомните класс `Auction` из раздела "Модели" в главе 1.

Для привязки к созданному ранее классу `Auction` просто укажите параметр типа `Auction` в числе параметров действия контроллера `Create`:

```
[HttpPost]
public ActionResult Create(Auction auction)
{
   // Создать объект Auction в базе данных.
   return View(auction);
}
```

Наиболее важным аспектом модели `Auction` в этой точке является тот факт, что имена свойств (`Title`, `Description` и т.д.) соответствуют именам полей формы, которые были отправлены действию `Create`. Имена свойств критически важны, поскольку привязка модели ASP.NET MVC пытается заполнить их значения из полей формы с совпадающими именами.

Если вы запустите это приложение, выполните отправку почты и обновите страницу, то увидите, что модель `Auction` заполнилась введенными ранее значениями. В этот момент действие просто возвращает заполненный параметр `auction` обратно представлению, которое может применяться для отображения значений формы пользователю с целью подтверждения отправки.

Это приближает нас на один шаг к получению приложения, которое делает что-то полезное, однако по-прежнему остается много чего нужно реализовать, начиная с действительного *сохранения* данных.

Сохранение данных в базе

Хотя инфраструктура ASP.NET MVC Framework не имеет никаких встроенных средств доступа к данным, существует множество популярных библиотек .NET доступа к данным, которые помогут упростить работу с базой данных.

Одной из таких библиотек является *Entity Framework* (EF) от Microsoft. Библиотека Entity Framework — это простая и гибкая инфраструктура *объектно-реляционного отображения* (object relational mapping — ORM), которая позволяет разработчикам запрашивать и обновлять данные в базе объектно-ориентированным путем. Более того, Entity Framework в действительности представляет собой часть платформы .NET Framework, с полной поддержкой и обилием доступной документации, обеспечиваемой Microsoft.

Инфраструктура Entity Framework предлагает несколько разных подходов к определению модели данных и применению этой модели для доступа в базу данных, но, пожалуй, самым интригующим подходом является *Code First* ("сначала код"). Образ мышления, заложенный в разработку Code First, заключается в том, что модель приложения является центральной частью и движущей силой всего происходящего во время разработки.

Entity Framework Code First: соглашение по конфигурации

При разработке Code First взаимодействие осуществляется через простые классы моделей (также называемые традиционными объектами CLR (Plain Old CLR Object — POCO)). Подход Code First в Entity Framework заходит настолько далеко, что даже генерирует на основе модели схему базы данных и использует эту схему для создания базы данных и ее сущностей (таблиц, отношений и т.д.) при запуске приложения.

Подход Code First делает это за счет следования определенным соглашениям, которые автоматически оценивают различные свойства и классы, образующие уровень моделей, с целью выяснения, каким образом информация в этих моделях должна быть сохранена, и даже как отношения между разными классами моделей могут быть эффективно представлены в терминах отношений базы данных.

Например, в образцовом приложении EBuy класс `Auction` отображается на таблицу базы данных `Auctions` и все его свойства представляют столбцы в этой таблице. Имена таблицы и столбцов автоматически выводятся из имен класса и его членов.

Показанная ранее модель `Auction` очень проста, но с ростом потребностей приложения сложность модели также будет возрастать: мы добавим дополнительные свойства, бизнес-логику и даже отношения с другими моделями. Однако это не проблема для Entity Framework Code First, поскольку это средство обычно способно обрабатывать более сложные модели с той же легкостью, что и простые. В главе 8 в простую модель `Auction`, показанную в этой главе, привносится более реалистичная сложность и демонстрируется, что подход Entity Framework Code First обладает возможностью обработки этих более сложных отображений (а также объясняется, что делать, если он не справляется с ними).

Создание уровня доступа к данным с помощью Entity Framework Code First

В основе подхода Entity Framework Code First лежит класс `System.Data.Entity.DbContext`. Этот класс (или созданные производные от него классы) выступает в качестве шлюза к базе данных, предоставляя все необходимые действия, связанные с данными. Чтобы приступить к использованию класса `DbContext`, понадобится создать собственный класс, производный от него, который на самом деле довольно прост:

```
using System.Data.Entity;
public class EbuyDataContext : DbContext
{
   public DbSet<Auction> Auctions { get; set; }
}
```

В этом примере (`EbuyDataContext.cs`) мы создали специальный класс контекста данных по имени `EbuyDataContext`, производный от `DbContext`. Этот отдельный класс определяет свойство `System.Data.Entity.DbSet<T>`, где `T` — это сущность, которая будет редактироваться и сохраняться в базе данных. В предшествующем примере мы определили `System.Data.Entity.DbSet<Auction>` для указания на то, что приложение нуждается в сохранении и редактировании экземпляров класса `Auction` в базе данных. Однако в контексте данных можно определять более одной сущности, и по мере продвижения разработки мы будем добавлять в класс `EbuyDataContext` дополнительные сущности (или свойства `DbSet`).

Если создание специального контекста данных осуществляется легко, то его использование еще легче, как продемонстрировано в следующем примере. В следующем фрагменте действие контроллера `Create` модифицируется для сохранения отправлен-

ного объекта Auction в базе данных, для чего объект Auction просто добавляется в коллекцию EbuyDataContext.Auctions с последующим сохранением изменений:

```
[HttpPost]
public ActionResult Create(Auction auction)
{
   var db = new EbuyDataContext();
   db.Auctions.Add(auction);
   db.SaveChanges();
   return View(auction);
}
```

На этот раз после запуска приложения и отправки заполненной формы в таблице Auctions базы данных появится новая строка, содержащая информацию, отравленную в форме.

Если вы продолжите пользоваться этим примером и поэкспериментируете с разными значениями в полях формы, то заметите, что привязка моделей ASP.NET MVC является весьма терпимой к ошибкам, позволяя вводить все что угодно и молча отказывая, когда она не может преобразовать отправленные значения формы в строгие типы (например, в ситуации, когда пользователь вводит строку ABC в поле типа int). Если необходим более строгий контроль над тем, какие данные сохраняются в базе, потребуется применять проверку достоверности данных к модели.

Проверка достоверности данных

Что касается данных, то обычно существует ряд применяемых правил и ограничений, таких как поля, которые не должны быть пустыми или значения которых должны находиться в заданном диапазоне, чтобы рассматриваться как "допустимые". Естественно, ASP.NET MVC распознает такие важные концепции, интегрируя их прямо в процесс обработки каждого запроса.

В качестве части процесса выполнения действия контроллера инфраструктура ASP.NET MVC Framework проверяет достоверность данных, которые передаются этому действию контроллера, заполняя объект ModelState любыми обнаруженными ошибками и передавая этот объект контроллеру. Затем действия контроллера могут запросить ModelState для выяснения допустимости запроса и отреагировать соответствующим образом, например, сохранить допустимый объект в базе данных или вернуть пользователя в исходную форму для исправления ошибок проверки достоверности, зафиксированных в недопустимом запросе.

Ниже приведен пример действия AuctionsController.Create, обновленного для проверки словаря ModelState с применением только что описанной логики "сохранить или исправить":

```
[HttpPost]
public ActionResult Create(Auction auction)
{
   if (ModelState.IsValid)
   {
      var db = new EbuyDataContext();
      db.Auctions.Add(auction);
      db.SaveChanges();
      return RedirectToAction("Index");
   }
   return View(auction);
}
```

Инфраструктура ASP.NET MVC Framework — это не единственное средство, которое может добавлять ошибки проверки достоверности в `ModelState`. Разработчики могут запускать собственную логику для обнаружения проблем, которые инфраструктура не может перехватить, и вручную добавлять информацию об ошибках, используя твкой метод:

```
ModelState.AddModelError(string key, string message)
```

Пусть, например, существует требование, что аукционы должны длиться, по крайней мере, один день. Другими словами, значение свойства `EndTime` объекта `Auction` должно превышать текущее время плюс один день.

Действие `AuctionsController.Create` может явно проверять это перед тем, как пытаться сохранить объект `Auction`, и предусматривать специальное сообщение об ошибке, когда такая ситуация возникает:

```
[HttpPost]
public ActionResult Create(Auction auction)
{
   if (auction.EndTime <= DateTime.Now.AddDays(1))
   {
      ModelState.AddModelError(
         "EndTime",
         "Auction must be at least one day long"
      );
   }

   if (ModelState.IsValid)
   {
      var db = new EbuyDataContext();
      db.Auctions.Add(auction);
      db.SaveChanges();
      return RedirectToAction("Index");
   }

   return View(auction);
}
```

Хотя этот подход работает довольно неплохо, он приводит к нарушению принципа разделения ответственности в приложении. В частности, контроллеры не должны содержать бизнес-логику подобного рода: бизнес-логика относится к модели. Итак, давайте перенесем бизнес-логику в модель.

Указание бизнес-правил с помощью аннотаций данных

Практика обеспечения качества данных — проверка достоверности данных — является настолько распространенной задачей при разработке приложений, что для разработчиков вполне естественно обращаться к одной из многих доступных инфраструктур для получения помощи в определении и выполнении логики проверки достоверности данных наиболее эффективным образом.

На самом деле это настолько общая потребность, что в рамках ядра .NET Framework поставляется очень эффективный и простой в использовании API-интерфейс проверки достоверности данных под названием Data Annotations (аннотации данных). Как должно быть понятно из названия, API-интерфейс Data Annotations предоставляет набор атрибутов .NET, которые разработчики могут применять к свойствам классов объектов данных. Эти атрибуты предлагают декларативный способ применения правил проверки достоверности непосредственно к модели.

Более того, привязка модели ASP.NET MVC обеспечивает поддержку аннотаций данных без дополнительного конфигурирования. Для демонстрации работы аннотаций данных ASP.NET MVC давайте рассмотрим процесс применения проверки достоверности к классу `Auction`. Перед началом применения логики проверки достоверности необходимо определить, какие значения ожидаются для свойств класса `Auction`. Какие поля должны быть обязательными? Имеют ли какие-то поля определенные диапазоны допустимых значений?

Обязательные поля

Поскольку свойства `Title` и `Description` класса `Auction` критически важны для описания продаваемого на аукционе товара, мы применим к этим двум полям аннотацию данных `RequiredAttribute`, пометив их как поля, которые обязательно должны иметь данные, чтобы считаться допустимыми:

```
[Required]
public string Title { get; set; }

[Required]
public string Description { get; set; }
```

В дополнение к пометке поля как обязательного с помощью `RequiredAttribute`, можно также обеспечить, чтобы строковые значения имели максимальную длину, применив для этого атрибут `StringLengthAttribute`. Например, было решено, что заголовки аукционных товаров должны сохраняться короткими, не превышая максимальную длину в 50 символов:

```
[Required, StringLength(50)]
public string Title { get; set; }
```

Если теперь пользователь отправит форму с полем `Title`, в котором введена строка с более чем 50 символами, средство проверки достоверности модели ASP.NET MVC сообщит об ошибке.

Допустимые диапазоны

Далее рассмотрим стартовую цену аукционного товара: она представляется свойством `StartPrice`, имеющим тип `decimal`. Поскольку `decimal` — это тип значения, свойство `StartPrice` будет всегда иметь, по крайней мере, значение по умолчанию, равное `0`, так что пометка этого свойства как обязательного будет избыточной. Тем не менее, со стартовыми ценами аукционных товаров связана другая логика: эти значения не могут быть отрицательными! Чтобы решить эту проблему, примените атрибут `RangeAttribute` к полю `StartPrice` и укажите в качестве минимального значения `1`. Так как `RangeAttribute` требует еще и максимального значения, укажите также верхний предел.

```
[Range(1, 10000]
public decimal StartPrice { get; set; }
```

В этом примере используется диапазон типа `double`, но аннотация `RangeAttribute` имеет также перегруженную версию (`Range(Type type, string min, string max)`) для поддержки диапазона любого типа, который реализует интерфейс `IComparable` и может быть создан путем разбора или преобразования строковых значений. Хорошим примером может служить проверка диапазона дат; например, следующая аннотация гарантирует, что дата находится после определенного момента времени:

```
[Range(typeof(DateTime), "1/1/2012", "12/31/9999"]
public DateTime EndTime { get; set; }
```

Этот пример обеспечивает, что значение свойства `EndTime` будет, по крайней мере, позже 1 января 2012 г.

Параметрами атрибутов .NET должны быть значения, которые известны на этапе компиляции и не могут вычисляться во время выполнения, что исключает использование таких значений, как `DateTime.Now`, для выяснения, относится ли дата к будущему. Вместо этого мы должны выбрать произвольную дату, например, `1/1/2012`, которая хотя и не обеспечит того, что введенная дата относится к моменту, находящемуся после отправки формы, но, по крайней мере, позволит избежать ввода дат из далекого прошлого.

Этот недостаток точности является компромиссом, на который пришлось пойти, чтобы иметь возможность пользоваться `RangeAttribute`. Если сложившаяся ситуация требует большей точности, придется прибегнуть к атрибуту `CustomValidationAttribute`, который позволяет выполнять произвольную логику для проверки достоверности свойств. Хотя возможность выполнения произвольного кода посредством `CustomValidatorAttribute`, несомненно, удобна, она представляет собой менее декларативный подход, который ограничивает информацию, доступную другим компонентам, таким как инфраструктура проверки достоверности на стороне клиента ASP.NET MVC.

Специальные сообщения об ошибках

В заключение важно отметить, что все аннотации данных предоставляют свойство `ErrorMessage`, которое можно использовать для указания сообщения об ошибке, отображаемого пользователю вместо стандартного сообщения об ошибке от API-интерфейса Data Annotations. Укажите желаемое значение для этого свойства в каждой аннотации данных, добавленной к модели.

Финальный класс, включающий все аннотации данных, которые обсуждались в этом разделе, должен выглядеть примерно так:

```
public class Auction
{
   [Required]
   [StringLength(50,
     ErrorMessage = "Title cannot be longer than 50 characters")]
   public string Title { get; set; }

   [Required]
   public string Description { get; set; }

   [Range(1, 10000,
     ErrorMessage = "The auction's starting price must be at least 1")]
   public decimal StartPrice { get; set; }

   public decimal CurrentPrice { get; set; }
   public DateTime EndTime { get; set; }
}
```

Теперь, когда в модели определена вся необходимая логика проверки достоверности, давайте вернемся к контроллеру и представлению и посмотрим, как отображать сообщения об ошибках проверки пользователю.

Отображение сообщений об ошибках проверки достоверности

Вы можете сказать, что добавленные правила проверки достоверности работают, поместив точку останова в действие `Create`, оправив недопустимые значения и проверив свойство `ModelState` на предмет фактического добавления ошибок проверки. Факт возвращения контроллером представления `Create` вместо добавления нового аукционного товара и перенаправления на другую страницу является еще одним доказательством того, что проверочные правила установлены корректно, а инфраструктура проверки достоверности работает. Проблема в том, что хотя представление `Create` может показывать поля с недопустимыми значениями в красной рамке, она все еще не отображает сообщения об ошибках, указывающие конкретные причины их возникновения. Давайте позаботимся об этом.

В качестве напоминания ниже приведена текущая разметка для свойства `Title`:

```
<p>
   @Html.LabelFor(model => model.Title)
   @Html.EditorFor(model => model.Title)
   @ViewData.ModelState["Title"]
</p>
```

Нам необходимо добавить к этой разметке еще одну строку для отображения любых сообщений, связанных с проверкой достоверности свойства `Title`. Простейший способ выяснить, возникли ли ошибки проверки свойства `Title`, заключается в обращении к `ModelState` напрямую — `ViewData.ModelState["Title"]` возвращает объект, содержащий коллекцию ошибок, которые относятся к свойству `Title`.

Затем можно пройти в цикле по этой коллекции, чтобы визуализировать сообщения об ошибках на странице:

```
<p>
   @Html.LabelFor(model => model.Title)
   @Html.EditorFor(model => model.Title)
   @foreach(var error in ViewData.ModelState["Title"].Errors)
   {
      <span class="error">@error.ErrorMessage</span>
   }
</p>
```

Хотя это работает довольно хорошо, ASP.NET MVC предлагает даже лучший подход к визуализации всех ошибок для заданного свойства: вспомогательный метод `Html.ValidationMessage(string modelName)`.

Вспомогательный метод `Html.ValidationMessage()` позволяет заменить весь показанный выше цикл `foreach` единственным вызовом метода и получить тот же самый результат:

```
@Html.ValidationMessageFor(model => model.Title)
```

Добавьте вызов `Html.ValidationMessage()` для каждого свойства в модели. Инфраструктура ASP.NET MVC теперь будет визуализировать все проблемы, возникающие во время проверки достоверности, прямо рядом с полями формы, к которым проверка применяется.

В дополнение к вспомогательному методу `Html.ValidationMessage()` уровня свойств, ASP.NET MVC также предоставляет вспомогательный метод `Html.ValidationSummary()`. Этот метод позволяет визуализировать все ошибки проверки достовер-

ности для формы в одном месте (например, в верхней части формы), предоставляя пользователю сводку по всем проблемам, которые должны быть устранены, чтобы форма могла быть успешно отправлена.

Вспомогательный метод `Html.ValidationSummary()` очень легко использовать — нужно просто вызвать `Html.ValidationSummary()` там, где должна располагаться сводка:

```
@using (Html.BeginForm())
{
   @Html.ValidationSummary()

   <p>
      @Html.LabelFor(model => model.Title)
      @Html.EditorFor(model => model.Title)
      @Html.ValidationMessageFor(model => model.Title)
   </p>

   <!-- Остальные поля формы... -->
}
```

Если теперь отправить недопустимые значения в полях формы, вы увидите сообщения об ошибках в двух местах (рис. 3.1): в сводке по проверке достоверности (благодаря вызову `Html.ValidationSummary()`) и рядом с полями (благодаря вызову `Html.ValidationMessage()`).

Рис. 3.1. *Отображение сообщений об ошибках посредством вспомогательного метода* `Html.ValidationSummary()`

Если вы хотите избежать отображения дублированных сообщений об ошибках, можете модифицировать вызовы `Html.ValidationMessage()` и указать короткое специальное сообщение, такое как единственный символ звездочки:

```
<p>
   @Html.LabelFor(model => model.Title)
   @Html.EditorFor(model => model.Title)
   @Html.ValidationMessageFor(model => model.Title, "*")
</p>
```

Ниже приведена полная разметка для представления `Create` после добавления всех средств проверки достоверности:

```
<h2>Create Auction</h2>
@using (Html.BeginForm())
{
   @Html.ValidationSummary()
   <p>
      @Html.LabelFor(model => model.Title)
      @Html.EditorFor(model => model.Title)
      @Html.ValidationMessageFor(model => model.Title, "*")
   </p>
   <p>
      @Html.LabelFor(model => model.Description)
      @Html.EditorFor(model => model.Description)
      @Html.ValidationMessageFor(model => model.Description, "*")
   </p>
   <p>
      @Html.LabelFor(model => model.StartPrice)
      @Html.EditorFor(model => model.StartPrice)
      @Html.ValidationMessageFor(model => model.StartPrice)
   </p>
   <p>
      @Html.LabelFor(model => model.EndTime)
      @Html.EditorFor(model => model.EndTime)
      @Html.ValidationMessageFor(model => model.EndTime)
   </p>
   <p>
      <input type="submit" value="Create" />
   </p>
}
```

Вся проверка достоверности, продемонстрированная до сих пор, производилась *на стороне сервера*, требуя полного цикла обмена между браузером и сервером для обработки каждого потенциально недопустимого запроса и выдачи в качестве ответа полностью визуализированного представления.

Хотя этот подход работает, он определенно не оптимален. В главе 4 показано, как реализовать проверку достоверности на стороне клиента, чтобы усовершенствовать этот подход и выполнять большинство (если не все) проверок прямо в браузере. Это позволит избежать дополнительных запросов к серверу, сохраняя как полосу пропускания, так и серверные ресурсы.

Резюме

В этой главе речь шла об использовании подхода Entity Framework Code First для создания и обслуживания базы данных приложения. Вы увидели, насколько легко с помощью Entity Framework устанавливать базу данных: это сводится к всего лишь нескольким строкам кода и не требует предварительного рисования диаграммы со схемой или написания SQL-запросов для моделирования и создания базы данных. Было показано, что Entity Framework работает за счет следования соглашениям, а также описаны некоторые базовые соглашения. Также кратко рассматривалось применение средства привязки моделей ASP.NET MVC для автоматического заполнения объектов состояния из входящего запроса.