ПРИЛОЖЕНИЕ Д

Управление состоянием в ASP.NET

В предшествующих двух приложениях внимание было сосредоточено на структуре и поведении страниц ASP.NET, а также на веб-элементах управления, которые они содержат. Настоящее приложение основано на этой информации и посвящено роли файла `Global.asax` и лежащего в основе типа `HttpApplication`. Вы увидите, что функциональность `HttpApplication` позволяет перехватывать многочисленные события, которые делают возможной трактовку веб-приложения как единого целого, а не набора автономных файлов `.aspx`, управляемых мастер-страницей.

В дополнение к исследованию типа `HttpApplication` здесь также рассматриваются все важные темы, касающиеся управления состоянием. Вы узнаете о роли состояния представления, переменных сеанса и приложения (включая кеш приложения), cookie-данных и API-интерфейса профилей ASP.NET (ASP.NET Profile API).

Проблема поддержки состояния

В начале приложения В было указано, что HTTP является протоколом *без запоминания состояния*. Данный факт делает разработку веб-приложений совершенно отличающейся от процесса построения исполняемой сборки. Например, при создании приложения с настольным пользовательским интерфейсом Windows можно рассчитывать на то, что переменные-члены, определенные в производном от `Form` классе, обычно будут существовать в памяти до тех пор, пока пользователь явно не завершит исполняемую программу:

```
public partial class MainWindow : Window
{
  // Данные состояния!
  private string userFavoriteCar = "Yugo";
}
```

Однако в среде World Wide Web нельзя исходить из такого же удобного предположения. Чтобы проверить сказанное, создадим новый проект пустого веб-сайта по имени `SimpleStateExample` и вставим в него веб-форму. Определим в файле отделенного кода строковую переменную уровня страницы по имени `userFavoriteCar`:

```
public partial class _Default : System.Web.UI.Page
{
  // Данные состояния?
  private string userFavoriteCar = "Yugo";
  protected void Page_Load(object sender, EventArgs e)
  {
  }
}
```

Затем построим очень простой пользовательский интерфейс (рис. Д.1).

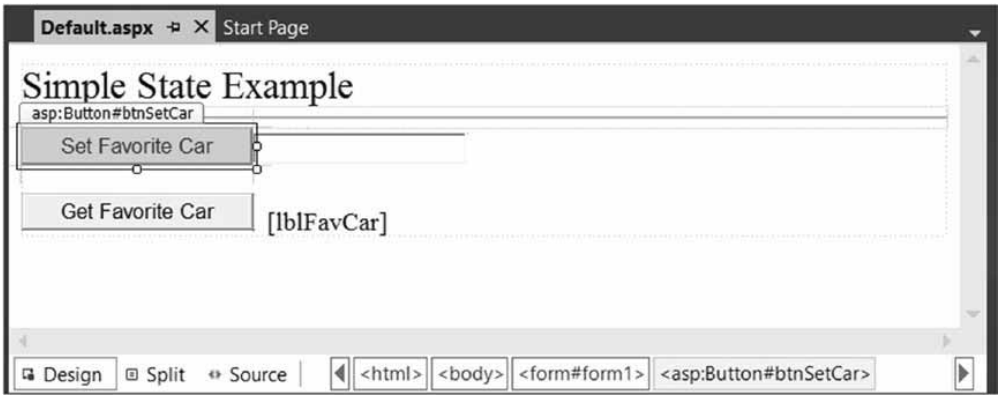

Рис. Д.1. Пользовательский интерфейс для простой страницы с состоянием

Обработчик события `Click` серверной стороны для кнопки Set Favorite Car (Установить предпочитаемый автомобиль) по имени `btnSetCar` позволяет пользователю присвоить переменной-члену типа `string` значение, находящееся внутри элемента `TextBox` (с именем `txtFavCar`):

```
protected void btnSetCar_Click(object sender, EventArgs e)
{
  // Сохранить предпочитаемый автомобиль в переменной-члене.
  userFavoriteCar = txtFavCar.Text;
}
```

Обработчик события `Click` для кнопки Get Favorite Car (Получить предпочитаемый автомобиль) по имени `btnGetCar` отображает текущее значение переменной-члена внутри элемента `Label` (с именем `lblFavCar`) страницы:

```
protected void btnGetCar_Click(object sender, EventArgs e)
{
  // Отобразить значение переменной-члена.
  lblFavCar.Text = userFavoriteCar;
}
```

При построении приложения с графическим пользовательским интерфейсом Windows вполне корректно предполагать, что после того, как пользователь установил начальное значение, оно запоминается на все время жизни настольного приложения. К сожалению, запустив созданное веб-приложение, вы обнаружите, что всякий раз, когда осуществляется обратная отправка веб-серверу (посредством щелчка на любой из кнопок), для строковой переменной `userFavoriteCar` устанавливается начальное значение `"Yugo"`. Следовательно, текст в `Label` постоянно фиксирован.

Поскольку протокол HTTP не имеет ни малейшего понятия о том, как автоматически запоминать данные после отправки HTTP-ответа, само собой разумеется, что объект `Page` уничтожается практически мгновенно. В результате, когда клиент отправляет обратно файл `.aspx`, конструируется новый объект `Page`, который сбрасывает значение всех переменных-членов уровня страницы. Безусловно, это серьезная проблема. Вообразите, до какой степени бесполезной была бы онлайновая торговля, если бы вся введенная ранее информация (вроде наименований приобретаемых товаров) отбрасывалась при каждой отправке данных веб-серверу. Когда необходимо запоминать информацию о пользователях, вошедших на сайт, приходится использовать разнообразные приемы управления состоянием.

На заметку! Описанная проблема никоим образом не ограничивается ASP.NET. Веб-приложения Java, приложения CGI, классические приложения ASP и приложения PHP также сталкиваются с необходимостью решения задачи управления состоянием.

Один из подходов к запоминанию значения строковой переменной `userFavoriteCar` между обратными отправками предусматривает сохранение этого значения в *переменной сеанса*. Состояние сеанса будет детально исследоваться далее в приложении. Тем не менее, ради полноты ниже показаны необходимые изменения для текущей страницы (обратите внимание, что закрытая переменная-член типа `string` больше не применяется, так что ее определение можно закомментировать либо вообще удалить):

```
public partial class _Default : System.Web.UI.Page
{
  // Данные состояния?
  // private string userFavoriteCar = "Yugo";

  protected void Page_Load(object sender, EventArgs e)
  {
  }

  protected void btnSetCar_Click(object sender, EventArgs e)
  {
    // Сохранить значение в переменной сеанса.
    Session["UserFavCar"] = txtFavCar.Text;
  }

  protected void btnGetCar_Click(object sender, EventArgs e)
  {
    // Получить значение из переменной сеанса.
    lblFavCar.Text = (string)Session["UserFavCar"];
  }
}
```

Если теперь запустить приложение, то значение предпочитаемого автомобиля будет сохраняться между обратными отправками благодаря объекту `HttpSessionState`, которым можно манипулировать косвенно через унаследованное свойство `Session`. Данные сеанса (которые будут подробно рассматриваться позже в приложении) — лишь один способ "запоминания" информации на веб-сайтах. В последующих разделах вы ознакомитесь с другими возможными вариантами, поддерживаемыми ASP.NET.

Исходный код. Веб-сайт `SimpleStateExample` доступен в подкаталоге `Appendix_E`.

Приемы управления состоянием ASP.NET

Инфраструктура ASP.NET предлагает несколько механизмов, которые можно использовать для сохранения информации о состоянии в веб-приложениях. Ниже перечислены распространенные варианты:

- применение состояния представления ASP.NET;
- использование состояния элемента управления ASP.NET;
- определение переменных уровня приложения;
- применение объекта кеша;
- определение переменных уровня сеанса;
- определение cookie-данных.

В дополнение к указанным приемам инфраструктура ASP.NET предоставляет готовый API-интерфейс Profile для хранения пользовательских данных на постоянной основе. Мы по очереди рассмотрим детали каждого подхода, начиная с состояния представления ASP.NET.

Роль состояния представления ASP.NET

Термин *состояние представления* уже несколько раз встречался здесь и в предшествующих приложениях, но без формального определения, поэтому давайте проясним его прямо сейчас. Без поддержки со стороны инфраструктуры разработчики веб-приложений вынуждены вручную перезаполнять значениями виджеты ввода на форме во время процесса конструирования исходящего HTTP-ответа.

В случае использования ASP.NET мы больше не обязаны вручную собирать и заново заполнять значениями виджеты HTML, т.к. исполняющая среда ASP.NET автоматически встраивает в форму скрытое поле (по имени `__VIEWSTATE`), которое будет передаваться между браузером и специфической страницей. Данные, присваиваемые полю `__VIEWSTATE`, являются закодированной по алгоритму Base64 строкой, которая содержит набор пар "имя-значение", представляющих значения всех виджетов пользовательского интерфейса на текущей странице.

За чтение входных значений из поля `__VIEWSTATE` с целью заполнения соответствующих переменных-членов в производном классе несет ответственность обработчик события `Init` базового класса `System.Web.UI.Page`. (Именно по этой причине обращаться к состоянию виджета внутри области действия обработчика события `Init` страницы в лучшем случае рискованно.)

Кроме того, непосредственно перед отправкой исходящего ответа запросившему браузеру данные `__VIEWSTATE` применяются для повторного заполнения виджетов формы. Очевидно, самая лучшая характеристика указанного аспекта ASP.NET заключается в том, что все происходит без какого-либо участия с вашей стороны. Разумеется, при необходимости всегда можно взаимодействовать, изменять или отключать такую стандартную функциональность. Чтобы понять, как это делается, давайте рассмотрим конкретный пример работы с состоянием представления.

Демонстрация работы с состоянием представления

Создадим новый проект пустого веб-сайта по имени `ViewStateApp` и добавим в него новую веб-форму под названием `Default.aspx`. Щелкнем правой кнопкой мыши на имени проекта и выберем в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet). Добавим NuGet-пакет `Microsoft.CodeDom.Providers.DotNetCompilerPlatform`, который предоставляет веб-сайту возможности C# 6. Добавим на страницу `.aspx` веб-элемент управления `ListBox` по имени `myListBox` и элемент управления `Button` по имени `btnPostback`.

На заметку! Для всех примеров в настоящем приложении понадобится добавлять NuGet-пакет `Microsoft.CodeDom.Providers.DotNetCompilerPlatform`, чтобы обеспечить поддержку возможностей C# 6.

В окне Properties (Свойства) среды Visual Studio отыщем свойство `Items` и добавим в `ListBox` четыре элемента `ListItem`, используя ассоциированное диалоговое окно. Результирующая разметка должна выглядеть так:

```
<asp:ListBox ID="myListBox" runat="server">
  <asp:ListItem>Item One</asp:ListItem>
```

```
  <asp:ListItem>Item Two</asp:ListItem>
  <asp:ListItem>Item Three</asp:ListItem>
  <asp:ListItem>Item Four</asp:ListItem>
</asp:ListBox>
```

Обратите внимание, что элементы списка `ListBox` жестко закодированы внутри файла `.aspx`. Вы уже знаете, что перед отправкой финального HTTP-ответа все определения `<asp:>` в веб-форме ASP.NET будут автоматически преобразованы в свои представления HTML (если они снабжены атрибутом `runat="server"`).

Директива `<%@ Page %>` имеет необязательный атрибут по имени `EnableViewState`, который по умолчанию установлен в `true`. Чтобы отключить поведение состояния представления, изменим директиву `<%@ Page %>` следующим образом:

```
<%@ Page Language="C#" AutoEventWireup="true"
       CodeFile="Default.aspx.cs" Inherits="_Default"
       EnableViewState ="false" %>
```

Итак, что же в точности означает отключение состояния представления? Ответ: в зависимости от обстоятельств. Исходя из предыдущего определения термина, могло бы показаться, что если отключить состояние представления для файла `.aspx`, то значения в `ListBox` не будут запоминаться между обратными отправками веб-серверу. Однако если вы запустите приложение в том виде, как есть, то можете быть удивлены, увидев, что информация в `ListBox` сохраняется независимо от того, сколько раз производится обратная отправка страницы.

На самом деле, если вы просмотрите разметку HTML, возвращенную браузеру (щелкнув правой кнопкой мыши на странице внутри браузера и выбрав в контекстном меню пункт View Source (Исходный код страницы)), то удивитесь еще больше, обнаружив, что скрытое поле `__VIEWSTATE` *по-прежнему присутствует*:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="/wEPDwUKLTM4MTM2MDM4NGRkqGC6gjEV25JnddkJiRmoIc10SIA=" />
```

Тем не менее, предположим, что элемент управления `ListBox` заполняется динамически в файле отделенного кода, а не в HTML-дескрипторе `<form>`. Для начала удалим объявления `<asp:ListItem>` из текущего файла `.aspx`:

```
<asp:ListBox ID="myListBox" runat="server">
</asp:ListBox>
```

Затем заполним список элементами внутри обработчика события `Load` в файле отделенного кода:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
  {
    // Заполнить ListBox динамически!
    myListBox.Items.Add("Item One");
    myListBox.Items.Add("Item Two");
    myListBox.Items.Add("Item Three");
    myListBox.Items.Add("Item Four");
  }
}
```

Отправив такую обновленную страницу, вы увидите, что при первом ее запросе браузером значения в `ListBox` присутствуют в том виде, в каком они были добавлены в коде. Однако после обратной отправки элемент управления `ListBox` неожиданно становится пустым. Первое правило состояния представления ASP.NET заключается в том, что его

работу можно заметить только тогда, когда есть виджеты, для которых значения генерируются динамически в коде. Если значения жестко закодированы внутри дескрипторов <form> файла .aspx, то состояние этих элементов будет всегда запоминаться между обратными отправками (даже если для заданной страницы атрибут EnableViewState установлен в false).

Если идея отключения состояния представления для всего файла .aspx кажется излишне радикальной, тогда имейте в виду, что каждый потомок базового класса System.Web.UI.Control наследует свойство EnableViewState, которое делает очень простым отключение состояния представления на поэлементной основе:

```
<asp:GridView id="myHugeDynamicallyFilledGridOfData" runat="server"
  EnableViewState="false">
</asp:GridView>
```

На заметку! Начиная с версии .NET 4.0, объемные значения данных состояния представления автоматически сжимаются, чтобы уменьшить размер скрытого поля формы __VIEWSTATE.

Добавление специальных данных состояния представления

Вдобавок к свойству EnableViewState класс System.Web.UI.Control предлагает защищенное свойство по имени ViewState. "За кулисами" оно обеспечивает доступ к объекту типа System.Web.UI.StateBag, который представляет все данные, содержащиеся в поле __VIEWSTATE. С применением индексатора типа StateBag в скрытое поле формы VIEWSTATE можно встраивать специальную информацию, используя набор пар "имя-значение". Вот простой пример:

```
protected void btnAddToVS_Click(object sender, EventArgs e)
{
  ViewState["CustomViewStateItem"] = "Some user data";
  lblVSValue.Text = (string)ViewState["CustomViewStateItem"];
}
```

Из-за того, что тип System.Web.UI.StateBag был спроектирован для оперирования с типами System.Object, при доступе к значению по заданному ключу потребуется явно приводить его к лежащему в основе типу данных (System.String в данном случае). Тем не менее, имейте в виду, что значения, помещенные в поле __VIEWSTATE, не могут быть буквально любым объектом. В частности, допустимыми типами являются только String, Integer, Boolean, ArrayList, Hashtable и массивы перечисленных типов.

Следовательно, поскольку страницы .aspx могут вставлять специальные фрагменты информации в строку __VIEWSTATE, имеет смысл выяснить, когда это может требоваться. В большинстве случаев специальные данные состояния представления лучше всего подходят для хранения пользовательских предпочтений. Скажем, можно было бы установить данные состояния представления, которые указывают, каким пользователь желает видеть интерфейс элемента GridView (например, порядок сортировки). Однако данные состояния представления не слишком хорошо подходят для хранения полномасштабных пользовательских данных, таких как элементы в корзине для покупок или кешированные объекты DataSet. Когда нужно хранить сложную информацию подобного рода, лучше работать с данными сеанса или данными приложения. Но прежде чем перейти к их рассмотрению, следует прояснить роль файла Global.asax.

Исходный код. Веб-сайт ViewStateApp доступен в подкаталоге Appendix_E.

Роль файла Global.asax

К настоящему моменту приложение ASP.NET может выглядеть всего лишь набором файлов .aspx и соответствующих веб-элементов управления. Наряду с тем, что веб-приложение можно строить, просто связывая вместе набор веб-страниц, скорее всего, вам понадобится способ взаимодействия с веб-приложением как с единым целым. Для такой цели в веб-приложение ASP.NET можно включить дополнительный файл Global.asax через пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент), как показано на рис. Д.2. (Обратите внимание, что здесь выбирается элемент Global Application Class (Глобальный класс приложения).)

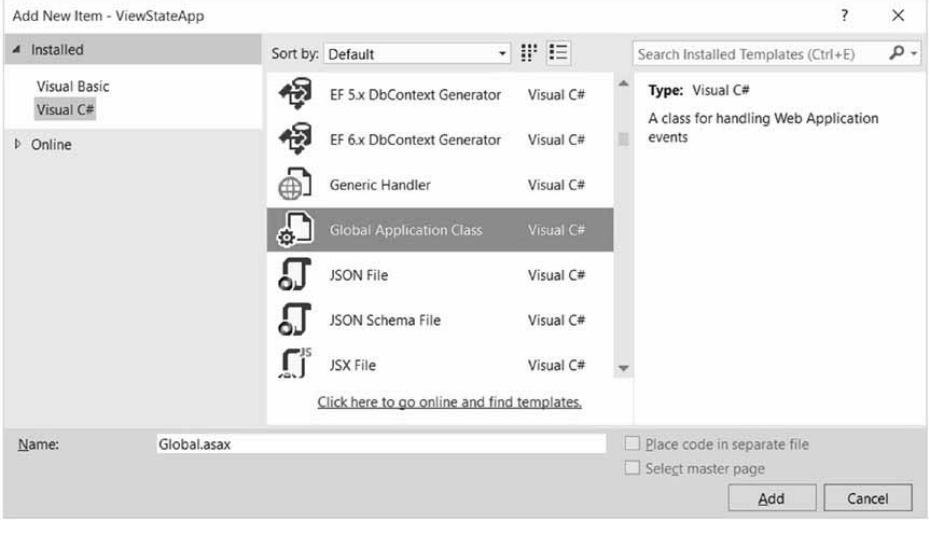

Рис. Д.2. Добавление файла Global.asax

Попросту говоря, Global.asax очень похож на запускаемый двойным щелчком файл .exe, который можно получить в мире ASP.NET, в том смысле, что этот тип представляет поведение времени выполнения самого веб-сайта. После вставки файла Global.asax в веб-проект вы увидите, что в нем находится всего лишь блок <script>, содержащий набор обработчиков событий:

```
<%@ Application Language="C#" %>

<script runat="server">
  void Application_Start(object sender, EventArgs e)
  {
    // Код, выполняемый при запуске приложения.
  }

  void Application_End(object sender, EventArgs e)
  {
    // Код, выполняемый при прекращении работы приложения.
  }

  void Application_Error(object sender, EventArgs e)
  {
    // Код, выполняемый при возникновении необработанной ошибки.
  }
```

```
  void Session_Start(object sender, EventArgs e)
  {
    // Код, выполняемый при запуске нового сеанса.
  }
  void Session_End(object sender, EventArgs e)
  {
    // Код, выполняемый при завершении сеанса.
    // Примечание. Событие Session_End инициируется, только если в файле
    // Web.config режим состояния сеанса установлен в InProc.
    // Если режим сеанса установлен в StateServer или SQLServer,
    // то данное событие не возникает.
  }
</script>
```

Тем не менее, внешность может быть обманчивой. Во время выполнения код внутри данного блока `<script>` организуется в класс, производный от `System.Web.HttpApplication`. Следовательно, в любом из предоставленных обработчиков событий можно получать доступ к членам родительского класса посредством ключевых слов `this` или `base`.

Как уже упоминалось, определенные внутри `Global.asax` члены представляют собой обработчики событий, которые позволяют взаимодействовать с событиями уровня приложения (и уровня сеанса). Обработчики событий кратко описаны в табл. Д.1.

Таблица Д.1. Основные обработчики событий в файле `Global.asax`

Обработчик события	Описание
`Application_Start()`	Этот обработчик событий вызывается в самом начале при запуске веб-приложения. Таким образом, за время жизни веб-приложения данное событие будет инициировано только однократно. Представляет собой идеальное место для определения данных уровня приложения, применяемых повсюду в веб-приложении
`Application_End()`	Этот обработчик событий вызывается при прекращении работы приложения, которое происходит, когда последний пользователь покидает приложение по тайм-ауту или приложение вручную завершается в IIS
`Session_Start()`	Этот обработчик событий вызывается, когда новый пользователь входит в приложение. Здесь можно устанавливать все специфические для пользователя данные, которые нужно сохранить между обратными отправками
`Session_End()`	Этот обработчик событий вызывается при завершении пользовательского сеанса (обычно по истечении предопределенного тайм-аута)
`Application_Error()`	Это глобальный обработчик ошибок, который вызывается, когда веб-приложение сгенерировало необработанное исключение

Глобальный обработчик исключений “последнего шанса”

Давайте сначала обсудим роль обработчика события `Application_Error()`. Вспомните, что определенная страница может обрабатывать событие `Error` с целью перехвата любого необработанного исключения, которое произошло в области действия самой страницы. Вдобавок обработчик события `Application_Error()` — это последнее место, где можно обработать исключение, которое не было обработано страницей. Как и в случае события `Error` уровня страницы, обращаться к определенному исключению `System.Exception` можно с использованием унаследованного свойства `Server`:

```
void Application_Error(object sender, EventArgs e)
{
  // Получить необработанную ошибку.
  Exception ex = Server.GetLastError();

  // Обработать ошибку...

  // По завершении обработки очистить ошибку.
  Server.ClearError();
}
```

Поскольку обработчик событий `Application_Error()` является обработчиком исключений "последнего шанса" для веб-приложения, довольно часто этот метод реализуется так, что пользователь переносится на заранее определенную страницу ошибки на сервере. Другие распространенные действия могут включать отправку сообщения электронной почты веб-администратору или запись во внешний журнал ошибок.

Базовый класс `HttpApplication`

Как упоминалось ранее, сценарий `Global.asax` динамически генерируется в виде класса, производного от базового класса `System.Web.HttpApplication`, который предлагает часть того же рода функциональности, что и класс `System.Web.UI.Page` (без видимого пользовательского интерфейса). В табл. Д.2 документированы основные свойства класса `HttpApplication`.

Таблица Д.2. Основные свойства класса `System.Web.HttpApplication`

Свойство	Описание
`Application`	Это свойство позволяет взаимодействовать с данными уровня приложения посредством типа `HttpApplicationState`
`Request`	Это свойство позволяет взаимодействовать с входящим запросом HTTP с применением лежащего в основе объекта `HttpRequest`
`Response`	Это свойство позволяет взаимодействовать с исходящим ответом HTTP, используя лежащий в основе объект `HttpResponse`
`Server`	Это свойство позволяет получить встроенный серверный объект для текущего запроса с применением лежащего в основе объекта `HttpServerUtility`
`Session`	Это свойство позволяет взаимодействовать с данными уровня сеанса, используя лежащий в основе объект `HttpSessionState`

И снова, учитывая, что файл `Global.asax` явно не документирует `HttpApplication` в качестве лежащего в основе базового класса, важно помнить, что все правила отношения "является" на самом деле будут применены.

Разница между состоянием приложения и состоянием сеанса

В ASP.NET состояние приложения поддерживается экземпляром типа `HttpApplicationState`. Класс `HttpApplicationState` позволяет разделять глобальную информацию между всеми пользователями (и всеми страницами) приложения ASP.NET. Можно не только разделять все данные приложения между всеми пользователями сайта; если значение элемента данных уровня приложения изменяется, то новое значение становится видимым всем пользователям после того, как они выполнят следующую обратную отправку.

С другой стороны, состояние сеанса используется, чтобы запоминать информацию для специфического пользователя (подобную товарам в корзине покупок). Физически состояние сеанса пользователя представляется типом класса `HttpSessionState`. Когда новый пользователь входит в веб-приложение ASP.NET, исполняющая среда автоматически назначает ему новый идентификатор сеанса, который по умолчанию устаревает после 20-минутного отсутствия активности. Таким образом, если на сайт вошло 20 000 пользователей, то существует 20 000 отдельных объектов `HttpSessionState`, каждому из которых автоматически назначен уникальный идентификатор сеанса. Отношения между веб-приложением и веб-сеансами демонстрируется на рис. Д.3.

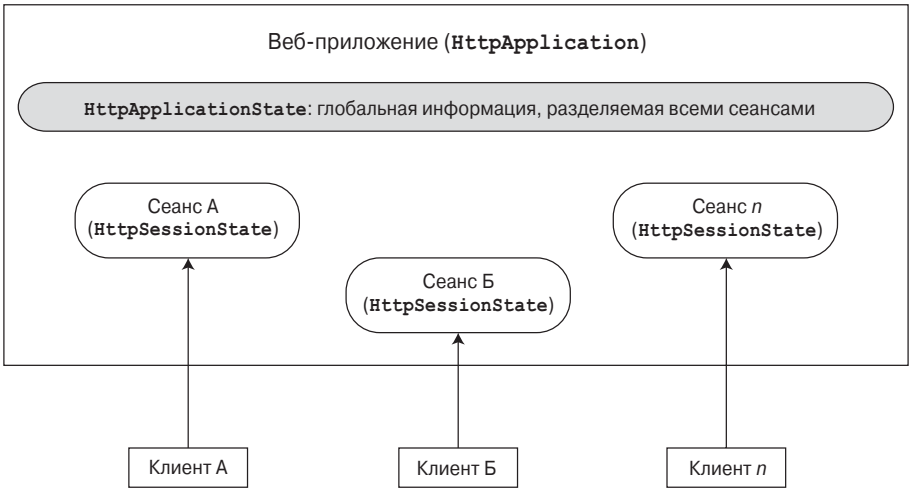

Рис. Д.3. Отличие между состоянием приложения и состоянием сеанса

Поддержка данных состояния уровня приложения

Тип `HttpApplicationState` позволяет разработчикам разделять глобальную информацию между множеством пользователей в приложении ASP.NET. В табл. Д.3 описаны некоторые основные члены этого типа.

Таблица Д.3. Члены типа `HttpApplicationState`

Член	Описание
`Add()`	Этот метод позволяет добавить в объект `HttpApplicationState` новую пару "имя-значение". Обратите внимание, что данный метод обычно *не* применяется, а вместо него используется индексатор класса `HttpApplicationState`
`AllKeys`	Это свойство возвращает массив объектов `string`, которые представляют все имена в объекте `HttpApplicationState`
`Clear()`	Этот метод очищает все элементы в объекте `HttpApplicationState`. Его функциональность эквивалентна функциональности метода `RemoveAll()`
`Count`	Это свойство получает количество элементов, хранимых в объекте `HttpApplicationState`
`Lock()`, `Unlock()`	Эти два метода применяются, когда необходимо изменить набор переменных приложения в безопасной к потокам манере
`RemoveAll()`, `Remove()`, `RemoveAt()`	Эти методы удаляют определенный элемент (по строковому имени) из объекта `HttpApplicationState`. Метод `RemoveAt()` удаляет элемент по числовому индексу

Чтобы проиллюстрировать работу с состоянием приложения, создадим проект пустого веб-сайта по имени AppState (и вставьте в него веб-форму). Затем добавим новый файл Global.asax. Когда создаются данные-члены, которые могут разделяться между всеми пользователями, необходимо устанавливать пары "имя-значение". В большинстве случаев самым естественным местом для этого является обработчик события Application_Start() в файле Global.asax.cs:

```
void Application_Start(Object sender, EventArgs e)
{
  // Установить некоторые переменные приложения.
  Application["SalesPersonOfTheMonth"] = "Chucky";
  Application["CurrentCarOnSale"] = "Colt";
  Application["MostPopularColorOnLot"] = "Black";
}
```

На протяжении жизненного цикла веб-приложения (т.е. до тех пор, пока работа приложения не будет прекращена вручную либо не истечет тайм-аут последнего пользователя) любой пользователь на любой странице может обращаться к таким значениям, когда они ему нужны. Предположим, что есть страница, которая отображает текущую скидку на автомобиль в элементе Label посредством обработчика события Click кнопки:

```
protected void btnShowCarOnSale_Click(object sender, EventArgs arg)
{
  lblCurrCarOnSale.Text = string.Format("Sale on {0}'s today!",
    (string)Application["CurrentCarOnSale"]);
}
```

Как и в случае свойства ViewState, возвращаемое из объекта HttpApplicationState значение потребуется привести к корректному типу, потому что свойство Application оперирует с общими типами System.Object.

Поскольку свойство Application может хранить объект любого типа, само собой разумеется, что внутрь состояния приложения можно помещать объекты специальных типов (или любые объекты .NET). Предположим, что в строго типизированном классе CarLotInfo решено поддерживать три переменные приложения:

```
public class CarLotInfo
{
  public CarLotInfo(string salesPerson, string currentCar, string mostPopular)
  {
    SalesPersonOfTheMonth = salesPerson;
    CurrentCarOnSale = currentCar;
    MostPopularColorOnLot = mostPopular;
  }
  public string SalesPersonOfTheMonth { get; set; }
  public string CurrentCarOnSale { get; set; }
  public string MostPopularColorOnLot { get; set; }
}
```

Имея такой вспомогательный класс, обработчик события Application_Start() можно было бы модифицировать следующим образом:

```
void Application_Start(Object sender, EventArgs e)
{
  // Поместить специальный объект в раздел данных приложения.
  Application["CarSiteInfo"] =
    new CarLotInfo("Chucky", "Colt", "Black");
}
```

Затем можно было бы получать доступ к информации с использованием открытых полей данных внутри обработчика события Click серверной стороны для элемента управления Button по имени btnShowAppVariables:

```
protected void btnShowAppVariables_Click(object sender, EventArgs e)
{
 CarLotInfo appVars =
    ((CarLotInfo)Application["CarSiteInfo"]);
 string appState = $"<li>Car on sale: { appVars.CurrentCarOnSale }</li>"
 appState += $"<li>Most popular color: { appVars.MostPopularColorOnLot }</li>";
 appState += $"<li>Big shot SalesPerson: { appVars.SalesPersonOfTheMonth }</li>"
 lblAppVariables.Text = appState;
}
```

С учетом того, что текущие данные о продаже машины теперь доступны из специального класса, обработчик события Click кнопки btnShowCarOnSale также должен быть изменен:

```
protected void btnShowCarOnSale_Click(object sender, EventArgs e)
{
  lblCurrCarOnSale.Text =
    $"Sale on {((CarLotInfo)Application["CarSiteInfo"]).CurrentCarOnSale
}'s today!";
}
```

Модификация данных приложения

Во время выполнения веб-приложения можно программно обновлять либо удалять любые или вообще все элементы данных уровня приложения с применением членов типа HttpApplicationState. Например, чтобы удалить специфический элемент данных, нужно просто вызвать метод Remove(). Если необходимо уничтожить все данные уровня приложения, тогда следует вызвать метод RemoveAll().

```
private void CleanAppData()
{
  // Удалить один элемент по строковому имени.
  Application.Remove("SomeItemIDontNeed");

  // Уничтожить все данные приложения!
  Application.RemoveAll();
}
```

Чтобы изменить значение существующего элемента данных уровня приложения, понадобится только сделать новое присваивание интересующему элементу данных. Предположим, что страница теперь имеет элемент управления Button, который позволяет пользователю изменять текущего преуспевающего продавца, прочитав его имя из элемента TextBox по имени txtNewSP. Обработчик события Click выглядит вполне ожидаемо:

```
protected void btnSetNewSP_Click(object sender, EventArgs e)
{
  // Установить нового продавца.
  ((CarLotInfo)Application["CarSiteInfo"]).SalesPersonOfTheMonth
    = txtNewSP.Text;
}
```

Запустив веб-приложение, после щелчка на новой кнопке обнаружится, что элемент данных уровня приложения обновился. Более того, поскольку переменные приложения доступны всем пользователям на любой странице веб-приложения, после запуска трех

или четырех экземпляров веб-браузера можно заметить, что при изменении текущего продавца в одном экземпляре все остальные экземпляры отобразят новое значение, как только произойдет обратная отправка. Возможный вывод показан на рис. Д.4.

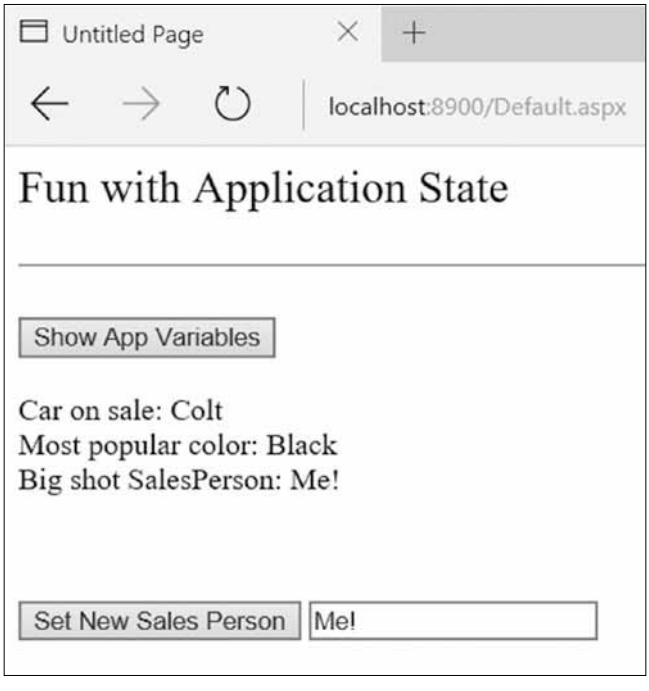

Рис. Д.4. Отображение данных приложения

Имейте в виду, что в ситуации, при которой набор переменных уровня приложения должен быть обновлен как единое целое, возникает риск повредить данные, т.к. формально возможно, что данные уровня приложения будут изменяться именно в тот момент, когда другой пользователь пытается получить к ним доступ. Вы могли бы избрать долгий путь и вручную реализовать блокировку с использованием потоковых примитивов из пространства имен `System.Threading`, но тип `HttpApplicatiobState` имеет два метода, `Lock()` и `Unlock()`, которые автоматически обеспечивают безопасность в отношении потоков:

```
// Безопасно обратиться к связанным данным приложения.
Application.Lock();
Application["SalesPersonOfTheMonth"] = "Maxine";
Application["CurrentBonusedEmployee"] = Application["SalesPersonOfTheMonth"];
Application.UnLock();
```

Обработка прекращения работы веб-приложения

Тип `HttpApplicationState` рассчитан на сохранение значений содержащихся в нем элементов до тех пор, пока не произойдет одна из двух ситуаций: у последнего пользователя сайта истечет время тайм-аута (или он выйдет явно) либо кто-то вручную прекратит работу сайта на веб-сервере IIS. В любом случае будет автоматически вызван метод `Application_End()` производного от `HttpApplication` типа. Внутрь этого обработчика событий можно поместить необходимый код очистки:

```
void Application_End(Object sender, EventArgs e)
{
  // Записать текущие переменные в базу данных или куда-нибудь еще.
}
```

Исходный код. Веб-сайт AppState доступен в подкаталоге Appendix_E.

Работа с кешем приложения

Инфраструктура ASP.NET предоставляет еще один более гибкий способ для обработки данных уровня приложения. Как вы помните, значения внутри объекта HttpApplicationState остаются в памяти до тех пор, пока веб-приложение активно и с ним производятся действия. Однако иногда может возникнуть необходимость в хранении фрагмента данных приложения в течение только определенного периода времени. Например, нужно получить ADO.NET-объект DataSet, который действителен на протяжении пяти минут. По истечении этого времени требуется получить обновленный объект DataSet для учета всех возможных изменений данных. Наряду с тем, что решить такую задачу формально возможно с применением объекта HttpApplicationState и реализованного вручную мониторинга, все значительно упростится за счет использования кеша приложения ASP.NET.

Объект System.Web.Caching.Cache в ASP.NET (к которому можно обратиться через свойство Context.Cache) позволяет определить объекты, доступные всем пользователям на всех страницах в течение фиксированного периода времени. В простейшей форме взаимодействие с кешем выглядит идентичным взаимодействию с типом HttpApplicationState:

```
// Добавить элемент в кеш.
// Этот элемент *не* устареет.
Context.Cache["SomeStringItem"] = "This is the string item";
// Получить элемент из кеша.
string s = (string)Context.Cache["SomeStringItem"];
```

На заметку! Для доступа к кешу из Global.asax необходимо применять свойство Context. Тем не менее, внутри области определения типа, производного от System.Web.UI.Page, к объекту Cache можно обращаться напрямую через свойство Cache страницы.

Помимо индексатора в классе System.Web.Caching.Cache определено совсем небольшое количество членов. Метод Add() можно использовать для вставки в кеш нового элемента, который в текущий момент не определен (если указанный элемент уже присутствует, тогда метод Add() ничего не делает). Метод Insert() также помещает элемент в кеш, но если элемент уже определен, то текущий элемент заменяется новым. Поскольку именно такое поведение обычно ожидается, мы сосредоточим внимание исключительно на методе Insert().

Использование кеширования данных

Давайте рассмотрим пример. Создадим новый проект пустого веб-сайта по имени CacheState и добавим в него веб-форму и файл Global.asax. Подобно любому элементу данных уровня приложения, поддерживаемому типом HttpApplicationState, кеш может хранить объекты производных от System.Object типов и часто заполняется внутри обработчика события Application_Start(). Целью настоящего примера будет автоматическое обновление содержимого DataSet каждые 15 секунд. Интересующий

нас объект DataSet будет содержать текущий набор записей из таблицы Inventory базы данных AutoLot, созданной во время обсуждения ADO.NET.

С учетом сказанного выше установим ссылку на сборку AutoLotDAL.dll (см. приложение В; сборка также включена в состав загружаемого кода примеров для настоящего приложения). Добавим к веб-сайту инфраструктуру Entity Framework (щелкнув правой кнопкой мыши на имени проекта и выбрав в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet)) и поместим в файл Web.config следующую строку подключения (в зависимости от установки SQL Server Express она может отличаться):

```
<connectionStrings>
  <add name="AutoLotConnection" connectionString="data source=(local)\
SQLEXPRESS2014;initial catalog=AutoLot;integrated security=True;MultipleActive
ResultSets=True;App=EntityFramework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Затем модифицируем файл Global.asax, как показано ниже:

```
<%@ Application Language="C#" %>
<%@ Import Namespace = "AutoLotDAL.Repos" %>

<script runat="server">
  // Определить статическую переменную-член Cache.
  static Cache _theCache;

  void Application_Start(Object sender, EventArgs e)
  {
    // Присвоить значение статической переменной theCache.
    _theCache = Context.Cache;

    // При запуске приложения прочитать текущие записи
    // из таблицы Inventory базы данных AutoLot.
    var theCars = new InventoryRepo().GetAll();

    // Сохранить DataTable в кеше.
    _theCache.Insert("CarList",
      theCars,
      null,
      DateTime.Now.AddSeconds(15),
      Cache.NoSlidingExpiration,
      CacheItemPriority.Default,
      UpdateCarInventory);
  }

  // Целевой метод для делегата CacheItemRemovedCallback.
  static void UpdateCarInventory(string key, object item,
    CacheItemRemovedReason reason)
  {
    var theCars = new InventoryRepo().GetAll();
    // Сохранить в кеше.
    _theCache.Insert("CarList",
      theCars,
      null,
      DateTime.Now.AddSeconds(15),
      Cache.NoSlidingExpiration,
      CacheItemPriority.Default,
      UpdateCarInventory);
  }
</script>
```

Первым делом обратите внимание на определение статической переменной-члена Cache. Причина в том, что были определены два статических члена, которым необходим доступ к объекту Cache. Вспомните, что статические методы не имеют доступа к унаследованным членам, а потому применять свойство Context нельзя!

Внутри обработчика события Application_Start() сначала получается список записей Inventory, который затем помещается в кеш приложения. Как и можно было предположить, метод Context.Cache.Insert() имеет несколько перегруженных версий. Здесь указывается значение для каждого возможного параметра. Взгляните на следующий прокомментированный вызов Insert():

```
_theCache.Insert("CarList",          // Имя для идентификации элемента в кеше.
  theCars,                           // Объект, помещаемый в кеш.
  null,                              // Есть ли зависимости у этого объекта?
  DateTime.Now.AddSeconds(15),       // Абсолютное время устаревания.
  Cache.NoSlidingExpiration,         // Не использовать скользящее устаревание
                                     //  (см. ниже).
  CacheItemPriority.Default,         // Уровень приоритета элемента кеша.
  UpdateCarInventory);               // Делегат для события CacheItemRemove.
```

Первые два параметра просто составляют пару “имя-значение” для элемента. Третий параметр позволяет определить объект CacheDependency (который в данном случае равен null, т.к. реализация IList<Inventory> ни от чего не зависит).

Параметр DateTime.Now.AddSeconds(15) задает абсолютное время устаревания. Это значит, что элемент гарантированно будет удален из кеша через 15 секунд. Абсолютное время устаревания удобно для элементов данных, которые нуждаются в постоянном обновлении (вроде биржевых показателей).

Параметр Cache.NoSlidingExpiration указывает, что элемент кеша не применяет скользящее устаревание. Скользящее устаревание представляет собой способ сохранения элемента в кеше на протяжении минимум определенного периода времени. Например, если установить значение скользящего устаревания в 60 секунд для элемента кеша, то он будет находиться там, по меньшей мере, 60 секунд. Если в течение этого времени любая веб-страница обращается к элементу кеша, тогда таймер сбрасывается, и существование элемента кеша продлевается еще на 60 секунд. Если же в заданный промежуток времени обращения отсутствуют, то элемент из кеша удаляется. Скользящее устаревание удобно для данных, которые требуют высоких затрат (времени) на генерацию, но не слишком часто используются веб-страницами.

Обратите внимание, что для отдельного элемента кеша указывать одновременно абсолютное устаревание и скользящее устаревание не допускается. Должно быть установлено либо абсолютное устаревание (с помощью Cache.NoSlidingExpiration), либо скользящее устаревание (посредством Cache.NoAbsoluteExpiration).

Наконец, как видно из сигнатуры метода UpdateCarInventory(), делегат CacheItemRemovedCallback может вызывать только методы с такой сигнатурой:

```
void UpdateCarInventory(string key, object item, CacheItemRemovedReason reason)
{
}
```

Итак, теперь при запуске приложения объект DataTable заполняется и кешируется. Каждые 15 секунд объект DataTable очищается, обновляется и заново вставляется в кеш. Чтобы увидеть это в действии, понадобится создать страницу, которая позволит осуществлять некоторое взаимодействие с пользователем.

Модификация файла .aspx

На рис. Д.5 показан пользовательский интерфейс, который позволяет вводить необходимые данные для вставки новой записи в базу данных (посредством трех элементов управления `TextBox`). Обработчик события `Click` для единственного элемента управления `Button` будет закодирован с целью поддержки манипуляций в базе данных. Наконец, вдобавок к описательным элементам `Label` элемент управления `GridView` в нижней части страницы будет применяться для отображения набора текущих записей в таблице `Inventory`.

CarId	Make	Color	PetName
1	VW1	Black	Zippy
2	Ford	Rust	Rusty
3	Saab	Black	Mel
4	Yugo	Yellow	Clunker
5	BMW	Black	Bimmer
6	BMW	Green	Hank
7	BMW	Pink	Pinky
13	Pinto	Black	Pete
45	BMW	Blue	Blueberry

Рис. Д.5. Графический пользовательский интерфейс приложения, работающего с кешем

В обработчике события `Load` страницы сконфигурируем элемент управления `GridView` для отображения текущих кешированных данных при первом получении страницы пользователем (не забыв импортировать в файл кода пространства имен `AutoLotDAL.Models` и `AutoLotDAL.Repos`):

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
  {
    carsGridView.DataSource = (IList<Inventory>)Cache["AppDataTable"];
    carsGridView.DataBind();
  }
}
```

В обработчике события `Click` кнопки Add This Car (Добавить этот автомобиль) вставим новую запись в базу данных `AutoLot`, используя тип `InventoryRepo`. После того как запись будет вставлена, вызовем вспомогательный метод по имени `RefreshGrid()`, который обновит пользовательский интерфейс:

```
protected void btnAddCar_Click(object sender, EventArgs e)
{
  // Обновить таблицу Inventory и вызвать RefreshGrid().
  new InventoryRepo().Add(new Inventory()
  {
    Color = txtCarColor.Text,
    Make = txtCarMake.Text,
    PetName = txtCarPetName.Text
  });
  RefreshGrid();
}

private void RefreshGrid()
{
  carsGridView.DataSource = new InventoryRepo().GetAll();
  carsGridView.DataBind();
}
```

Чтобы проверить работу кеша, начнем с запуска текущей программы (нажав <Ctrl+F5>) и скопируем URL, появившийся в браузере, в буфер обмена. Запустим второй экземпляр браузера и вставим скопированный URL в его строку адреса. На экране должны быть видны два экземпляра браузера с загруженной страницей `Default.aspx`, отображающей идентичные данные.

В первом экземпляре браузера добавим новую запись об автомобиле. Очевидно, что в результате получится обновленный элемент управления `GridView`, отображаемый в браузере, который инициировал обратную отправку.

Во втором экземпляре браузера щелкнем на кнопке обновления (или нажмите <F5>). Новая запись об автомобиле должна появиться не сразу, т.к. обработчик события `Page_Load` читает напрямую из кеша. (Если новая запись видна, то 15 секунд истекли. Нужно либо действовать быстрее, либо увеличить период времени, в течение которого `DataTable` остается в кеше.) Подождем несколько секунд и еще раз щелкнем на кнопке обновления во втором экземпляре браузера. Теперь должен появиться новый элемент, поскольку данные в кеше устарели, и целевой метод делегата `CacheItemRemoveCallback` автоматически обновил кешированные данные.

Таким образом, главное преимущество типа `Cache` заключается в том, что появляется шанс отреагировать на удаление элемента из кеша. В приведенном примере определенно можно было бы избежать применения `Cache` и просто заставить обработчик события `Page_Load` всегда читать прямо из базы данных `AutoLot` (но тогда работа страницы существенно замедлилась бы). Тем не менее, идея должна быть ясна: кеш позволяет автоматически обновлять данные, используя механизм кеширования.

Исходный код. Веб-сайт `CacheState` доступен в подкаталоге `Appendix_E`.

Поддержка данных сеанса

Итак, исследование данных уровня приложения и кешированных данных завершено. Теперь давайте выясним роль данных, специфичных для пользователя. Как уже упоминалось, *сеанс* — это просто взаимодействие конкретного пользователя с веб-приложением, которое представлено уникальным объектом `HttpSessionState`. Чтобы поддерживать информацию о состоянии для конкретного пользователя, можно применять свойство `Session` в классе веб-страницы или в файле `Global.asax`. Классическим примером необходимости в поддержке данных пользователя является корзина покупок. Если 10 человек зашли в электронный магазин, тогда каждый из них будет иметь уникальный набор наименований товаров, который он намерен приобрести, и такие данные должны поддерживаться.

Когда новый пользователь входит в веб-приложение, исполняющая среда .NET автоматически назначает ему уникальный идентификатор сеанса, служащий для идентификации данного пользователя. С каждым идентификатором сеанса связан специальный экземпляр типа `HttpSessionState`, в котором хранятся специфичные для пользователя данные. Вставка или извлечение данных сеанса синтаксически идентично манипулированию данными приложения, например:

```
// Добавить/извлечь данные сеанса для текущего пользователя.
Session["DesiredCarColor"] = "Green";
string color = (string) Session["DesiredCarColor"];
```

В файле `Global.asax` можно перехватывать момент начала и конца сеанса посредством обработчиков событий `Session_Start()` и `Session_End()`. Внутри `Session_Start()` допускается свободно создавать любые элементы данных, специфичные для пользователя, в то время как `Session_End()` позволяет выполнять любую работу, которая может требоваться при завершении пользовательского сеанса.

```
<%@ Application Language="C#" %>
...
void Session_Start(Object sender, EventArgs e)
{
  // Новый сеанс! При необходимости выполнить подготовительные действия.
}
void Session_End(Object sender, EventArgs e)
{
  // Пользователь вышел или отключен по тайм-ауту.
  // При необходимости выполнить очистку.
}
```

Подобно состоянию приложения состояние сеанса может хранить объекты любых производных от `System.Object` типов, включая специальные классы. Например, предположим, что создан новый проект пустого веб-сайта (по имени `SessionState`), в котором определен класс `UserShoppingCart`:

```
public class UserShoppingCart
{
  public string DesiredCar {get; set;}
  public string DesiredCarColor {get; set;}
  public float DownPayment {get; set;}
  public bool IsLeasing {get; set;}
  public DateTime DateOfPickUp {get; set;}
  public override string ToString() =>
    $"Car: {DesiredCar}<br>Color: {DesiredCarColor}<br>$ Down: {DownPayment}" +
    $"<br>Lease: {IsLeasing}<br>Pick-up Date: {DateOfPickUp.ToShortDateString()}";
}
```

Вставим в проект файл `Global.asax`. Внутри обработчика `Session_Start()` теперь каждому пользователю можно назначить новый экземпляр класса `UserShoppingCart`:

```
void Session_Start(Object sender, EventArgs e)
{
  Session["UserShoppingCartInfo"] = new UserShoppingCart();
}
```

Во время посещения пользователем веб-страниц можно брать экземпляр `UserShoppingCart` и заполнять его поля специфичными для пользователя данными. Предположим, что есть простая страница `.aspx`, которая определяет набор элементов управления для ввода, соответствующих полям типа `UserShoppingCart`, элемент управления `Button`, используемый для установки значений, и два элемента `Label`, предназначенные для отображения идентификатора сеанса пользователя и информации о сеансе (рис. Д.6).

Рис. Д.6. Графический пользовательский интерфейс приложения, демонстрирующего работу с сеансами

Обработчик события `Click` серверной стороны для элемента управления `Button` довольно прост (он извлекает значения из элементов `TextBox` и отображает данные корзины покупок в элементе `Label`):

```
protected void btnSubmit_Click(object sender, EventArgs e)
{
  // Установить предпочтения для текущего пользователя.
  var cart = (UserShoppingCart)Session["UserShoppingCartInfo"];
  cart.DateOfPickUp = myCalendar.SelectedDate;
  cart.DesiredCar = txtCarMake.Text;
  cart.DesiredCarColor = txtCarColor.Text;
  cart.DownPayment = float.Parse(txtDownPayment.Text);
  cart.IsLeasing = chkIsLeasing.Checked;
  lblUserInfo.Text = cart.ToString();
  Session["UserShoppingCartInfo"] = cart;
}
```

Внутри метода `Session_End()` можно сохранить поля `UserShoppingCart` в базе данных или где-нибудь еще (как будет показано в конце приложения, API-интерфейс Profile делает это автоматически). Кроме того, можно реализовать метод `Session_Error()` для перехвата любого ошибочного ввода (или задействовать разнообразные элементы управления проверкой достоверности на странице `Default.aspx` для обработки ошибок такого рода).

После запуска двух или трех экземпляров браузера с одним и тем же URL обнаружится, что каждый пользователь может наполнять собственную корзину покупок, которая отображается на его уникальный экземпляр класса `HttpSessionState`.

Дополнительные члены класса `HttpSessionState`

Помимо индексатора в классе `HttpSessionState` определены другие интересные члены. Свойство `SessionID` возвращает уникальный идентификатор текущего пользователя. Если желательно увидеть в этом примере автоматически назначенный идентификатор сеанса, тогда следует обработать событие `Load` страницы, как показано ниже:

```
protected void Page_Load(object sender, EventArgs e)
{
  lblUserID.Text = $"Here is your ID: { Session.SessionID }";
}
```

Методы `Remove()` и `RemoveAll()` могут применяться для очистки элементов экземпляра `HttpSessionState`, связанного с пользователем:

```
Session.Remove("SomeItemWeDontNeedAnymore");
```

В классе `HttpSessionState` также определен набор членов, которые управляют политикой устаревания текущего сеанса. По умолчанию каждый пользователь располагает 20 минутами отсутствия активности, прежде чем объект `HttpSessionState` будет уничтожен. Таким образом, если пользователь входит в веб-приложение (и, следовательно, получает уникальный идентификатор сеанса), но на протяжении 20 минут не проявляет активности, то исполняющая среда предполагает, что пользователь больше не заинтересован в сайте и уничтожает его данные сеанса. Период устаревания сеанса можно изменить, установив его для каждого пользователя индивидуально с помощью свойства `Timeout`. Наиболее подходящим местом для этого является метод `Session_Start()`:

```
void Session_Start(Object sender, EventArgs e)
{
  // Для каждого пользователя установить 5-минутный период отсутствия активности.
  Session.Timeout = 5;
  Session["UserShoppingCartInfo"]
    = new UserShoppingCart();
}
```

На заметку! Если настраивать значение `Timeout` для каждого пользователя не требуется, тогда можно изменить 20-минутное стандартное значение для всех пользователей через атрибут `timeout` элемента `<sessionState>` внутри файла `Web.config` (рассматривается в конце приложения).

Преимущество свойства `Timeout` связано с возможностью установки значения тайм-аута для каждого пользователя по отдельности. Предположим, что создано веб-приложение, которое позволяет пользователям платить дифференцированную плату за различные уровни членства. Мы могли бы указать, что привилегированные пользователи должны иметь тайм-аут длительностью в один час, тогда как обычные — только 30 секунд. Такая возможность вызывает вопрос: как запомнить специфическую для пользователя информацию (вроде текущего уровня членства) между визитами на сайт? Один из вариантов предусматривает использование типа `HttpCookie`.

Исходный код. Веб-сайт `SessionState` доступен в подкаталоге `Appendix_E`.

Cookie-наборы

Следующий прием управления состоянием предусматривает сохранение данных внутри *cookie-набора*, который часто реализуется как текстовый файл (или набор файлов) на машине пользователя. Когда пользователь входит на определенный сайт, браузер проверяет наличие на пользовательской машине cookie-файла для конкретного URL, и если файл присутствует, то добавляет его данные к запросу HTTP.

Принимающая веб-страница серверной стороны может затем прочитать cookie-данные для создания графического интерфейса на основе предпочтений текущего пользователя. Наверняка вы замечали, что после посещения одного из любимых веб-сайтов он каким-то образом "знает" разновидность содержимого, которое вы предпочитаете просматривать. Причина (отчасти) в том, что в cookie-наборах, хранящихся на вашем компьютере, содержится информация, которая касается этого веб-сайта.

На заметку! Точное местоположение cookie-файлов зависит от применяемого браузера и установленной операционной системы.

Вполне очевидно, что содержимое cookie-файлов будет варьироваться от веб-сайта к веб-сайту, но имейте в виду, что в конечном итоге они все равно являются текстовыми файлами. Таким образом, cookie-наборы — наихудший выбор для хранения конфиденциальной информации о текущем пользователе (такой как номер кредитной карты, пароль и т.п.). Даже если вы зашифруете данные, то все равно есть шанс, что настойчивый хакер расшифрует их и воспользуется в злоумышленных целях. Но, так или иначе, cookie-наборы играют определенную роль в разработке веб-приложений, поэтому давайте выясним, как ASP.NET поддерживает данный прием управления состоянием.

Создание cookie-наборов

Прежде всего, cookie-наборы ASP.NET могут быть сконфигурированы как постоянные или временные. *Постоянный* cookie-набор обычно рассматривается как классическое определение cookie-данных в том, что набор пар "имя-значение" физически сохраняется на жестком диске пользователя. *Временный* cookie-набор (также называемый *сеансовым* cookie-набором) содержит те же данные, что и постоянный cookie-набор, но пары

"имя-значение" никогда не сохраняются на жестком диске пользователя, а существуют *только* в течение периода, пока открыт браузер. Как только пользователь закрывает браузер, все данные, содержащиеся в сеансовом cookie-наборе, уничтожаются.

Класс `System.Web.HttpCookie` представляет серверную сторону cookie-данных (постоянных или временных). Для создания нового cookie-набора в коде веб-страницы необходимо обратиться к свойству `Response.Cookies`. Как только новый элемент `HttpCookie` вставлен во внутреннюю коллекцию, пары "имя-значение" отправляются браузеру внутри заголовка HTTP.

Чтобы наглядно увидеть поведение cookie-наборов, создадим новый проект пустого веб-сайта по имени `CookieStateApp` и добавим в него веб-форму с пользовательским интерфейсом, показанным на рис. Д.7.

Рис. Д.7. Пользовательский интерфейс приложения `CookieStateApp`

Внутри обработчика события `Click` кнопки Write This Cookie (Записать этот cookie-набор) создадим новый объект `HttpCookie` и вставим его в коллекцию `Cookie`, доступную через свойство `HttpRequest.Cookies`. Следует иметь в виду, что данные не будут сохраняться на жестком диске пользователя, если только явно не установить срок хранения в свойстве `HttpCookie.Expires`. Таким образом, следующая реализация создаст временный cookie-набор, который будет уничтожен, когда пользователь закроет браузер:

```
protected void btnCookie_Click(object sender, EventArgs e)
{
 // Создать временный cookie-набор.
 HttpCookie theCookie = new HttpCookie(txtCookieName.Text,txtCookieValue.Text);
 Response.Cookies.Add(theCookie);
}
```

Однако приведенный ниже код сгенерирует постоянный cookie-набор, который будет действителен в течение трех месяцев, начиная с текущей даты:

```
protected void btnCookie_Click(object sender, EventArgs e)
{
 HttpCookie theCookie = new HttpCookie(txtCookieName.Text,txtCookieValue.Text);
 theCookie.Expires = DateTime.Now.AddMonths(3);
 Response.Cookies.Add(theCookie);
}
```

Чтение входящих cookie-данных

Вспомните, что браузер является той сущностью, которая отвечает за доступ к постоянным cookie-наборам при переходе на ранее посещенную страницу. Если браузер решает отправить cookie-набор серверу, тогда доступ к входящим cookie-данным в коде страницы `.aspx` можно получить через свойство `HttpRequest.Cookies`. В целях иллюстрации реализуем обработчик события `Click` для кнопки Show Cookie Data (Показать cookie-данные):

```
protected void btnShowCookie_Click(object sender, EventArgs e)
{
  string cookieData = "";
  foreach (string s in Request.Cookies)
  {
    cookieData +=
     $"<li><b>Name</b>: {s}, <b>Value</b>: { Request.Cookies[s]?.Value }</li>";
  }
  lblCookieData.Text = cookieData;
}
```

Запустив приложение и щелкнув на кнопке Show Cookie Data, вы увидите, что cookie-данные действительно были оправлены браузером и успешно доступны в коде `.aspx` на стороне сервера (рис. Д.8).

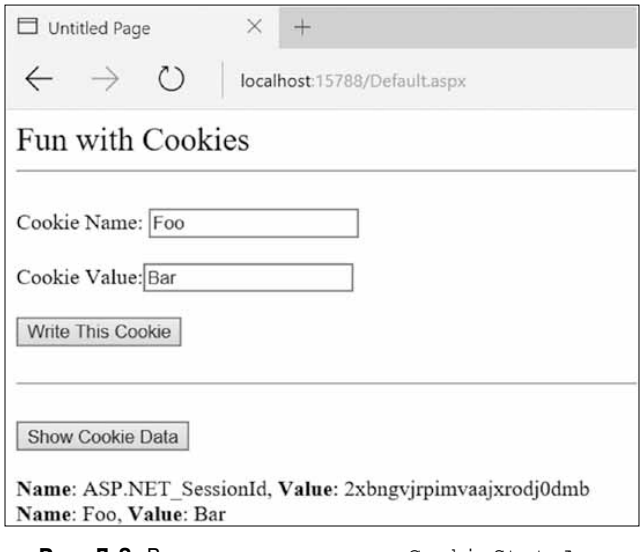

Рис. Д.8. Выполнение приложения `CookieStateApp`

Исходный код. Веб-сайт `CookieStateApp` доступен в подкаталоге `Appendix_E`.

Роль элемента <sessionState>

К настоящему моменту вы ознакомились с многочисленными способами запоминания информации о пользователях. Вы видели, что манипулирование состоянием представления и данными приложения, кеша, сеанса и cookie-набора в коде осуществляется более или менее сходным образом (посредством индексатора класса). Вы также узнали, что в файле `Global.asax` имеются методы, которые позволяют перехватывать и реагировать на события, возникающие на протяжении времени жизни веб-приложения.

По умолчанию ASP.NET будет хранить состояние сеанса внутри процесса. Положительной стороной является максимально быстрый доступ к информации. Тем не менее, отрицательная сторона заключается в том, что если произойдет аварийный отказ этого домена приложения (по любой причине), то все данные состояния пользователя разрушатся. Более того, когда данные состояния хранятся во внутрипроцессной сборке `.dll`, нет возможности взаимодействовать с сетевой веб-фермой. Такой стандартный режим хранения работает достаточно хорошо, если веб-приложение размещено на единственном веб-сервере. Однако, как и можно было предположить, эта модель не идеальна для фермы веб-серверов, т.к. состояние сеанса "замкнуто" внутри определенного домена приложения.

Хранение данных сеанса на сервере состояния сеансов ASP.NET

В ASP.NET исполняющую среду можно инструктировать о необходимости размещения сборки `.dll` состояния сеанса в суррогатном процессе, который называется сервером состояния сеансов ASP.NET (`aspnet_state.exe`). В таком случае сборку `.dll` можно выгрузить из `aspnet_wp.exe` в отдельный файл `.exe`, который может располагаться на любой машине внутри веб-фермы. Даже если вы намерены запускать процесс `aspnet_wp.exe` на той же машине, что и веб-сервер, вы все равно получите выигрыш от вынесения данных состояния в отдельный процесс (т.к. это более надежно).

Чтобы задействовать сервер состояния сеансов, первым делом запустим Windows-службу `aspnet_state.exe` на целевой машине посредством ввода следующей команды в командной строке разработчика Visual Studio (для этого понадобятся права администратора):

```
net start aspnet_state
```

В качестве альтернативы Windows-службу `aspnet_state.exe` можно запустить через значок Services (Службы) в группе Administrative Tools (Администрирование) панели управления (рис. Д.9).

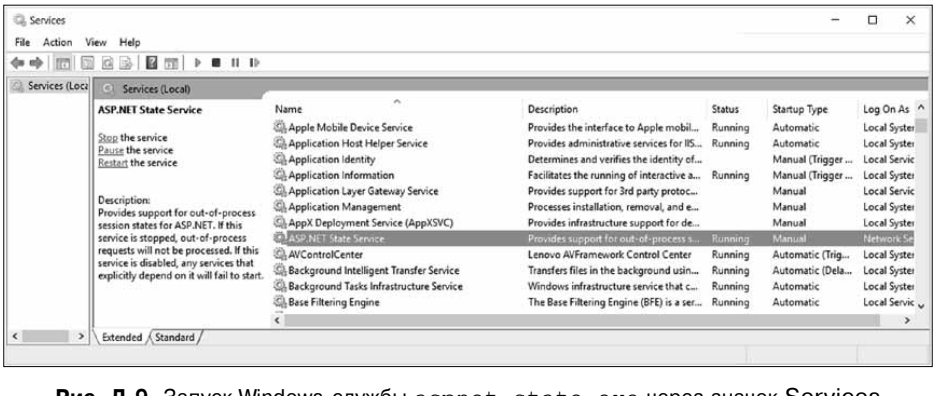

Рис. Д.9. Запуск Windows-службы `aspnet_state.exe` через значок Services

Основное преимущество такого подхода связано с возможностью конфигурирования службы aspnet_state.exe в окне ее свойств на автоматический запуск при начальной загрузке операционной системы на машине. После того, как сервер состояния сеансов запущен, добавим в файл Web.config следующий элемент <sessionState>:

```
<system.web>
  <sessionState
    mode="StateServer"
    stateConnectionString="tcpip=127.0.0.1:42626"
    sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
    cookieless="false"
    timeout="20"
  />
  ...
</system.web>
```

Вот и все! Начиная с этого момента, среда CLR будет хранить данные, относящиеся к сеансу, внутри процесса aspnet_state.exe. Если произойдет аварийный отказ домена приложения, в котором размещено веб-приложение, то данные сеанса останутся в целости. Вдобавок обратите внимание, что элемент <sessionState> может также поддерживать атрибут stateConnectionString. Стандартное значение адреса TCP/IP (127.0.0.1) указывает на локальную машину. Если вы хотите, чтобы исполняющая среда .NET использовала службу aspnet_state.exe, функционирующую на другой машине (т.е. случай веб-фермы), то можете соответствующим образом изменить адрес TCP/IP.

Хранение информации о сеансах в выделенной базе данных

Наконец, если для веб-приложения требуется самая высокая степень изоляции и надежности, то можно заставить исполняющую среду хранить все данные о состоянии сеанса внутри Microsoft SQL Server. Необходимое изменение файла Web.config выглядит очень просто:

```
<sessionState
  mode="SQLServer"
  stateConnectionString="tcpip=127.0.0.1:42626"
  sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
  cookieless="false"
  timeout="20"
/>
```

Тем не менее, прежде чем попробовать запустить ассоциированное веб-приложение, следует удостовериться, что целевая машина (указанная в атрибуте sqlConnectionString) подходящим образом сконфигурирована. При установке .NET Framework 4.6 SDK (или Visual Studio соответствующей версии) предоставляются два файла с именами InstallSqlState.sql и UninstallSqlState.sql, по умолчанию находящиеся в каталоге C:\Windows\Microsoft.NET\Framework*<версия>*. На целевой машине потребуется запустить файл InstallSqlState.sql с применением инструмента наподобие Microsoft SQL Server Management Studio (который поставляется вместе с Microsoft SQL Server).

После выполнения SQL-сценария InstallSqlState.sql вы обнаружите новую базу данных SQL Server (ASPState), которая содержит несколько хранимых процедур, вызываемых исполняющей средой ASP.NET, и набор таблиц, используемых для хранения данных сеансов. (Кроме того, база данных tempdb обновляется набором таблиц, предназначенных для подкачки.) Как и можно было предположить, конфигурация веб-приложения с хранением сеансовых данных в SQL Server является самой медленной из всех

возможных вариантов. Преимущество связано с тем, что пользовательские данные хранятся надежнее всего (они не утрачиваются, даже когда происходит перезагрузка веб-сервера).

На заметку! Если для хранения сеансовых данных применяется сервер состояния сеансов ASP.NET или SQL Server, то вы должны удостовериться, что все специальные типы, помещенные в объект `HttpSessionState`, помечены атрибутом `[Serializable]`.

Введение в API-интерфейс ASP.NET Profile

До сих пор вы исследовали различные приемы, которые позволяли запоминать данные уровня пользователей и уровня приложения. Однако многие веб-сайты требуют возможности сохранения пользовательской информации между сеансами. Например, предположим, что нужно предоставить пользователям функциональность создания учетных записей на сайте. Или, может быть, необходимо сохранять экземпляры `ShoppingCart` между сеансами (для сайта онлайнового магазина). Либо интересует, скажем, сохранение базовых пользовательских предпочтений (темы и т.д.).

Хотя для хранения такой информации можно было бы построить специальную базу данных (с несколькими хранимыми процедурами), впоследствии пришлось бы создавать специальную библиотеку кода для взаимодействия с объектами базы данных. Задача не обязательно будет сложной, но суть в том, что именно *вы* отвечаете за построение инфраструктуры подобного рода.

Чтобы помочь справиться с такими ситуациями, в состав ASP.NET входит готовый API-интерфейс управления профилями пользователей (ASP.NET Profile API) и система базы данных, предназначенная для указанной конкретной цели. В дополнение к обеспечению необходимой инфраструктуры API-интерфейс Profile позволяет определять подлежащие сохранению данные прямо внутри файла `Web.config` (для упрощения); тем не менее, можно также сохранять любой тип, помеченный атрибутом `[Serializable]`. Перед тем как погрузиться в эту тему, давайте посмотрим, где API-интерфейс Profile будет хранить указанные данные.

База данных `ASPNETDB.mdf`

Каждый веб-сайт ASP.NET, построенный с помощью Visual Studio, поддерживает папку `App_Data`. По умолчанию API-интерфейс Profile (а также другие службы вроде API-интерфейса членства в ролях ASP.NET, который здесь не рассматривается) конфигурируется на работу с локальной базой данных SQL Server по имени `ASPNETDB.mdf`, расположенной в папке `App_Data`. Такое стандартное поведение определяется настройками в файле `machine.config` для текущей установки платформы .NET на машине. В действительности, когда код задействует службу ASP.NET, требующую наличия папки `App_Data`, файл `ASPNETDB.mdf` будет автоматически создан, если он еще не существует.

Если взамен нужно, чтобы исполняющая среда ASP.NET взаимодействовала с файлом `ASPNETDB.mdf`, находящимся на другой машине в сети, или базу данных `ASPNETDB.mdf` предпочтительнее установить на экземпляре SQL Server 7.0 (либо последующей версии), тогда файл `ASPNETDB.mdf` придется создать вручную с использованием утилиты командной строки `aspnet_regsql.exe`. Подобно любой хорошей утилите командной строки `aspnet_regsql.exe` поддерживает многочисленные аргументы, но если запустить ее без аргументов (в окне командной строки разработчика):

```
aspnet_regsql
```

то откроется мастер с графическим пользовательским интерфейсом, который проведет по всему процессу создания и установки базы данных ASPNETDB.mdf на выбранной машине (и версии SQL Server).

Предполагая, что сайт не работает с локальной копией базы данных в папке App_Data, финальный шаг заключается в модификации файла Web.config с целью указания уникального местоположения файла ASPNETDB.mdf. Пусть файл ASPNETDB.mdf установлен на машине по имени ProductionServer. Приведенное ниже неполное содержимое файла machine.config будет инструктировать API-интерфейс Profile о том, где по умолчанию расположены необходимые элементы базы данных (для изменения стандартных настроек можно добавить специальный файл Web.config):

```
<configuration>
  <connectionStrings>
    <add name="LocalSqlServer"
        connectionString ="Data Source=ProductionServer;Integrated
        Security=SSPI;Initial Catalog=aspnetdb;"
        providerName="System.Data.SqlClient"/>
  </connectionStrings>
  <system.web>
    <profile>
      <providers>
       <clear/>
       <add name="AspNetSqlProfileProvider"
           connectionStringName="LocalSqlServer"
           applicationName="/"
           type="System.Web.Profile.SqlProfileProvider, System.Web,
           Version=4.0.0.0,
           Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
      </providers>
    </profile>
  </system.web>
</configuration>
```

Подобно большинству файлов .config содержимое выглядит гораздо сложнее, чем есть на самом деле. По существу здесь определяется элемент <connectionString> с необходимыми данными, за которым следует именованный экземпляр SqlProfileProvider (стандартный поставщик, применяемый независимо от физического местоположения ASPNETDB.mdf).

На заметку! Для простоты предположим, что будет использоваться автоматически сгенерированная база данных ASPNETDB.mdf, расположенная в подкаталоге App_Data веб-приложения.

Определение пользовательского профиля в файле Web.config

Как уже упоминалось, пользовательский профиль определяется внутри файла Web.config. По-настоящему замечательный аспект этого подхода заключается в том, что с профилем можно взаимодействовать в строго типизированной манере, применяя в файлах кода унаследованное свойство Property. В качестве примера создадим новый проект пустого веб-сайта по имени FunWithProfiles, добавим в него новый файл .aspx и откроем файл Web.config для редактирования.

Нашей целью будет создание профиля, который моделирует домашние адреса пользователей, находящихся в сеансе, а также общее количество их посещений веб-сайта. Данные профиля вполне предсказуемо определяются внутри элемента <profile> с ис-

пользованием набора пар “имя-значение”. Взгляните на следующий профиль, который создан внутри элемента <system.web>:

```
<profile>
  <properties>
   <add name="StreetAddress" type="System.String" />
   <add name="City" type="System.String" />
   <add name="State" type="System.String" />
   <add name="TotalPost" type="System.Int32" />
  </properties>
</profile>
```

Здесь для каждого элемента профиля указано имя и тип данных CLR (разумеется, можно было бы добавить дополнительные элементы для почтового кода, имени и т.д., но и без них идея должна быть ясна). Строго говоря, атрибут type не является обязательным и по умолчанию для него принимается тип System.String. Существует множество других атрибутов, которые могут быть указаны в элементе профиля для дальнейшего уточнения способа хранения данной информации в ASPNETDB.mdf. В табл. Д.4 кратко описаны избранные атрибуты.

Таблица Д.4. Избранные атрибуты данных профиля

Атрибут	Примеры значений	Описание
allowAnonymous	true | false	Ограничивает или разрешает анонимный доступ к данному значению. Если установлен в false, то анонимные пользователи не будут иметь доступ к этому значению профиля
defaultValue	Строка	Значение, которое должно быть возвращено, если свойство не было явно установлено
name	Строка	Уникальный идентификатор для данного свойства
provider	Строка	Поставщик, применяемый для управления этим значением. Переопределяет настройку defaultProvider в файле Web.config или machine.config
readOnly	true | false	Ограничивает или разрешает доступ для записи. Стандартным значением является false (т.е. допускается не только чтение)
serializeAs	String | XML | Binary	Формат значения при записи в хранилище данных
type	Элементарный тип | Тип, определяемый пользователем	Элементарный тип или класс .NET. Имена классов должны быть полностью заданными (например, MyApp.UserData.ColorPrefs)

Вы увидите некоторые из этих атрибутов в действии при модификации текущего профиля. А пока давайте посмотрим, как программно обращаться к данным профиля в коде страниц.

Доступ к данным профиля в коде

Вспомните, что общее предназначение API-интерфейса ASP.NET Profile заключается в автоматизации процесса записи данных в выделенную базу данных (и чтения данных из нее). Чтобы попрактиковаться, изменим пользовательский интерфейс стра-

ницы `Default.aspx`, добавив набор элементов управления `TextBox` (и описательных элементов `Label`) для ввода адреса пользователя — улицы, города и штата. Кроме того, добавим элемент управления `Button` (по имени `btnSubmit`) и еще один элемент `Label` (с именем `lblUserData`) для отображения сохраненных данных (рис. Д.10).

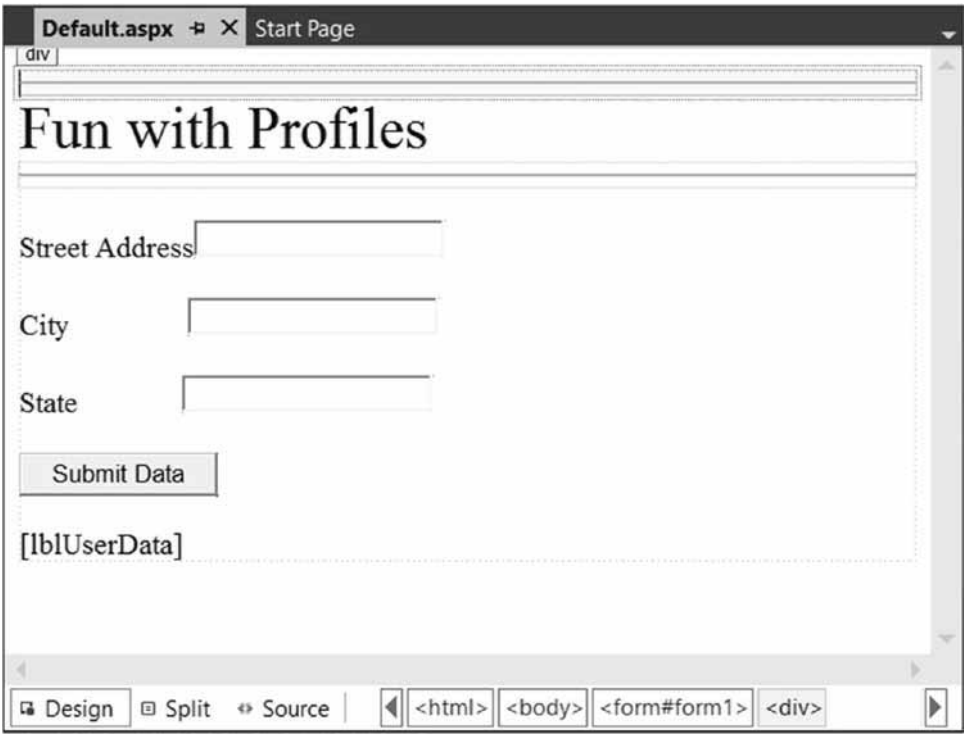

Рис. Д.10. Пользовательский интерфейс страницы `Default.aspx` веб-сайта `FunWithProfiles`

Внутри обработчика события `Click` кнопки Submit Data (Отправить данные) посредством унаследованного свойства `Profile` сохраним каждую единицу данных профиля на основе информации, введенной пользователем в соответствующем поле `TextBox`. После сохранения всех элементов данных в `ASPNETDB.mdf` прочитаем их из базы данных и сформируем строку для отображения в элементе `Label` по имени `lblUserData`. Наконец, обработаем событие `Load` страницы и отобразим ту же информацию в элементе `Label`. Таким образом, когда пользователь зайдет на страницу, он увидит текущие настройки. Ниже приведен полный код:

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    GetUserAddress();
  }
  protected void btnSubmit_Click(object sender, EventArgs e)
  {
    // Здесь происходит запись в базу данных.
    Profile.StreetAddress = txtStreetAddress.Text;
```

```
    Profile.City = txtCity.Text;
    Profile.State = txtState.Text;
    // Получить настройки из базы данных.
    GetUserAddress();
  }
  private void GetUserAddress()
  {
    // Здесь происходит чтение из базы данных.
    lblUserData.Text =
      $"You live here: {Profile. StreetAddress }, {Profile.City}, {Profile.State}";
  }
}
```

Запустив приложение, можно заметить длительную задержку при первом запросе `Default.aspx`. Причина связана с созданием на лету файла `ASPNETDB.mdf` в папке `App_Data` (в чем можно убедиться, обновив окно Solution Explorer и заглянув в папку `App_Data`).

Также обнаружится, что при первом посещении страницы элемент `Label` по имени `lblUserData` не отображает никаких данных профиля, потому что они еще не были добавлены в подходящую таблицу базы данных `ASPNETDB.mdf`. После ввода значений в полях `TextBox` и обратной отправки страницы серверу в элементе `lblUserData` отобразятся сохраненные данные.

Теперь перейдем к действительно интересному аспекту этой технологии. Если закрыть браузер и перезапустить веб-сайт, то выяснится, что ранее введенные данные профиля на самом деле сохранились, т.к. в `Label` отображается корректная информация. Возникает очевидный вопрос: как они были сохранены?

В рассматриваемом примере API-интерфейс Profile использует сетевую идентичность Windows, которая получена из текущих учетных данных машины. Однако при построении публично доступных веб-сайтов (где пользователи не относятся к какому-то домену) понадобится обеспечить интеграцию API-интерфейса Profile с моделью аутентификации ASP.NET на основе форм, а также поддержку понятия “анонимных профилей”, которые позволяют хранить данные профиля для пользователей, не имеющих в данный момент активной идентичности на сайте.

На заметку! Темы, связанные с безопасностью ASP.NET (такие как аутентификация на основе форм или анонимные профили), в книге не рассматриваются. Подробные сведения ищите в документации .NET Framework 4.6 SDK.

Группирование данных профиля и сохранение специальных объектов

В завершение давайте посмотрим, каким образом данные профиля могут определяться в файле `Web.config`. В текущем профиле просто определены четыре порции данных, которые доступны напрямую через тип `Profile`. При построении более сложных профилей может быть удобно группировать связанные данные под уникальным именем. Взгляните на следующее изменение:

```
<profile>
  <properties>
    <group name ="Address">
      <add name="StreetAddress" type="String" />
      <add name="City" type="String" />
```

```
      <add name="State" type="String" />
    </group>
    <add name="TotalPost" type="Integer" />
  </properties>
</profile>
```

На этот раз определена специальная группа по имени `Address`, включающая название улицы, город и штат пользователя. Для доступа к таким данным внутри страниц потребуется модифицировать код, указав `Profile.Address` для получения каждого подэлемента. Например, вот обновленный метод `GetUserAddress()` (обработчик события `Click` для кнопки Submit Data нуждается в похожих изменениях):

```
private void GetUserAddress()
{
  // Здесь происходит чтение из базы данных.
  lblUserData.Text =
    $"You live here: {Profile.Address.StreetAddress},
     {Profile.Address.City}, " +
    $"{Profile.Address.State}";
}
```

Перед запуском приложения необходимо удалить файл `ASPNETDB.mdf` из папки `App_Data`, чтобы обеспечить обновление схемы базы данных. Затем пример веб-сайта должен выполняться без ошибок.

На заметку! Профиль может содержать столько групп, сколько вы считаете нужным. Просто определяйте множество элементов `<group>` внутри `<properties>`.

И, наконец, полезно отметить, что профиль может также хранить в базе данных `ASPNETDB.mdf` специальные объекты (и извлекать их). В целях иллюстрации предположим, что необходимо построить специальный класс (или структуру) для представления данных адреса пользователя. Единственное требование, предъявляемое API-интерфейсом Profile к такому типу, заключается в том, что он должен быть помечен атрибутом `[Serializable]`:

```
[Serializable]
public class UserAddress
{
  public string Street = string.Empty;
  public string City = string.Empty;
  public string State = string.Empty;
}
```

При наличии такого класса определение профиля может быть модифицировано следующим образом (специальная группа удалена, хотя это вовсе не обязательно):

```
<profile>
  <properties>
    <add name="AddressInfo" type="UserAddress" serializeAs ="Binary"/>
    <add name="TotalPost" type="Integer" />
  </properties>
</profile>
```

Обратите внимание, что в случае добавления к профилю типов `[Serializable]` в атрибуте `type` указывается полностью заданное имя сохраняемого типа. Как вы увидите в окне IntelliSense среды Visual Studio, основными вариантами являются двоичные, XML и строковые данные. Теперь, когда информация адреса организована в виде специального класса, понадобится модифицировать кодовую базу:

```
private void GetUserAddress()
{
  // Здесь происходит чтение из базы данных.
  lblUserData.Text =
   $"You live here: {Profile.AddressInfo.Street}, {Profile.AddressInfo.City}, " +
   $"{Profile.AddressInfo.State}";
}
```

Конечно, с API-интерфейсом Profile связаны многие другие аспекты кроме тех, которые здесь обсуждались. Например, свойство `Profile` в действительности инкапсулирует тип `ProfileCommon`. С помощью этого типа можно программно получать всю информацию о конкретном пользователе, удалять (или добавлять) профили в `ASPNETDB.mdf`, обновлять данные профиля и т.д.

Более того, API-интерфейс Profile имеет многочисленные точки расширения, которые позволяют оптимизировать способ доступа диспетчера профилей к таблицам базы данных `ASPNETDB.mdf`. Как и можно было ожидать, существует много способов сокращения количества обращений к базе данных. Заинтересованным читателям рекомендуем обратиться за подробностями в документацию .NET Framework 4.6 SDK.

Исходный код. Веб-сайт `FunWithProfiles` доступен в подкаталоге `Appendix_E`.

Резюме

Настоящее приложение дополнило имеющиеся у вас знания ASP.NET сведениями о применении типа `HttpApplication`. Вы видели, что тип `HttpApplication` предлагает несколько стандартных обработчиков событий, которые позволяют перехватывать разнообразные события уровня приложения и уровня сеанса. Большая часть приложения была посвящена рассмотрению различных приемов управления состоянием. Вспомните, что состояние представления используется для автоматического заполнения значениями виджетов HTML между обратными отправками определенной страницы. Кроме того, вы узнали о разнице между данными уровня приложения и данными уровня сеанса, об управлении cookie-наборами и о кеше приложения ASP.NET.

Наконец, вы ознакомились с интерфейсом ASP.NET Profile API. Как было показано, эта технология предлагает готовое решение задачи сохранения пользовательских данных между сеансами. С применением конфигурационного файла `Web.config` веб-сайта можно определять любое количество элементов профиля (включая группы элементов и типы `[Serializable]`), которые будут автоматически сохраняться в базе данных `ASPNETDB.mdf`.