Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

ЯЗЫК C# И ПЛАТФОРМА .NET
(часть 7)

Курс лекций
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Минск 2009

СОДЕРЖАНИЕ

7. ASP.NET MVC

7.1. ASP.NET MVC – БАЗОВЫЕ КОНЦЕПЦИИ

Модель-представление-контроллер (*Model-View-Controller, MVC*) – архитектурный шаблон, состоящий из трёх компонентов (рис. 1):

1. *Модель* представляет данные, с которыми работает приложение. Модель реализовывает логику обработки данных согласно заданным бизнес-правилам и обеспечивает чтение и сохранение данных во внешних хранилищах.
2. *Представление* обеспечивает способ отображения модели.
3. *Контроллер* обрабатывает внешние запросы и координирует изменение модели и актуальность представления.

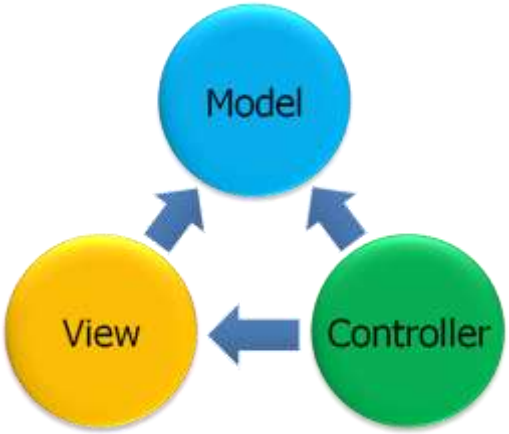

Рис. 1. Модель-представление-контроллер.

Как представление, так и контроллер зависят от модели. Однако модель не зависит ни от представления, ни от контроллера. Это одно из ключевых достоинств MVC. Оно позволяет строить модель независимо от визуального представления, а также создавать несколько различных представлений и контроллеров для одной модели.

Классический подход к разработке веб-приложений на основе ASP.NET, известный как *ASP.NET Web Forms*, стремится перенести принципы создания обычных оконных приложений в веб-среду. ASP.NET Web Forms предлагает использовать готовые элементы управления и событийную модель программирования с сохранением состояния страницы. Однако такой подход обладает рядом недостатков:

- Сохранение состояния страницы основано на механизме ViewState, что ведёт к передаче больших объёмов данных между клиентом и сервером.
- Понимание этапов жизненного цикла страницы и связанных с ним событий не является тривиальным.
- Разработчик имеет ограниченный контроль над HTML-кодом, так как стандартные элементы управления выполняют свой рендеренг в HTML предопределённым образом.

- Адреса и ссылки, генерируемые ASP.NET, затрудняют индексирование и поисковую оптимизацию сайта.
- Событийная модель провоцирует на использование *антипаттерна Smart UI*, при котором бизнес-правила помещаются не в слой бизнес-логики, а в обработчики событий.
- Модульное тестирование приложений ASP.NET Web Forms затруднено.

ASP.NET MVC – это каркас для разработки веб-приложений, созданный как альтернатива ASP.NET Web Forms. Как следует из названия, ASP.NET MVC базируется на ASP.NET, но предполагает использование шаблона MVC. Ниже перечислены основные отличительные черты ASP.NET MVC.

1. *Открытость*. Исходные коды ASP.NET MVC доступны для анализа.
2. Четкое *разделение компонентов* приложения, опирающееся на использовании шаблона MVC.
3. *Расширяемость*. При работе с ASP.NET MVC практически для каждого элемента каркаса можно использовать либо реализацию по умолчанию, либо наследование или полную замену элемента для удовлетворения специфических нужд.
4. Лучшие возможности для *модульного тестирования*.
5. Полный *контроль* над генерируемой *HTML-разметкой*.
6. *Система маршрутизации* – программист полностью контролирует используемые в приложении URLs и схему их построения.
7. Использование *конвенций именования*: хотя следования данным правилам не обязательно, применение соглашений значительно уменьшает объем кода, создаваемого программистом.
8. Применение возможностей ASP.NET и платформы .NET 3.5. ASP.NET MVC является альтернативой ASP.NET Web Forms, но не отвергает саму платформу ASP.NET. Многие механизмы ASP.NET с успехом применяются в ASP.NET MVC. Возможно построение *гибридных приложений*, сочетающих ASP.NET MVC и ASP.NET Web Forms.

Первая публичная демонстрация ASP.NET MVC была проведена на конференции разработчиков в октябре 2007 года. Официальный выход ASP.NET MVC состоялся в марте 2009 года. В настоящий момент (осень 2009 года) доступна предварительная версия ASP.NET MVC 2.0. Известно, что каркас ASP.NET MVC будет включён в состав .NET Framework 4.0.

7.2. СТРУКТУРА ВЕБ-ПРИЛОЖЕНИЯ В ASP.NET MVC

Чтобы начать работу с ASP.NET MVC, следует скачать этот каркас (например, с http://www.asp.net/mvc) и установить его. На компьютере необходимо наличие .NET Framework 3.5 (желательно .NET Framework 3.5 SP1).

После установки ASP.NET MVC в Visual Studio станут доступны шаблоны проектов MVC (*File*/*New*/*Project*/*Web*/*ASP.NET MVC Web Application*). Новый проект MVC обладает определенной структурой (рис. 2).

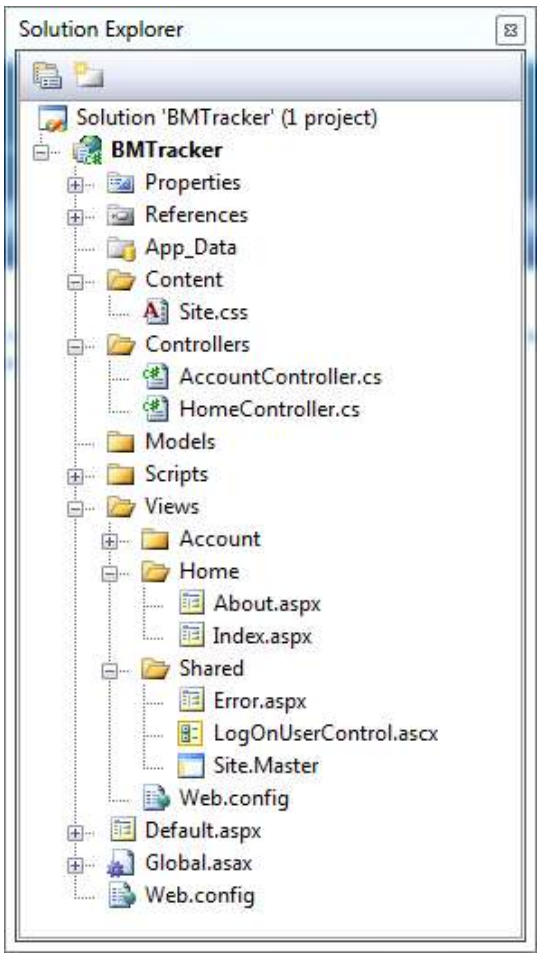

Рис. 2. Структура проекта ASP.NET MVC.

В табл. 1 перечислены основные элементы проекта MVC и их назначение.

Таблица 1

Основные элементы проекта ASP.NET MVC

Имя папки или файла	**Предполагаемое назначение**	**Примечания**
`/App_Data`	Стандартная папка ASP.NET для размещения файлов с данными приложения (`*.xml`, `*.mdf`, `*.mdb`)	
`/bin`	Стандартная папка ASP.NET для скомпилированных сборок	Сборки, используемые в приложении, могут размещаться в GAC
`/Content`	Папка для статических файлов, связанных с сайтом (изображения, файлы CSS)	Статический контент может располагаться в любом каталоге
`/Controllers`	Папка, используемая по умолчанию для MVC- контроллеров	
`/Models`	Здесь могут размещаться классы, представляющие модель приложения	Часто для описания модели создаётся отдельный проект
`/Scripts`	Папка для клиентских файлов JavaScript	По умолчанию содержит файлы для поддержки ASP.NET AJAX и файлы библиотеки jQuery

`/Views`	Здесь содержатся представления (файлы `*.aspx`) и частичные представления (файлы `*.ascx`)	По конвенции именования, представления для контроллера `XyzController` помещаются в папку `/Views/Xyz/`. Представление для метода `DoAction()` должно называться `DoAction.aspx`
`/Views/Shared`	Здесь содержатся шаблоны представлений (обычно это эталонные страницы `*.master`) или общие представления	Если MVC не находит представление `/Views/Xyz/DoAction.aspx` (`*.ascx`), то производится поиск `/Views/Shared/DoAction.aspx`
`/Views/web.config`	Конфигурационный файл, запрещающий прямое обращение к файлам представлений	
`/default.aspx`	Этот файл не используется в ASP.NET MVC, но нужен для совместимости с IIS 6	
`/global.asax`	В этом фале происходит регистрация маршрутов вашего MVC-приложения	
`/web.config`	Конфигурационный файл, по умолчанию содержит настройки инфраструктуры ASP.NET MVC	

Любой новый MVC-проект готов к запуску. Он содержит две простые страницы, два контролера, визуальный стиль, и может быть использован как основа для собственных разработок (рис. 3).

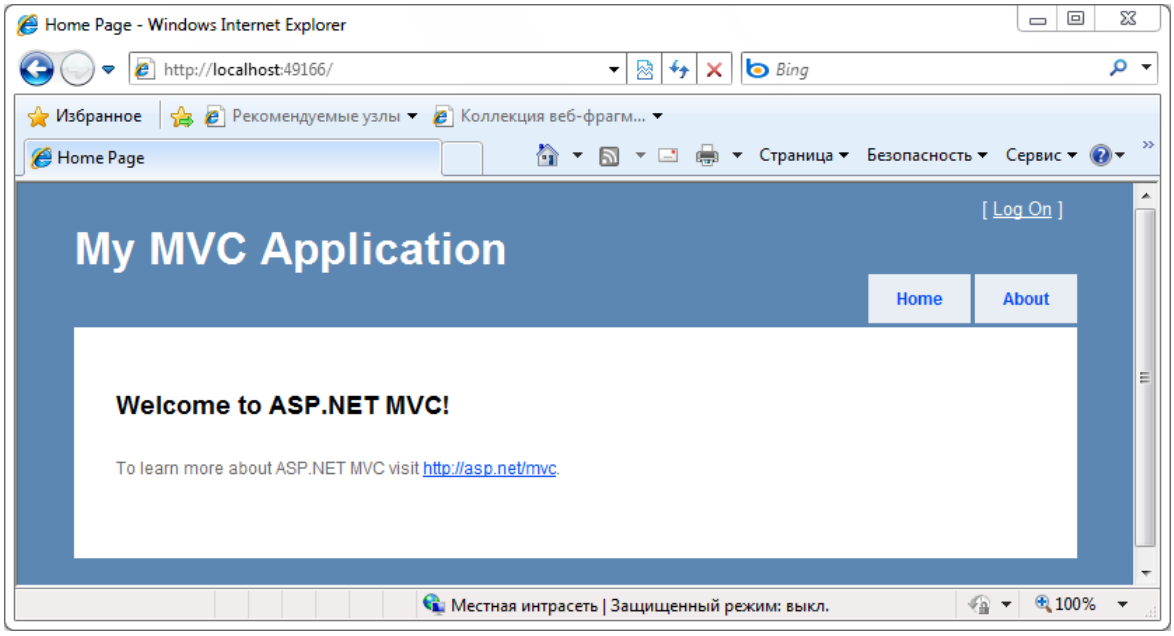

Рис. 3. Новое приложение ASP.NET MVC в браузере.

7.3. МАРШРУТИЗАЦИЯ

URL – это уникальный адрес ресурса в веб-среде. В URL можно выделить *базовую часть* (имя протокола, адрес сайта и, возможно, порт доступа), *путь к ресурсу* на сайте и *параметры ресурса*. Только базовая часть URL является обязательной.

В .NET Framework 3.5 SP1 введён механизм *маршрутизации URL*. В веб-приложении можно организовать *таблицу маршрутов*, где отдельный *маршрут* связывает шаблон URL с неким *обработчиком запроса*. *Шаблон URL* – это строка, описывающая внебазовую часть URL и содержащая *параметры*, записанные в фигурных скобках. При поступлении клиентского запроса *модуль маршрутизации* ищет в таблице первый подходящий маршрут. Затем у найденного маршрута вызывается обработчик запроса.

Для описания маршрута используются классы `RouteBase` и `Route` из пространства имён `System.Web.Routing`. `RouteBase` - абстрактный класс; его наследник `Route` поддерживает работу с шаблонами URL. Основные свойства класса `Route`:

- `Url` – строка, представляющая шаблон URL;
- `RouteHandler` – обработчик запроса для маршрута. Это объект, который должен реализовывать интерфейс `IRouteHandler`;
- `Defaults` – коллекция значений по умолчанию для параметров маршрута;
- `Constraints` – коллекция ограничений на параметры маршрута;
- `DataTokens` – набор значений, передаваемых обработчику маршрута, но не являющихся параметрами шаблона URL.

Свойства `Defaults`, `Constraints` и `DataTokens` имеют тип `RouteValueDictionary`. Это обычный словарь «строка-объект». Особенностью класса `RouteValueDictionary` является конструктор, принимающий в качестве параметра произвольный объект (возможно, анонимного типа) и создающий словарь на основе имён свойств этого объекта.

Рассмотрим пример кода, создающего объект класса `Route`. Конструктор `Route` имеет несколько перегруженных версий, однако любая из них требует указания значений для `Url` и `RouteHandler`. В примере кода в качестве обработчика маршрута указан класс `MvcRouteHandler`, что стандартно для ASP.NET MVC. Если используется обработчик `MvcRouteHandler`, маршрут должен содержать параметры `controller` и `action`. Обратите внимание на использование анонимного типа для задания свойства `Defaults`.

```
var r = new Route("{controller}/{action}", new MvcRouteHandler());
r.Defaults = new RouteValueDictionary(
                    new {controller = "Products", action = "List"});
```

Разберем некоторые нюансы работы со свойством `Defaults`. Если параметр присутствует в шаблоне URL, но не указан в словаре `Defaults`, то этот параметр рассматривается как обязательная часть URL. Чтобы сделать параметр опциональным, нужно указать его в наборе `Defaults` со значением `null`. Также

парамерами URL становятся элементы `Defaults`, которые не указаны в самом шаблоне URL.

```
// color должен присутствовать в URL
var r1 = new Route("catalog/{color}", new MvcRouteHandler())
         {
             Defaults = new RouteValueDictionary(
                     new {controller = "Products", action = "List"})
         };

// color может отсутствовать в URL
var r2 = new Route("catalog/{color}", new MvcRouteHandler())
         {
             Defaults = new RouteValueDictionary(
                 new {controller = "Products", action = "List",
                      color = (string) null})
         };
```

Свойство маршрута `Constraints` позволяет наложить на параметры шаблона URL определённые ограничения. Ограничением является или строка, представляющей регулярное выражение, или объект, реализующий интерфейс `IRouteConstraint`.

```
var r1 = new Route("Articles/{id}", new MvcRouteHandler());
r1.Constraints = new RouteValueDictionary(new {id = @"\d{1,6}"});

var r2 = new Route("Articles/{id}", new MvcRouteHandler());
r2.Constraints = new RouteValueDictionary(
                 new {httpMethod = new HttpMethodConstraint("GET")});
```

Класс `HttpMethodConstraint`, который использован в примере кода, - это стандартный класс ASP.NET MVC для фильтрации методов HTTP. Для такого ограничения не важно, какой у него ключ в словаре `Constraints`.

Таблица маршрутизации приложения хранится в статической коллекции `RouteTable.Routes`. Обычно эта коллекция заполняется в обработчике события `Application_Start()` в файле `global.asax`. В ASP.NET MVC файл `global.asax` выглядит следующим образом (удалены директивы `using` и пространство имён):

```
public class MvcApplication : HttpApplication
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapRoute(
             "Default", // Имя маршрута (можно указать null)
             "{controller}/{action}/{id}", // URL с параметрами
              new {controller = "Home", action = "Index", id = ""});
    }
```

```
        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
```

`MapRoute()` и `IgnoreRoute()` – это методы расширения для работы с `RouteTable.Routes`, описанные в ASP.NET MVC. Вызов `MapRoute()` влечёт вызов конструктора класса `Route`, где в качестве обработчика маршрута указан объект класса `MvcRouteHandler`. Маршрут, по умолчанию используемый в ASP.NET MVC, состоит из имени контроллера, имени метода контроллера и параметра метода. Для этих параметров заданы некоторые значения по умолчанию. Вызов `IgnoreRoute()` используется для игнорирования маршрута. Технически, игнорируемый маршрут обрабатывается классом `StopRoutingHandler`.

7.4. КОНТРОЛЛЕРЫ И ДЕЙСТВИЯ

В шаблоне MVC контроллеры отвечают за *логику приложения*, которая включает получение пользовательского ввода, выполнение команд над моделью и управление представлениями. В объектно-ориентированных языках контроллер обычно представлен как класс, методы которого называют *действиями контроллера* (*actions*).

В ASP.NET MVC класс контроллера должен реализовывать интерфейс `IController`. Это единственное строгое требование к контроллеру. Ниже дано описание интерфейса `IController`, а также пример примитивного контроллера.

```
public interface IController
{
    void Execute(RequestContext requestContext);
}

public class HelloWorldController : IController
{
    public void Execute(RequestContext requestContext)
    {
        requestContext.HttpContext.Response.Write("Hello, world!");
    }
}
```

При практическом использовании ASP.NET MVC создание контроллера только на основе `IController` не применяется. Обычно пользовательские контроллеры наследуются от класса `System.Web.Mvc.Controller` (дополнительно применяется соглашение об именовании – имя класса-контроллера заканчивается на `Controller`). Методы пользовательского контроллера становятся действиями и доступны из веб, если выполняются все следующие условия:

- Метод объявляен как `public` и не является статическим;
- Метод не объявлен как универсальный;

- Метод не помечен атрибутом [NonAction];
- У метода нет перегруженных версий, за исключением случаев, когда эти версии помечены атрибутами [NonAction] или [AcceptVerbs].

Получение входных данных

Обычно действию контроллера требуются для выполнения некие входные данные. В ASP.NET MVC существуют три основных способа для получения контроллером такой информации.

Первый способ предполагает извлечение входных данных из различных *контекстных объектов*, доступных в контроллере при наследовании от System.Web.Mvc.Controller. В табл. 2 перечислены основные контекстные объекты.

Таблица 2

Основные контекстные объекты, доступные в контроллере

Свойство для доступа	Его тип	Описание
Request.QueryString	NameValueCollection	Переменные, переданные в GET-запросе
Request.Form	NameValueCollection	Переменные, переданные в POST-запросе
Request.Cookies	HttpCookieCollection	Cookies, посланные браузером
Request.HttpMethod	string	HTTP-метод запроса (GET, POST,…)
Request.Headers	NameValueCollection	Все HTTP-заголовки запроса
Request.Url	string	Запрашиваемый URL
RouteData.Route	RouteBase	Объект из таблицы маршрутизации, соответствующий запросу
RouteData.Values	RouteValueDictionary	Словарь параметров маршрута и их значений
HttpContext.Cache	Cache	Кеш приложения
HttpContext.Session	HttpSessionStateBase	Информация сессии пользователя
User	IPrincipal	Информация об аутентифицированном пользователе
TempData	TempDataDictionary	Данные, сохранённые при обработке предыдущего запроса

Второй способ получения входных данных – описать и использовать параметры метода-действия. Каркас ASP.NET MVC пытается самостоятельно заполнить значения параметров, используя данные контекстных объектов контроллера. Для каждого из параметров выполняется поиск значения по имени параметра без учета регистра. В случае неудачи или ошибки приведения типов, параметры ссылочного типа получают значение null, а для параметров типов-значений генерируется исключение.

Третий способ извлечения входных данных предполагает использование специального механизма ASP.NET MVC, называемого *привязка к модели* (*model binding*). Вместо атомарных входных параметров привязка к модели позволяет указать единственный параметр-объект. Заполнение атрибутов этого объекта будет выполнено автоматически, подобно тому, как это делалось для прими-

тивных параметров во втором способе. Пусть, например, имеется модельный класс `Person`:

```
public class Person
{
    public string Name { get; set; }
    public int? Age { get; set; }
}
```

Можно создать следующий метод-действие для обновления `Person`:

```
public ActionResult Update(string name, int? age)
{
    var person = new Person{ Name = name, Age = age};
    ViewData["Message"] = person.Name + " " + person.Age;
    return View();
}
```

Использование привязки к модели позволяет упростить метод `Update()`:

```
public ActionResult Update(Person person)
{
    ViewData["Message"] = person.Name + " " + person.Age;
    return View();
}
```

Привязку к модели можно запустить принудительно, вызвав метод `UpdateModel()`.

```
public ActionResult Update()
{
    var person = new Person();
    UpdateModel(person);
    ViewData["Message"] = person.Name + " " + person.Age;
    return View();
}
```

Генерирование выходных данных

При использовании контроллеров, унаследованных от `System.Web.Mvc.Controller`, действия возвращают объект для описания результата своей работы. Как правило, используется класс `ActionResult` и его наследники[1], перечисленные в табл. 3.

[1] Метод-действие может возвращать произвольный объект или быть объявленным как `void`. В первом случае на основе результата создаётся объект класса `ContentResult`, во втором – возвращается объект `EmptyResult`.

Таблица 3

Встроенные типы для представления результата действия

Имя типа	Описание
ViewResult	Возвращает представление по умолчанию для контроллера или представление с указанным именем
PartialViewResult	Возвращает частичное представление по умолчанию для контроллера или частичное представление с указанным именем
RedirectToRouteResult	Вызывает перенаправление (HTTP 302) на указанный метод-действие или маршрут
RedirectResult	Вызывает перенаправление (HTTP 302) на указанный URL
ContentResult	Возвращает браузеру обычный текст (можно указать значение заголовка content-type)
FileResult	Передаёт клиенту двоичные данные (файл, массив байт)
JsonResult	Сериализует объект в формате JSON и передаёт клиенту
JavaScriptResult	Посылает клиенту фрагмент кода на JavaScript или js-файл
HttpUnauthorizedResult	Посылает клиенту HTTP-код 401 для запуска механизма авторизации
EmptyResult	Не выполняет никаких действий

Во фрагменте кода, представленном ниже, показаны примеры использования типов для результата действия. Заметим, что для создания объектов рассматриваемых типов можно использовать как обычный конструктор, так и встроенные методы класса System.Web.Mvc.Controller.

```
public ActionResult Index()
{
    // 1а. Возвращаем представление, используя конструктор
    return new ViewResult { ViewName = "HomePage" };
    // 1б. Возвращаем представление, используем встроенный метод
    return View("HomePage");
    // 1в. Без параметров - возвращаем представление по умолчанию
    return View();

    // 2. Выполняем редирект на действие контроллера
    // (имени контроллера нет, значит действие в том же контроллере)
    return RedirectToAction("Show");

    // 3. Выполняем редирект на указанный маршрут по имени в таблице
    // маршрутизации (можно передать параметры маршрута)
    return RedirectToRoute("MyNameRoute", new { param = "value" });

    // 4. Выполняем редирект на указанный URL
    return Redirect("http://example.com");

    // 5. Возвращаем обычный текст
    return Content("This is a plain text", "text/plain");

    // 6. Сериализуем некий объект в JSON и возвращаем клиенту
    return Json(obj);
```

```
        // 7. Передаём клиенту файл
        return File("somefile.pdf", "application/pdf");
    }
```

Когда результатом действия является представление, возникает задача передачи информации этому представлению. Для этого применимы несколько вариантов. Класс `Controller` имеет коллекцию `ViewData`, представляющую собой слаботипизированный словарь. Эта коллекция доступна и в представлении:

```
// код в контроллере
ViewData["Message"] = "Welcome to ASP.NET MVC!";

<%-- код в представлении --%>
<%= Html.Encode(ViewData["Message"]) %>
```

У `ViewData` есть специальное свойство `Model` (произвольный объект). Свойство можно задать непосредственно или использовать для этого параметр метода `View()`.

```
var person = new Person{ Name = "Mister X", Age = 20};
// первый вариант
ViewData.Model = person;
// второй вариант
return View(person);
```

Ещё один вариант передачи данных представлению - *строготипизированные представления* – будет рассмотрен в дальнейшем.

При выполнении переадресации на маршрут или действие удобно использовать для хранения промежуточных данных особую коллекцию `TempData`, доступную в теле контроллера. `TempData` работает как коллекция `Session`, однако данные, которые туда помещены, сохраняются ровно на один следующий запрос.

```
TempData["myKey"] = person;
```

Использование фильтров действий

Контроллеры и действия могут быть снабжены *фильтрами*, добавляющими дополнительные шаги в процесс обработки запроса. Существуют четыре вида фильтров: фильтры авторизации, действия, результата и исключения. Фильтры авторизации позволяют ограничить доступ к контроллерам или действиям по роли или имени пользователя. Фильтры исключений указывают способ обработки исключения, возникшего при выполнении действия. Фильтры действий и результатов могут определить методы, выполняемые инфраструктурой ASP.NET MVC до или после действия и генерирования результата.

Технически, любой фильтр – это .NET атрибут. Для каждого вида фильтров имеется отдельный интерфейс и базовый класс реализации[1] (табл. 4):

Таблица 4

Интерфейсы и базовые реализации фильтров

Вид фильтр	Интерфейс	Базовый класс
Авторизация	`IAuthorizationFilter`	`AuthorizeAttribute`
Действие	`IActionFilter`	`ActionFilterAttribute`
Результат	`IResultFilter`	`ActionFilterAttribute`
Исключение	`IExceptionFilter`	`HandleErrorAttribute`

Следующий код демонстрирует пример создания простого пользовательского фильтра. В качестве базового класса выбран `ActionFilterAttribute`, реализующий интерфейсы `IActionFilter` и `IResultFilter`.

```
public class ShowMessageAttribute : ActionFilterAttribute
{
    public string Message { get; set; }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
      context.HttpContext.Response.Write("BeforeAction " + Message);
    }
    public override void OnActionExecuted(ActionExecutedContext context)
    {
      context.HttpContext.Response.Write("AfterAction " + Message);
    }
    public override void OnResultExecuting(ResultExecutingContext context)
    {
      context.HttpContext.Response.Write("BeforeResult " + Message);
    }
    public override void OnResultExecuted(ResultExecutedContext context)
    {
      context.HttpContext.Response.Write("AfterResult " + Message);
    }
}
```

Фильтры могут применяться как к отдельному действию, так и ко всему контроллеру. В этом случае считается, что они распространяются на каждое действие контроллера. Альтернативой фильтрам-атрибутам может служить переопределение виртуальных методов класса `System.Web.Mvc.Controller`: `OnActionExecuting()`, `OnActionExecuted()`, `OnResultExecuting()`, `OnResultExecuted()`, `OnAuthorization()` и `OnException()`.

Опишем некоторые встроенные фильтры ASP.NET MVC. Фильтр `[Authorize]` – это фильтр авторизации. При применении фильтра указывается список пользователей и (или) список ролей. Для выполнения действия пользо-

[1] Все классы фильтров наследуются от `FilterAttribute`. У `FilterAttribute` имеется целочисленное свойство `Order` для управления порядком применения фильтров.

ватель должен быть авторизован, указан в списке пользователей и иметь хотя бы одну из указанных ролей.

```
public class MicrosoftController : Controller
{
    [Authorize(Users = "billg, steveb", Roles = "chairman, ceo")]
    public ActionResult BuySmallCompany(string name, double price)
    { . . . }
}
```

Если при выполнении действия было сгенерировано необработанное исключение, фильтр [HandleError] позволит вывести представление с описанием ошибки. У [HandleError] можно задать такие параметры как тип исключения, имя представления и имя шаблона (master page) для представления. Если не задан тип исключения, обрабатываются все исключения. Если не указано имя представления, используется представление Error.

```
[HandleError]
public class HomeController : Controller
{ . . . }
```

Фильтр [OutputCache] позволяет кэшировать результат отдельного действия или всех действий контроллера. Свойства фильтра [OutputCache] совпадают с параметрами директивы @OutputCache, применяемой в классическом ASP.NET.

```
public class StockTradingController : Controller
{
    [OutputCache(Duration = 60)]
    public ViewResult CurrentRiskSummary()
    { . . . }
}
```

В заключение опишем несколько атрибутов, применяемых к методам контроллера, но не являющихся фильтрами. Атрибут [NonAction] экранирует public-метод так, что этот метод не распознаётся как действие. Атрибут [ActionName] даёт действию имя, с которым сопоставляются параметры URL. Атрибут [AcceptVerbs] позволяет указать HTTP-методы, для которых будет вызываться действие.

```
[AcceptVerbs(HttpVerbs.Get)]
public ActionResult DoSomething() { . . . }

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult DoSomething(int someParam) { . . . }
```

7.5. ПРЕДСТАВЛЕНИЯ

Представления отображают данные модели. Идеология MVC предполагает возможность использования различных (заменяемых) представлений для одной задачи отображения. По умолчанию в ASP.NET MVC применяется «движок» представлений Web Forms View Engine, основанный на элементах классического ASP.NET – aspx-страницах и ascx-элементах. Как отдельные проекты, для ASP.NET MVC существуют «движки» представления NVelocity, Brail, Spark, NHaml.

Основы Web Forms View Engine

В Web Forms View Engine отдельное представление является aspx-страницей[1]. Следуя конвенции именования, для контроллера `XyzController` и действия `Action()` представление должно называться `Action.aspx` и располагаться в папке `~/Views/Xyz` или в папке `~/Views/Shared`. Базовым классом для представлений является `System.Web.Mvc.ViewPage` или универсальный класс `System.Web.Mvc.ViewPage<T>`. В первом случае представление называется *нетипизированным* (или *слаботипизированным*), во втором - *строготипизированным*. Предполагается, что параметром универсального класса является один из типов модели.

```
<%@ Page Inherits="System.Web.Mvc.ViewPage" %>
This is a <i>very</i> simple view.
```

Как и любая aspx-страница, представление может иметь ссылку на эталонную страницу. В отличие от страниц классического ASP.NET, представления лишены файла Code Behind. Это связано с отказом от событийной модели и традиционных элементов управления. В ASP.NET MVC разработчик строит внешний вид страницы, используя чистый HTML и вкрапления серверного кода. Для облегчения данного процесса имеется набор методов, обеспечивающий вывод стандартных HTML-элементов[2].

Рассмотрим пример. Пусть имеется класс `Person`:

```
public class Person
{
    public string Name { get; set; }
    public int? Age { get; set; }
    public IEnumerable<Person> Children { get; set; }
}
```

Создадим строготипизированное представление для вывода информации об объекте класса `Person` (обратите внимание на вкрапления серверного кода):

---

[1] Существует понятие *частичного представления* – ascx-элемент управления.
[2] В принципе, использование классических элементов управления ASP.NET возможно.

```
<%@ Page Language="C#"
         Inherits="System.Web.Mvc.ViewPage<Person>" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title><%= Model.Name %></title>
</head>
  <body>
    <h1>Information about <%= Model.Name %></h1>
    <p><%= Model.Name %> is <%= Model.Age %> years old.</p>
    <h3>Children:</h3>
    <ul>
        <% foreach (var child in Model.Children) { %>
        <li>
            <b> <%= child.Name %></b>, age <%= child.Age %>
        </li>
        <% } %>
    </ul>
  </body>
</html>
```

Приведённый пример демонстрирует, как представление может получить данные модели. В строготипизированном представлении данные доступны через типизированное свойство `Model`. В любом представлении есть коллекция `ViewData` - слаботипизированный словарь с дополнительным свойством `Model`.

Методы для вывода HTML-элементов

В классе `System.Web.Mvc.ViewPage` определено свойство `Html`, имеющее тип `System.Web.Mvc.HtmlHelper`, и свойство `Url` типа `System.Web.Mvc.UrlHelper`[1]. При импортировании в проект пространства имён `System.Web.Mvc.Html` (это происходит по умолчанию) у класса `HtmlHelper` появляется набор методов расширения для генерирования элементов HTML.

Первую группу методов расширения составляют методы для элементов ввода – флажков, текстовых полей, паролей. В примере кода демонстрируется использование всех методов данной группы.

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>
<html>
  <body>
    <p><%= Html.CheckBox("Check Box ", true) %></p>
    <p><%= Html.Hidden("Скрытое поле", "info") %></p>
    <p><%= Html.RadioButton("Radio button", "info", true) %></p>
    <p><%= Html.Password("Пароль", "info") %></p>
    <p><%= Html.TextArea("Область ввода", "info", 5, 20, null)%></p>
    <p><%= Html.TextBox("Поле ввода", "info", true) %></p>
  </body>
</html>
```

[1] В строготипизированном представлении это типы `HtmlHelper<T>` и `UrlHelper<T>`.

Первый параметр методов из описываемой группы задаёт значение HTML-атрибутов `id` и `name`. Для вычисления атрибута `value` применяется следующий алгоритм:

1. Если задано `ViewData.ModelState["name"].Value.RawValue`, где *name* – это имя HTML-элемента, то это значение записывается в атрибут `value`. Иначе переходим к шагу 2. Словарь `ModelState` хранит данные формы, переданные в предыдущем запросе. Подробнее о `ModelState` будет рассказано в следующем параграфе.

2. Если задан второй строковый параметр методов расширения, то он записывается в атрибут `value`. Иначе переходим к шагу 3.

3. В атрибут `value` записывается результат вызова функции `ViewData.Eval("name")`. Функция `Eval()` сканирует словарь `ViewData`, а затем свойства объекта `Model` для нахождения элемента с указанным именем.

Методы расширения могут принимать дополнительный параметр, на основе которого строится набор атрибутов HTML-элемента. Это может быть словарь «строка-объект» или произвольный объект (как при работе с маршрутами).

```
<%= Html.TextBox("mytext", "val",
             new Dictionary<string, object>{ {"attr", "value"} }) %>

<%-- Эквивалентный вызов, но с использованием объекта --%>
<%= Html.TextBox("mytext", "val", new {attr="value"}) %>
```

Вторая группа методов предназначена для вывода ссылок и построения URL. Это методы расширения класса `HtmlHelper` - `ActionLink()` и `RouteLink()`, и методы расширения класса `UrlHelper` - `Content()`, `Action()` и `RouteUrl()`.

Метод `ActionLink()` генерирует тег `<a>` и имеет несколько перегруженных версий. В простейшем варианте указывается отображаемый текст и имя действия того же контроллера, который вызвал отображение представления.

```
<%= Html.ActionLink("Link text", "Action") %>
```

Можно явно задать имя контроллера (без суффикса `Controller`) и (или) передать набор значений для параметров маршрута, определяющего контроллер.

```
<%= Html.ActionLink("Text", "Action", "Home") %>
<%= Html.ActionLink("Text", "Action", new { controller = "Home" })%>
<%-- если параметра p нет в шаблоне URL,
     этот вызов транслируется в ссылку на /Home/Action?p=val --%>
<%= Html.ActionLink("Link text", "Action",
                    new { controller = "Home", p = "val" })%>
```

Метод `ActionLink()` также можно использовать для создания обычной веб-ссылки.

```
<%-- транслируется в https://www.example.com/Home/About#anchor --%>
<%= Html.ActionLink("Hi", "About", "Home", "https",
                    "www.example.com", "anchor", new { }, null)%>
```

Метод RouteLink() генерирует тег <a> для маршрута. В качестве параметров метод принимает текст ссылки, имя маршрута в таблице маршрутизации, набор параметров маршрута. Методы расширения класса UrlHelper похожи на методы ActionLink() и RouteLink(), но генерируют не тег <a>, а простую строку, описывающую ссылку.

Два метода расширения предназначены для вывода выпадающего списка и списка выбора[1]. Это методы DropDownList() и ListBox(). Данные методы используют параметры типов MultySelectList и SelectList. Эти классы создают списки выбора на основе коллекций.

```
<% var list = new SelectList(regionsData,  // коллекция объектов
                             "RegionID",   // поле объекта для value
                             "RegionName", // поле объекта для text
                             3);           // выбранное значение %>
<%= Html.DropDownList("List", list, "Choose") %>

<%= Html.ListBox("List", new MultiSelectList(new [] {"A", "B"})) %>
```

Метод расширения Encode() позволяет преобразовать текст, чтобы он не воспринимался браузерами как HTML. Например, символ < представляется в виде < и так далее. Использование Encode() уменьшает риск XSS-атак.

```
<%= Html.Encode("I'm <b>\"HTML\"-encoded</b>") %>
```

Если в проекте ASP.NET MVC установить ссылку на сборку Microsoft.Web.Mvc.dll и подключить одноимённое пространство имён, то становятся доступны ещё несколько методов расширения для HTML (вывод изображения, кнопка Submit, ссылка MailTo и другие).

Построение HTML-форм ввода

Для получения информации элементы ввода на HTML-странице должны быть сгруппированы в форму. Тег и атрибуты формы можно задать вручную или применить методы расширения класса HtmlHelper BeginForm() и EndForm(). Метод BeginForm() можно использовать с инструкцией using, что ведёт к автоматической генерации закрывающего тега формы.

```
<% Html.BeginForm("MyAction", "MyController"); %>
    здесь располагаются элементы ввода и кнопка submit
<% Html.EndForm(); %>

<%-- эквивалентный синтаксис --%>
<% using (Html.BeginForm("MyAction", "MyController")) { %>
    здесь располагаются элементы ввода и кнопка submit
<% } %>
```

[1] Вывод списков на странице можно организовать при помощи вкраплений серверного кода.

Достоинство метода `BeginForm()` заключается в том, что при помощи его параметров легко сформировать у формы правильный атрибут `action`. В простейшем варианте `BeginForm()` используется без параметров – в этом случае запрос отправляется тому же действию, которое вызвало представление. Можно указать имя действия и имя контроллера (см. пример выше), параметры маршрута, метод отправки формы (POST или GET) и имена и значения произвольных HTML-атрибутов:

```
<% Html.BeginForm("MyAction", "MyController",
                  new { param = "val" }, FormMethod.Get); %>
```

Работа с частичными представлениями

Web Forms View Engine поддерживает *частичные представления*, созданные как ascx-элементы управления. Базовым классом для частичных представлений является `System.Web.Mvc.ViewUserControl` или универсальный класс `System.Web.Mvc.ViewUserControl<T>`. Частичные представления обладают почти всеми свойствами и методами обычных представлений. Например, доступны свойства `ViewData`, `Html`, `Url`.

Если частичное представление создано, оно выводится на страницу при помощи метода расширения `Html.RenderPartial()`. Метод принимает в качестве параметра имя частичного представления и, возможно, пользовательский объект. Этот объект доступен в частичном представлении через свойство `ViewData.Model`.

Рассмотрим типичный пример использования частичных представлений. Опишем частичное представление `PersonInfo.ascx` для отображения информации объекта класса `Person`:

```
<%@ Control Language="C#"
            Inherits="System.Web.Mvc.ViewUserControl<Person>" %>
Возраст <%= Model.Name %> составляет <%= Model.Age %> лет
```

Используем `PersonInfo.ascx`, чтобы вывести список объектов:

```
<%@ Page Language="C#"
         Inherits="System.Web.Mvc.ViewPage< Person>" %>
<html>
  <body>
    Спискок людей:
    <ul>
        <% foreach (var p in (IEnumerable)ViewData["people"]) { %>
        <li>
            <% Html.RenderPartial("PersonInfo", p); %>
        </li>
        <% } %>
    </ul>
  </body>
</html>
```

7.6. ВВОД ДАННЫХ

В этом параграфе обсуждаются аспекты, связанные с пользовательским вводом и проверкой данных в приложениях ASP.NET MVC.

Привязка к модели

Привязка к модели – механизм ASP.NET MVC для связывания данных HTTP-запроса с параметрами действия и пользовательскими объектами. Пусть, например, имеется следующий метод-действие:

```
public ActionResult RegisterUser(string email, DateTime dateOfBirth)
{ . . . }
```

При вызове `RegisterUser()` компонент `ControllerActionInvoker` задействует классы[1] `DefaultModelBinder` и `ValueProviderDictionary`, которые для каждого параметра метода выполнят поиск и конвертирование соответствующих данных в запросе. Поиск осуществляется по имени параметра в словарях `Request.Form`, `RouteData.Values`, `Request.QueryString` по порядку (например, `Request.Form["email"]`, `RouteData.Values["email"]`, `Request.QueryString["email"]`). Если данные не найдены, параметры ссылочного типа получают значение `null`, а для параметров типов значений генерируется исключение `InvalidOperationException`. При конвертировании данных из словаря `Request.Form` применяются настройки языковой культуры сервера (это важно для таких параметров, как `dateOfBirth`).

Привязка к модели работает и для объектов пользовательских типов. При этом идентификаторы данных запроса *желательно* строить по схеме *`префикс.свойство`*, где *`префикс`* - имя параметра действия, *`свойство`* – имя атрибута объекта.

```
public ActionResult RegisterUser(Person p)
{ . . . }

<% using (Html.BeginForm("RegisterUser")) { %>
  <p>Email address: <%= Html.TextBox("p.Email") %></p>
  <p>Date of birth: <%= Html.TextBox("p.DateOfBirth") %></p>
  <input type="submit" value="Save" />
<% } %>
```

Для настройки привязки к модели можно использовать атрибут `[Bind]`, применяемый либо к параметру, либо к классу модели. При помощи `[Bind]` можно указать свойства объекта для применения или игнорирования привязки:

```
// У объекта p будем привязывать свойства Name и Email
public ActionResult RegisterUser(
                    [Bind(Include = "Name,Email")] Person p)
{ . . . }
```

[1] Можно заменить стандартную реализацию собственными компонентами.

```
// У объекта p будем привязывать все свойства, кроме DateOfBirth
public ActionResult RegisterUser(
                    [Bind(Exclude = "DateOfBirth")] Person p)
{ . . . }
```

Атрибут `[Bind]` также позволяет задать префикс идентификаторов данных на HTML-форме, если этот префикс отличается от имени параметра.

```
// На форме используем имена newuser.Name и т.п.
public ActionResult RegisterUser([Bind(Pefix = "newuser")] Person p)
{ . . . }
```

Привязка к модели работает не только в случае скалярных объектов, но и для коллекций. Пусть, например, имеется следующий фрагмент формы:

```
Введите названия трёх фильмов: <br />
<%= Html.TextBox("movies") %> <br />
<%= Html.TextBox("movies") %> <br />
<%= Html.TextBox("movies") %>
```

Данные формы могут быть приняты действием с параметром `List<string>`:

```
public ActionResult DoSomething(List<string> movies)
{ . . . }
```

Во всех приведённых примерах привязка к модели выполнялась инфраструктурой ASP.NET MVC. Программист может инициировать привязку, используя методы контроллера `UpdateModel()` и `TryUpdateModel()`. Оба метода получают в качестве параметра объект, атрибуты которого необходимо заполнить информацией. В случае ошибок привязки метод `UpdateModel()` генерирует исключение, а метод `TryUpdateModel()` просто возвращает значение `false`.

```
public ActionResult RegisterUser()
{
    var person = new Person();
    if (TryUpdateModel(person))
    {    // делаем что-то с объектом person
    }
    else
    {    // возникли ошибки преобразования
    }
}
```

В заключение подраздела приведём пример кода, в котором привязка к модели используется для получения данных загружаемого файла.

```
<%-- форма для загрузки файла сконструирована вручную,
     так как необходимо указать правильный enctype --%>
```

```
<form action="<%= Url.Action("UploadPhoto") %>" method="post"
      enctype="multipart/form-data">
    Upload a photo: <input type="file" name="photo" />
    <input type="submit" />
</form>

public ActionResult UploadPhoto(HttpPostedFileBase photo)
{
    // Сохраняем файл на диске
    photo.SaveAs("my_file.dat");
    // или работаем с файлом как с массивом байт
    byte[] uploadedBytes = new byte[photo.ContentLength];
    photo.InputStream.Read(uploadedBytes, 0, photo.ContentLength);
    . . .
}
```

Проверка данных

При выполнении привязки ASP.NET MVC использует специальное временное хранилище информации типа `ModelStateDictionary`, доступное через свойство контроллера `ModelState`. В `ModelState` хранятся все параметры запроса и информация об ошибках конвертирования, возникших при привязке. Программист может проверить наличие ошибок, обратившись к булевскому свойству `ModelState.IsValid`. Используя метод `ModelState.AddModelError()`, можно добавить для указанного параметра запроса сообщение об ошибке.

Рассмотрим пример работы с `ModelState`. Пусть имеется класс модели для описания деловой встречи:

```
public class Appointment
{
    public string ClientName { get; set; }
    public DateTime AppointmentDate { get; set; }
}
```

Создадим HTML-форму и контроллер для ввода данных о встрече.

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Назначение встречи</title>
</head>
  <body>
    <h1>Назначение встречи</h1>
    <% using (Html.BeginForm()) { %>
    <p>Имя клиента: <%= Html.TextBox("appt.ClientName") %> </p>
    <p>Дата встречи: <%=Html.TextBox("appt.AppointmentDate",
                        DateTime.Now.ToShortDateString()) %> </p>
    <p><%= Html.CheckBox("acceptsTerms") %>
    <label for="acceptsTerms"> Данные подтверждаю </label> </p>
```

```
    <input type="submit" value="Сохранить" />
    <% } %>
  </body>
</html>
```

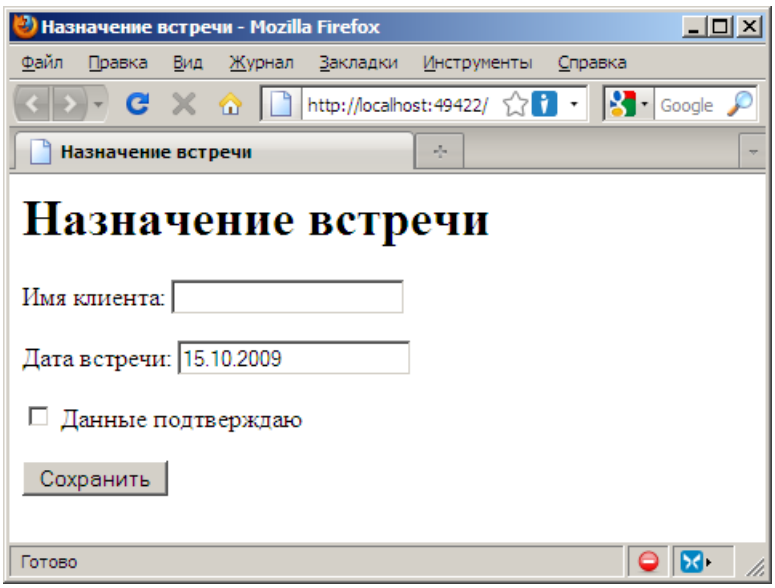

Рис. 4. Форма для ввода информации о встрече.

```
public class BookingController : Controller
{
    [AcceptVerbs(HttpVerbs.Get)]
    public ActionResult MakeBooking()
    {
        return View();
    }
}
```

Действие `MakeBooking()` показывает пустую форму. Так как на форме используется `BeginForm()` без параметров, при нажатии кнопки «Сохранить» форма будет отправлена действию `MakeBooking()`. Чтобы корректно обработать отправку формы, добавим в контроллер `BookingController` перегруженную версию `MakeBooking()`, указав другое значение в атрибуте `[AcceptVerbs]`:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult MakeBooking(Appointment appt, bool acceptsTerms)
{
    if (string.IsNullOrEmpty(appt.ClientName))
    ModelState.AddModelError("appt.ClientName", "Введите имя");
    if (ModelState.IsValidField("appt.AppointmentDate"))
    {
        // Формат даты правильный,
        // но нужно проверить на логическую корректность
        if (appt.AppointmentDate < DateTime.Now.Date)
            ModelState.AddModelError("appt.AppointmentDate",
```

```
                                        "Эта дата уже прошла");
        }
        if (!acceptsTerms)
            ModelState.AddModelError("acceptsTerms",
                                     "Нужно подтверждение");
        if (ModelState.IsValid)
        {
            // Здесь надо добавить сохранения инфо в БД
            // Показывает представление о том, что все ОК
            return View("Completed", appt);
        }
        else
           // Показываем форму, чтобы пользователь исправил ошибки
           return View();
    }
```

В перегруженной версии метода `MakeBooking()` информация о введенных данных проверяется, а найденные ошибки записываются в хранилище `ModelState`. Если ошибки есть (`ModelState.IsValid==false`), показывается представление с формой ввода. Чтобы пользователь увидел информацию об ошибках, нужно добавить вывод этой информации в код HTML-страницы. Для этого можно использовать метод расширения `Html.ValidationSummary()`.

```
<h1>Назначение встречи</h1>
<% using (Html.BeginForm()) { %>
<%= Html.ValidationSummary() %>
. . .
```

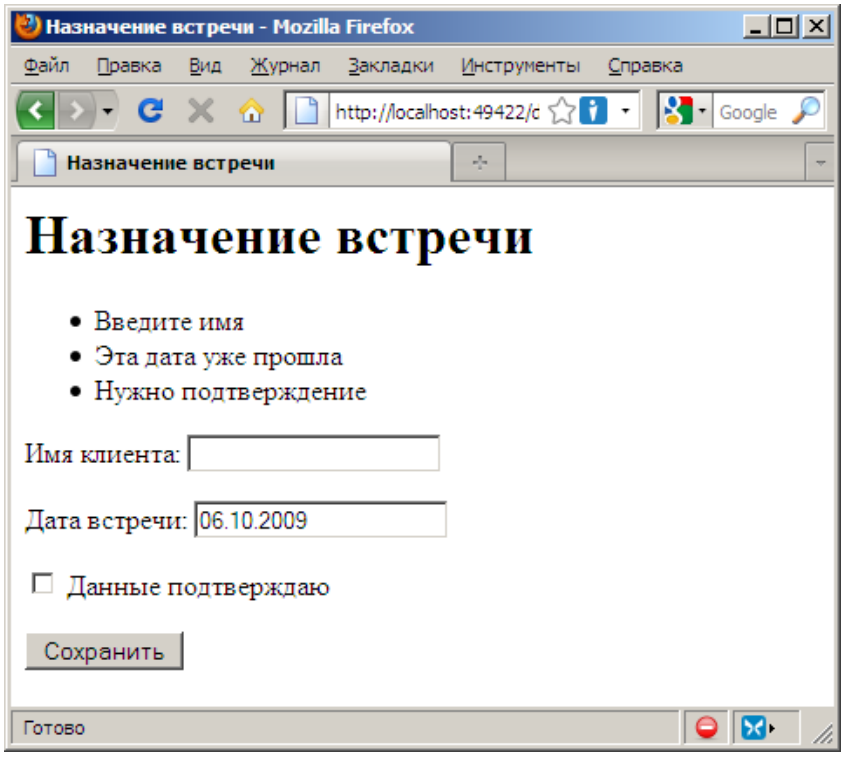

Рис. 5. Форма с информацией об ошибках.

Вместо вывода суммарного описания всех ошибок при помощи `Html.ValidationSummary()`, можно печатать точечные сообщения об ошибках, используя `Html.ValidationMessage()`:

```
<p>Имя клиента: <%= Html.TextBox("appt.ClientName") %>
                <%= Html.ValidationMessage("appt.ClientName")%></p>
```

7.7. ПОДДЕРЖКА AJAX В ASP.NET MVC

Для поддержки технологии AJAX в приложениях ASP.NET MVC можно воспользоваться методами расширения класса `System.Web.Mvc.AjaxHelper`: `ActionLink()`, `RouteLink()`, `BeginForm()` и `BeginRouteForm()`. В классе `ViewPage` определено свойство `Ajax`, имеющее тип `AjaxHelper`. Так как работа указанных методов расширения базируется на клиентских JavaScript-библиотеках `MicrosoftAjax.js` и `MicrosoftMvcAjax.js`, необходимо подключить эти библиотеки в представлении или на эталонной странице.

Рассмотрим применение метода `Ajax.ActionLink()`. Создадим представление для вывода времени в различных часовых зонах. При выборе зоны запрос будет отсылаться серверу асинхронно.

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>

<!doctype html public "-//w3c//dtd xhtml 1.0 strict//en"
                "http://www.w3.org/tr/xhtml1/dtd/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <script src="<%= Url.Content("~/Scripts/MicrosoftAjax.js") %>"
            type="text/javascript"></script>
    <script src="<%= Url.Content("~/Scripts/MicrosoftMvcAjax.js")%>"
            type="text/javascript"></script>
</head>
<body>
    <h2>Который час?</h2>
    <p>Показать время в зоне:
        <%= Ajax.ActionLink("UTC", "GetTime", new { zone = "utc" },
                new AjaxOptions { UpdateTargetId = "myRes" }) %>
        <%= Ajax.ActionLink("BST", "GetTime", new { zone = "bst" },
                new AjaxOptions { UpdateTargetId = "myRes" }) %>
        <%= Ajax.ActionLink("MDT", "GetTime", new { zone = "mdt" },
                new AjaxOptions { UpdateTargetId = "myRes" }) %></p>
    <div id="myRes" style="border: 2px dotted red; padding: .5em;">
        Результат здесь
    </div>
    <p>Страница сгенерирована
        <%= DateTime.UtcNow.ToString("h:MM:ss tt") %>(UTC)</p>
</body>
</html>
```

Обратите внимание на теги, которые подключают клиентские JavaScript-библиотеки. Метода `Ajax.ActionLink()` в нашем примере принимает как параметр текст ссылки, имя действия (`"GetTime"`), набор параметров для действия (`new { zone = "utc" }`) и указание на то, какой HTML-элемент следует обновить при получении ответа от сервера (`UpdateTargetId = "myRes"`).

Чтобы сделать пример полностью работоспособным, требуется в контроллере описать действие, вызываемое при AJAX-запросе. Для нашего случая достаточно, чтобы действие генерировало результат в виде простой строки.

```
public string GetTime(string zone)
{
    DateTime time = DateTime.UtcNow.AddHours(offsets[zone]);
    return string.Format(
                  "<div>Время в зоне {0} : {1:h:MM:ss tt}</div>",
                  zone.ToUpper(), time);
}

private Dictionary<string, int> offsets =
                       new Dictionary<string, int> { { "utc", 0 },
                                                     { "bst", 1 }, { "mdt", -6 }};
```

Заметим, что если браузер клиента не поддерживает JavaScript, метод `Ajax.ActionLink()` работает как обычная ссылка, генерируемая `Html.ActionLink()`. В серверном методе можно распознать AJAX-запросы при помощи булевской функции `Request.IsAjaxRequest()`.

```
public ActionResult GetTime(string zone)
{
    DateTime time = DateTime.UtcNow.AddHours(offsets[zone]);
    if (Request.IsAjaxRequest())
    {
        return Content(string.Format(
                      "<div>Время в зоне {0} : {1:h:MM:ss tt}</div>",
                      zone.ToUpper(), time));
    }
    else
    {
        // TODO: необходимо описать представление GetTime
        return View(time);
    }
}
```

Метод `Ajax.ActionLink()` имеет множество перегруженных версий, которые в отношении параметров в основном соответствуют версиям метода `Html.ActionLink()`. Однако `Ajax.ActionLink()` принимает дополнительный параметр типа `AjaxOptions` для конфигурирования AJAX-запроса. Параметры конфигурации представлены в табл. 5. Все параметры являются необязательными, кроме `UpdateTargetId`.

Таблица 5

Свойства класса `AjaxOptions`

Имя свойства	Описание
`Confirm`	Если задано это строковое свойство, перед отправкой AJAX-запроса показывается окно для подтверждения или отмены запроса
`HttpMethod`	Имя HTTP-метода (глагола), используемого при отправке запроса
`InsertionMode`	Режим работы с содержимым HTML-элемента для приёма ответа сервера (замещение содержимого, дополнение)
`LoadingElementId`	Идентификатор HTML-элемента, который становится видимым при отправке запроса и скрывается при получении ответа от сервера. Обычно свойство используется для *индикаторов запроса*
`OnBegin`	Имя JS-функции, вызываемой перед началом AJAX-запроса. Запрос не выполняется, если функция возвращает значение `false`
`OnComplete`	Имя JS-функции, вызываемой после завершения AJAX-запроса. Результат запроса игнорируется, если функция возвращает `false`
`OnSuccess`	Имя JS-функции, вызываемой после успешного завершения AJAX-запроса (вызывается после функции, указанной в `OnComplete`)
`OnFailure`	Имя JS-функции, вызываемой после неудачного завершения AJAX-запроса (вызывается после функции, указанной в `OnComplete`)
`UpdateTargetId`	Идентификатор того HTML-элемента, в который будет помещён ответ сервера
`Url`	Если значение указано, AJAX-запрос направляется на этот адрес

Иногда в AJAX-запрос необходимо включить пользовательские данные. В этом случае следует использовать метод `Ajax.BeginForm()`. Его параметры аналогичны параметрам метода `Html.BeginForm()` с дополнительным параметром типа `AjaxOptions`.