

## 6) Маршруты, фильтры и обработчики маршрутов MVC //

### Интернационализация приложений [ASP.NET](#)

#### Маршруты, фильтры и обработчики маршрутов MVC

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 параметрами 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.

### **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<sup>1</sup> (табл. 4):

Таблица 4

Интерфейсы и базовые реализации фильтров

| Вид фильтр  | Интерфейс                            | Базовый класс                         |
|-------------|--------------------------------------|---------------------------------------|
| Авторизация | <a href="#">IAuthorizationFilter</a> | <a href="#">AuthorizeAttribute</a>    |
| Действие    | <a href="#">IActionFilter</a>        | <a href="#">ActionFilterAttribute</a> |
| Результат   | <a href="#">IResultFilter</a>        | <a href="#">ActionFilterAttribute</a> |
| Исключение  | <a href="#">IExceptionFilter</a>     | <a href="#">HandleErrorAttribute</a>  |

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)
    {

```

---

<sup>1</sup> Все классы фильтров наследуются от [FilterAttribute](#). У [FilterAttribute](#) имеется целочисленное свойство `Order` для управления порядком применения фильтров.

```

        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 пользователь 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) { . . . }
```

### **6.1.3 Обработчики маршрутов**

Для создания собственного обработчика маршрутов необходимо помнить о процессе маршрутизации состоящем из следующих этапов:

1. Модуль `UrlRoutingModule` пытается сопоставить текущий запрос с маршрутами в таблице `RouteTable`.
2. Если сопоставление завершилось удачно, то модуль маршрутизации выбирает обработчик маршрутов сопоставленного маршрута - объект **`IRouteHandler`**.
3. Затем у объекта `IRouteHandler` вызывается метод `GetHandler`, который возвращает объект `IHttpHandler`, используемый для обработки запроса.
4. У обработчика `IHttpHandler` вызывается метод `ProcessRequest` для обработки запросов.

По умолчанию обработчик запросов или объект `IRouteHandler` представляет экземпляр класса `MvcRouteHandler`, который возвращает объект `MvcHandler`, применяющий интерфейс `IHttpHandler`. Этот объект `MvcHandler` отвечает за инициализацию контроллера, который потом уже вызывает одно из своих действий. Однако переопределить это поведение, указав при определении маршрута свой обработчик маршрутов. Для начала нужно создать сам обработчик. Создадим класс `MyRouteHandler`:

```
namespace Routing.RouteHandlers{
```

```
public class MyRouteHandler : IRouteHandler{

    public IHttpHandler GetHttpHandler(RequestContext requestContext)

    { return new MyHttpHandler();    } }
```

```
public class MyHttpHandler : IHttpHandler

{

    public bool IsReusable { get { return false; } }


    public void ProcessRequest(HttpContext context)

    { context.Response.Write("Hi Mickael!!!"); } }
```

Нам нужно собственно два класса: класс, реализующий интерфейс IHttpHandler, который и будет обрабатывать запрос; и класс, реализующий интерфейс IRouteHandler, который сопоставляется с маршрутом, и вызывает первый класс.

Теперь в классе RouteConfig можно прописать маршрут, который будет обрабатываться нашим обработчиком:

```
public class RouteConfig{

    public static void RegisterRoutes(RouteCollection routes)

    { routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapHttpRoute(

            name: "DefaultApi",

            routeTemplate: "api/{controller}/{id}",

            defaults: new { id = RouteParameter.Optional }

        );


        Route newRoute = new Route("{controller}/{action}", new MyRouteHandler());

        routes.Add(newRoute); } }
```

Если использовать этот обработчик в приложении то ему можно адресовать соответствующий запрос, например Home/Index после чего браузер выведем нашу строку из метода **ProcessRequest**.

## Интернационализация приложений ASP.NET

Если ваш веб-сайт ориентирован на пользователей из различных частей света, эти пользователи могут хотеть просматривать контент на вашем веб-сайте на их собственном языке. К счастью, .NET Framework уже имеет компоненты, которые поддерживают использование различных языков и культур. **Интернационализация** или **мультиязычность** оперирует терминами **Глобализация** и **Локализация**.

**Глобализация** - процесс проектирования приложений, поддерживающих различные культуры, **локализация** - процесс настройки приложения для определенной культуры. Благодаря мультиязычности сайт становится доступным гораздо большему числу потенциальных пользователей, обеспечивает большие выгоды, например, увеличение дохода, увеличение аудитории и т.д.

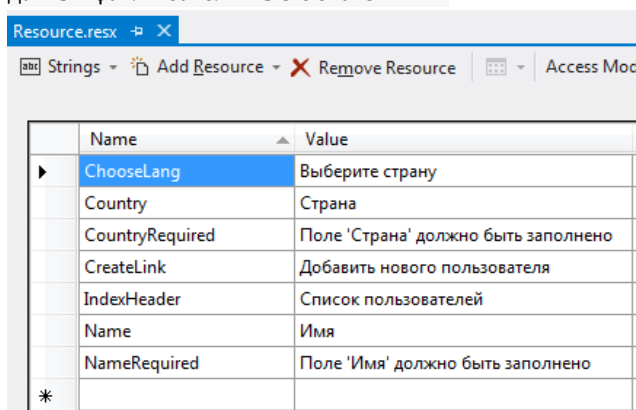
ASP.NET отслеживает два значения, связанных с культурой - Culture и UICulture. Значение Culture определяет результаты таких функций, как дата и форматирование валюты. UICulture определяет, какие ресурсы должны быть загружены ResourceManager. ResourceManager просто находит специфичные для культуры ресурсы, определяемые CurrentUICulture. Каждый тред в .NET имеет собственные объекты CurrentCulture и CurrentUICulture. Например, если текущая культура треда (CurrentCulture) определена как "en-US" (США, английский), DateTime.Now.ToLongDateString() выведет "Saturday, January 08, 2011", но если CurrentCulture определена как "es-CL" (испанский, Чили), метод выведет "sábado, 08 de enero de 2011".

Существует несколько концепций создания мультиязычного сайта:

- Создать для каждой отдельной культуры свое представление и затем в зависимости от выбранной культуры рендерить его в ответ клиенту.
- Использовать ресурсы, когда у нас одно представление для всех культур, и в зависимости от выбранной культуры подгружаются определенные для нее ресурсы.
- Смешивание вышеупомянутых концепций.

Если использовать второй вариант, то:

1. Нужно добавить модель, снабдить её свойствами, атрибуты к свойствам применяются по желанию программиста.
2. Создать контекст данных:  
`public class PersonContext : DbContext{`  
`public DbSet<Name> Names { get; set; } }`
3. Создать ресурсы. Добавить в проект папку Resources, добавить туда файл Resource.resx. Это будет файл ресурсов по умолчанию, который будет, к примеру использоваться для русского языка. Затем открыть данный файл и заполнить его значениями:



| Name            | Value                               |
|-----------------|-------------------------------------|
| ChooseLang      | Выберите страну                     |
| Country         | Страна                              |
| CountryRequired | Поле 'Страна' должно быть заполнено |
| CreateLink      | Добавить нового пользователя        |
| IndexHeader     | Список пользователей                |
| Name            | Имя                                 |
| NameRequired    | Поле 'Имя' должно быть заполнено    |
| *               |                                     |

Это и есть весь набор используемых ресурсов, и мы можем определить множество разных ресурсов. Подобный набор надо указать и у других ресурсов, только указав значения на соответствующем языке. Кроме того, мы можем в коде представлений обратиться к ресурсам, например: @Resources.Resource.NameRequired. Здесь Resources - пространство имен, Resource - тип ресурса (образуется от названия файла ресурсов), а NameRequired - имя. В таком же формате можно использовать и другие ресурсы. Далее можно создать файлы ресурсов и для других культур скопировав данный файл и поместив в ту же папку, предварительно переименовав.

Существуют различные способы определения культуры и непосредственной локализации, например, можно переопределять класс контроллера, доопределяя в нем OnActionExecuted и ExecuteCore. Но в данном случае



создадим свой фильтр действий, который будет срабатывать при обращении к действиям контроллера и производить локализацию.

4. Создать в проекте папку *Filters* и добавить в нее класс *CultureAttribute* со следующим содержимым:

```
public class CultureAttribute : FilterAttribute, IActionFilter{

    public void OnActionExecuted(ActionExecutedContext filterContext){

        string cultureName = null;

        // Получаем куки из контекста, которые могут содержать установленную культуру

        HttpCookie cultureCookie = filterContext.HttpContext.Request.Cookies["lang"];

        if (cultureCookie != null)        cultureName = cultureCookie.Value;

        else        cultureName = "ru";

        // Список культур

        List<string> cultures = new List<string>() { "ru", "en", "de" };

        if (!cultures.Contains(cultureName))

        {

            cultureName = "ru";

            //установка культуры

            Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName);

            Thread.CurrentThread.CurrentUICulture = CultureInfo.CreateSpecificCulture(cultureName);

        }

        public void OnActionExecuting(ActionExecutingContext filterContext)

        {

            //не реализован    }}

    }
```

Метод `OnActionExecuted` срабатывает после вызова действия контроллера. В начале он получает установленную культуру из куков. Затем мы смотрим, есть ли такая культура в списке.

5. Создать соответствующий контроллер и представление. По сути контроллер будет отличаться от обычного только тем, что к нему будет применен ранее созданный атрибут. И практически всю логику по локализации выполняет этот атрибут. [Culture]

Под *ресурсом* в данном параграфе понимается строка, изображение или данные иного типа, хранящиеся отдельно от исполняемых файлов (кода). Ресурсы позволяют изменять вид приложения, не производя его перекомпиляцию и, прежде всего, используются в задачах *локализации*.

*Файл ресурсов* – это обычный XML-файл, для которого принято использовать расширение `*.resx`. В файле хранятся данные в виде набора пар «имя ресурса-значение ресурса». Прежде всего, заметим, что с точки зрения приложения ASP.NET ресурсы делятся на *глобальные* и *локальные*. Глобальные ресурсы приложения хранятся в специальном каталоге приложения `App_GlobalResources`. Пусть, например, в этот каталог помещен файл `Resource.resx`, содержащий строковый ресурс `name` со значением `Alex`. Тогда доступ к этому ресурсу из кода страницы или из программного кода может быть выполнен так:

```
tbxSearch.Text = Resources.Resource.name;
```

Можно считать что ASP.NET создает для каждого глобального ресурса строго типизированный класс, имя которого совпадает с именем файла ресурсов (в нашем случае – `Resource`), а сами ресурсы доступны как свойства этого класса. Сгенерированные классы находятся в пространстве имен `Resources`.

Предполагается, что само имя файла непосредственно содержит указание на идентификатор культуры. Так, в предыдущем примере, имя `Resource.resx` обозначает ресурсы, используемые по умолчанию (если соответствие не будет найдено), имя `Resource.en.resx` обозначает ресурсы англоязычных стран, а имя `Resource.en-US.resx` – ресурсы для английского языка США. Контекст культуры для элементов управления определяет свойство `CurrentUICulture` класса `Thread`. Таким образом, при наличии файла `Resource.de-DE.resx` с соответствующим свойством `name`, следующий код будет использовать немецкий ресурс:

```
Thread.CurrentThread.CurrentUICulture = new CultureInfo("de-DE");  
  
tbxSearch.Text = Resources.Resource.name;
```

В реальных веб-приложениях информацию о культуре передает браузер клиента и она подобным образом обычно не устанавливается. Можно определить регион, используемый по умолчанию, в файле `web.config`:

```
<globalization enableClientBasedCulture="true"  
               culture="de-DE" uiCulture="de-DE"/>
```

Для указания региона для отдельной страницы можно применить директиву страницы и её атрибуты `Culture` и `UICulture`.

При создании страницы в Visual Studio достаточно переключится на дизайнер страниц и выполнить команду `Tools | Generate Local Resources`. Для страницы будет автоматически выполнена генерация файла локальных ресурсов. Он размещается в папке `App_LocalResources`, и его имя – такое же как у страницы, но с расширением `.resx`. Имена ресурсов в этом файле построены по схеме: *«имя элемента управления Resource номер.имя свойства»*. А если обратиться к коду разметки страницы, то у элементов управления можно заметить специальные атрибуты `meta:resourcekey`:

```
<asp:ImageButton ID="btnSearchGo" runat="server"  
                meta:resourcekey="btnSearchGoResource1" />
```

Несложно понять, что именно при помощи этих атрибутов связывается элемент управления и группа ресурсов с соответствующим именем в файле ресурсов. Все последующие ресурсы, специфические для данного региона, придется добавлять вручную, копируя сгенерированные ресурсы и присваивая им соответствующее имя (например, `Default.aspx.en-US.resx`).

Если необходимо локализовать не элемент управления на странице, а статический текст страницы, то следует воспользоваться новым элементом управления `Localize`. Он должен охватывать статический текст, тогда эта часть страницы будет автоматически включена в процесс генерирования ресурсов.