Модели

Модели

Одним из ключевых компонентов паттерна MVC являются **модели**. Ключевая задача моделей - описание структуры и логики используемых данных.

Модели

Все используемые сущности в приложении выделяются в отдельные модели, которые и описывают структуру каждой сущности. В зависимости от задач и предметной области мы можем выделить различное количество моделей в приложении.

Модели

Все модели оформляются как обычные классы на языке C#. Например, если мы работаем с данными пользователей, то мы могли бы определить в проекте следующую модель, которая представляет пользователя:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public int Age { get; set; }
}
```

Анемичная модель

Модель Person определяет ряд свойств: уникальный идентификатор Id, имя и возраст пользователя. Это классическая **анемичная модель**. Анемичная модель не имеет поведения и хранит только состояние в виде свойств.

Анемичная модель

В языке C# для представления подобных моделей удобно использовать классы record:

```
public record class Person(int Id, string Name, int Age);
```

Модели

Однако модель необязательно должна состоять только из свойств. Кроме того, она может иметь конструктор, какие-нибудь методы, поля, в общем представлять стандартный класс на языке C#. Модели, которые также определяют поведение, в противоположность анемичным моделям называют **"толстыми" моделями** (Rich Domain Model / Fat Model / Thick Model)

Модели

Например, мы можем уйти от анемичной модели, добавив к ней какое-нибудь поведение:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(int id, string name, int age)
    {
        Id = id;
        Name = name;
        Age = age;
    }
    public string PrintInfo() => $"{Id}. {Name} ({Age})";
}
```

Модели

В приложении ASP.NET MVC Core модели можно разделить по степени применения на несколько групп:

- Модели, объекты которых хранятся в специальных хранилищах данных (например, в базах данных, файлах xml и т. д.)
- Модели, которые используются для передачи данных представление или наоборот, для получения данных из представления. Такие модели еще называются моделями представления
- Вспомогательные модели для промежуточных вычислений

Модели

Как правило, для хранения моделей создается в проекте отдельная папка **Models**. Модели представления нередко помещаются в отдельную папку, которая нередко называется **ViewModels**. В реальности, это могут быть каталоги с любыми названиями, можно помещать модели хоть в корень проекта, но более распространенным стилем являются названия Models и ViewModels.

Модели

Класс **Person** в папке **Models**

```
namespace MvcApp.Models
{
    public record class Person(int Id, string Name, int Age);
}
```

Модели

Класс HomeController в папке Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcApp.Models; // пространство имен модели Person

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        List<Person> people = new List<Person>
        {
            new Person(1, "Tom", 37),
            new Person(2, "Bob", 41),
            new Person(3, "Sam", 28)
        };
        public IActionResult Index()
        {
            return View(people);
        }
    }
}
```

Модели

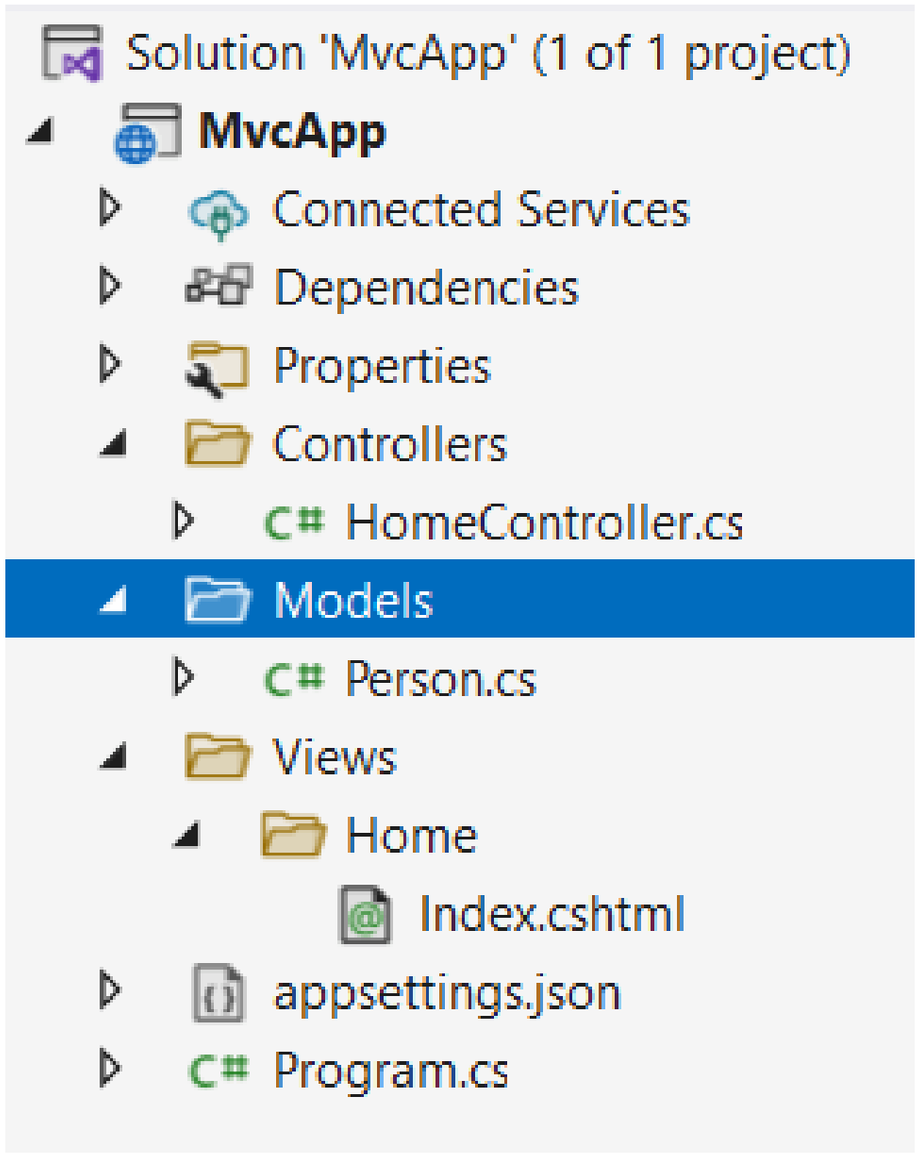
Представление Index.cshtml в папке Views/Home

```
@using MvcApp.Models
@model IEnumerable<Person>

<h2>People</h2>
<table class="table">
        <tr><td>Id</td><td>Name</td><td>Age</td></tr>
    @foreach (var p in Model)
    {
        <tr><td>@p.Id</td><td>@p.Name</td><td>@p.Age</td></tr>
    }
</table>
```

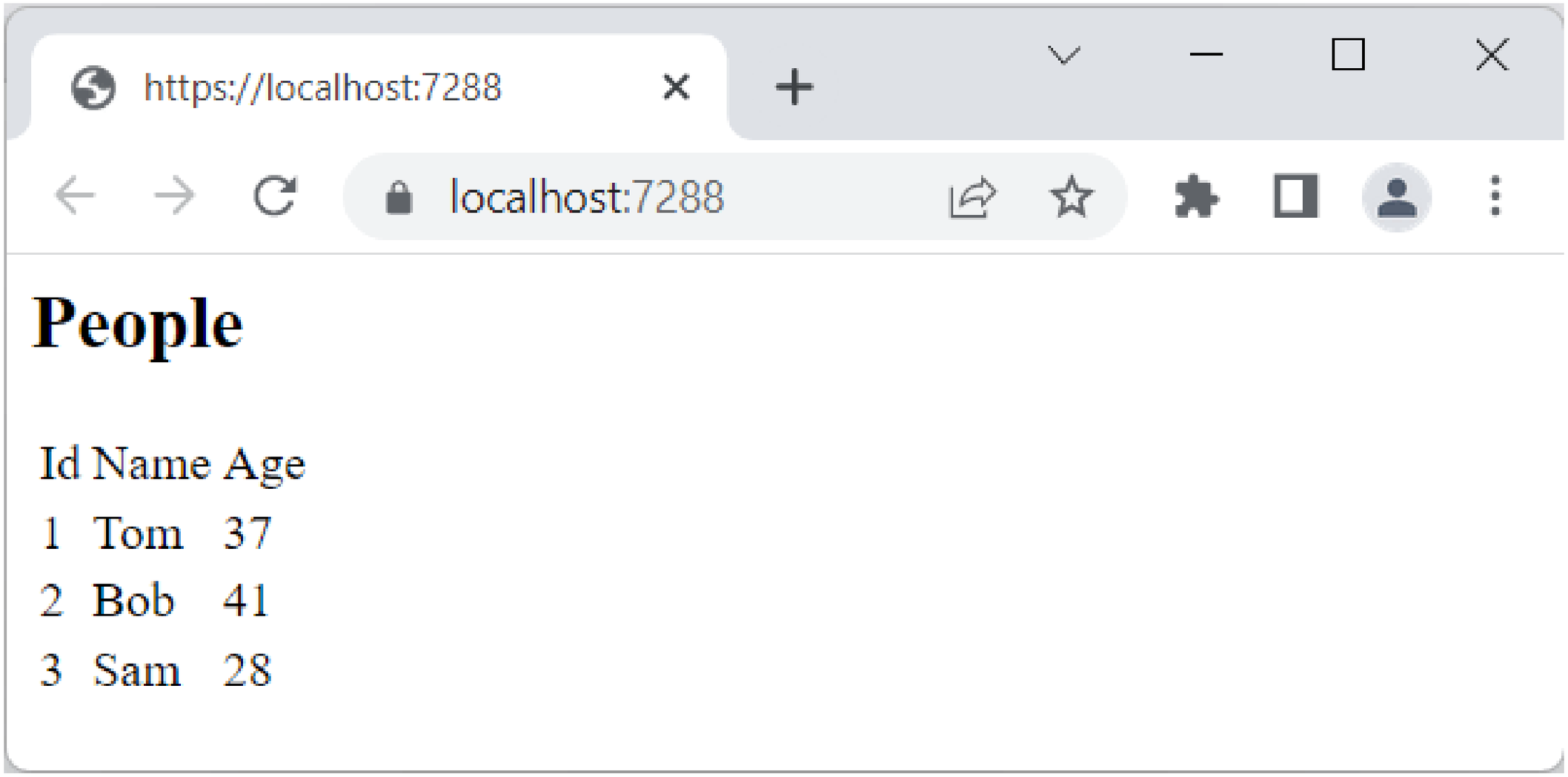
Модели

В итоге структура проект будет выглядеть следующим образом:



Модели

И при обращении к приложению веб-станица выведет список моделей:



Модели представления
View Model

Модели представления View Model

В зависимости от сложности проекта можно использовать одну и ту же модель для хранения данных в базе данных, для передачи данных в представление и получения данных из представления. Однако нередко все же модели могут не совпадать. Например, нам не надо передавать в представление все данные определенной модели или надо передать в представление объекты сразу двух моделей. И в этом случае мы можем воспользоваться **моделями представления**.

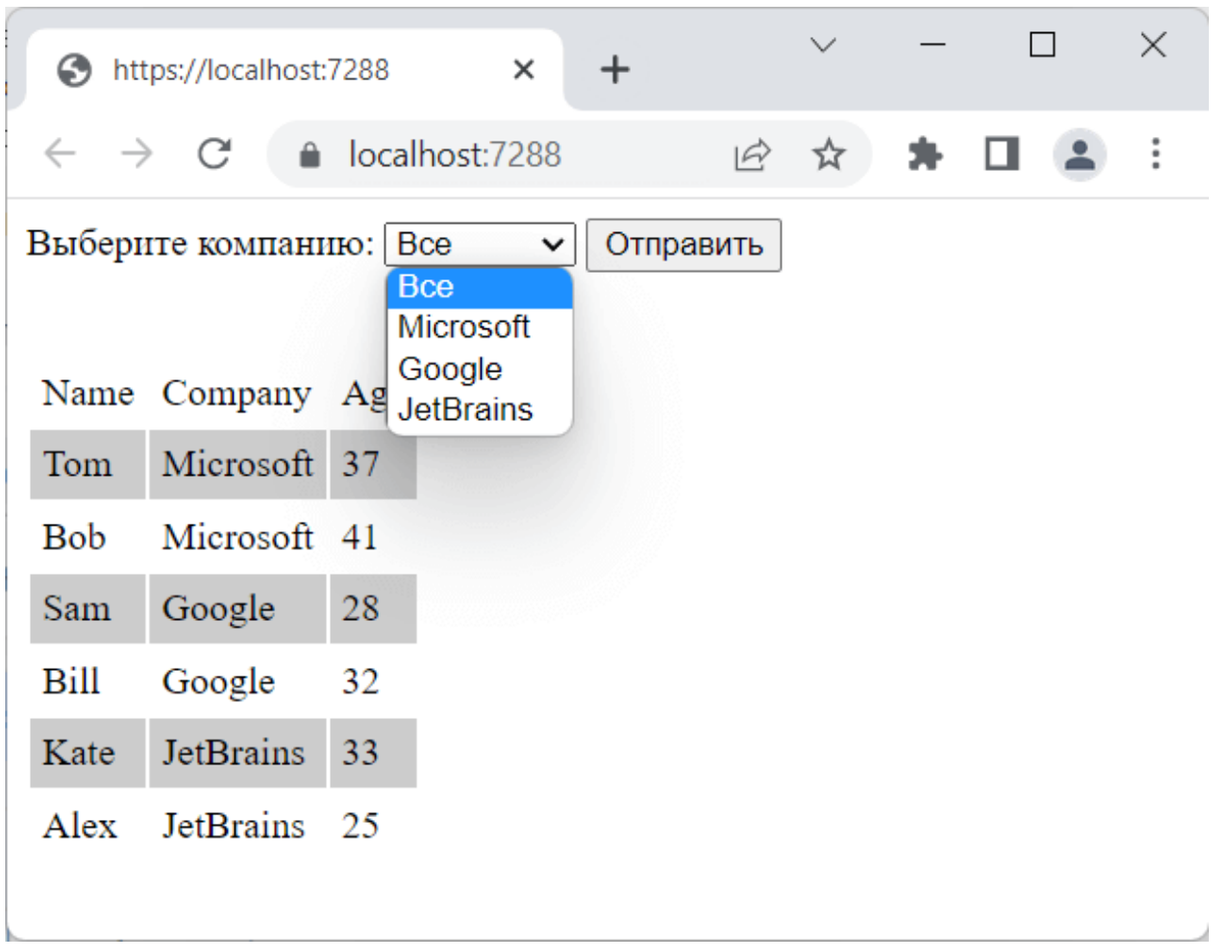
Модели представления View Model

Рассмотрим простейший пример работы с моделями. Допустим, в проекте в папке Model у нас есть следующие модели Person и Company.

```
namespace MvcApp.Models
{
    public record class Person(int Id, string Name, int Age, Company Work);
    public record class Company(int Id, string Name, string Country);
}
```

Модели представления View Model

И, допустим, нам надо выводить на страницу список пользователей и фильтровать их по компаниям. Наподобие следующего:



Модели представления View Model

Очевидно, что этих двух моделей - Person и Company для решения поставленной задачи нам недостаточно. И нам надо создадить специальную модель для передачи данных в представление или модель представления (иными словами View Model).

Модели представления View Model

Модель CompanyModel. Эта модель упрощает передачу списка компаний в представление.

```
namespace MvcApp.ViewModels
{
    public record class CompanyModel(int Id, string Name);
}
```

Модели представления View Model

Модель представления IndexViewModel. С помощью этой модели мы сможем передать в представление сразу и список компаний, и список пользователей.

```
using MvcApp.Models;

namespace MvcApp.ViewModels
{
    public class IndexViewModel
    {
        public IEnumerable<Person> People { get; set; } = new List<Person>();
        public IEnumerable<CompanyModel> Companies { get; set; } = new List<CompanyModel>();
    }
}
```

Модели представления View Model

HomeController:

```
public IActionResult Index(int? companyId)
{
  // формируем список компаний для передачи в представление
  List<CompanyModel> compModels = companies
        .Select(c => new CompanyModel(c.Id, c.Name)).ToList();

// добавляем на первое место
  compModels.Insert(0, new CompanyModel(0, "Все"));

  IndexViewModel viewModel = new() { Companies = compModels, People = people };

  // если передан id компании, фильтруем список
  if (companyId != null && companyId > 0)
     viewModel.People = people.Where(p => p.Work.Id == companyId);

  return View(viewModel);
}
```

Модели представления View Model

Представление Index.cshtml в папке Views/Home:

```
@using MvcApp.ViewModels
@using MvcApp.Models
@model IndexViewModel

<style>
td{padding:5px;}
tr:nth-child(even) {background: #CCC}
tr:nth-child(odd) {background: #FFF}
</style>

<form>
    <label>Выберите компанию:</label>
    <select name="companyId" >
        @foreach(CompanyModel comp in Model.Companies)
        {
            <option value="@comp.Id">@comp.Name</option>
        }
    </select>
    <input type="submit" />
</form>
<br />
<table>
    <tr><td>Name</td><td>Company</td><td>Age</td></tr>
    @foreach (Person p in Model.People)
    {
        <tr><td>@p.Name</td><td>@p.Work.Name</td><td>@p.Age</td></tr>
    }
</table>
```

Модели представления View Model

И теперь у нас получится веб-страница, как на первом скриншоте, на которой используется фильтрация.

Модели представления View Model

И теперь у нас получится веб-страница, как на первом скриншоте, на которой используется фильтрация.

Привязка модели

Привязка модели

Привязка модели или Model binding представляет механизм сопоставления значений из HTTP-запроса с параметрами метода контроллера. При этом параметры могут представлять как простые типы (int, float и т.д.), так и более сложные типы данных, например, объекты классов.

Привязка модели

Допустим, на сервер приходит запрос:

https://localhost:7288/Home/Index?name=Tom

```
public class HomeController : Controller
{
    public string Index(string name) => $"Name: {name}";
}
```

Привязка модели

При использовании стандартного маршрута для обслуживания данного запроса будет выбран метод *Index* контроллера *Home*. Поскольку данный метод принимает параметр с именем **name**, то механизм привязки по этому имени будет искать в среди пришедших данных значение с ключом *name*.

Привязка модели

Чтобы найти и сопоставить данные из запроса с параметрами метода используется привязчик модели (model binder), который представляет объект интерфейса IModelBinder.

Привязка модели

Для поиска значений привязчик модели используется следующие источники в порядке приоритета:

- Данные форм. Хранятся в объекте *Request.Form*
- Данные маршрута, то есть те данные, которые формируются в процессе сопоставления строки запроса маршруту. Хранятся в объекте *RouteData.Values*
- Данные строки запроса. Хранятся в объекте *Request.Query*

Привязка модели

То есть в нашем случае, когда на сервер придет запрос, привязчик модели последовательно будет просматривать в поиске значения для параметра name следующие пути:

- *Request.Form["name"]*
- *RouteData.Values["name"]*
- *Request.Query["name"]*

Управление привязкой

Управление привязкой

Фреймворк MVC предоставляет ряд атрибутов, с помощью которых мы можем изменить стандартный механизм привязки.

BindRequired и BindNever

Атрибут ***BindRequired*** требует обязательного наличия значения для свойства модели.

Атрибут ***BindNever*** указывает, что свойство модели надо исключить из механизма привязки.

Управление привязкой

Модель User:

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public int Age { get; set; }
    public bool HasRight { get; set; }
}
```

Управление привязкой

Метод, который в качестве параметра принимает объект модели User:

```
public string AddUser(User user)
{
    return $"Id: {user.Id}  Name: {user.Name}  Age: {user.Age}  HasRight:
  {user.HasRight}";
}
```

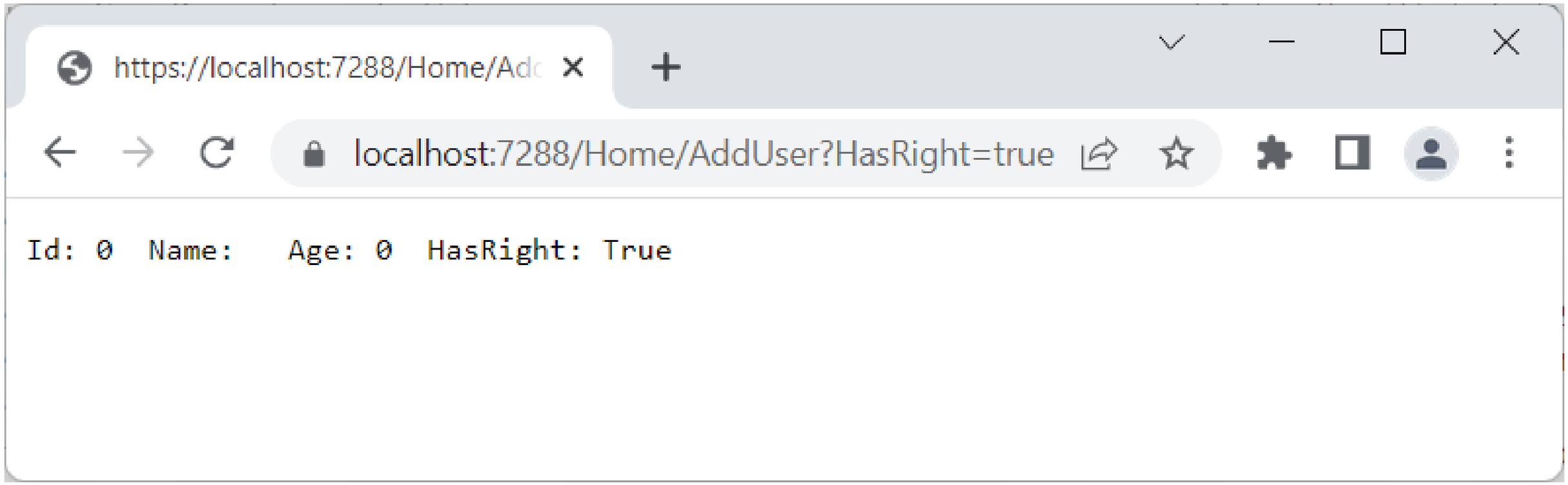
Управление привязкой

В данном случае не столь важно, отправляются данные через строку запроса или форму. Здесь важен механизм привязки. Так, мы можем обратиться к этому методу со следующим запросом:

https://localhost:7288/Home/AddUser?HasRight=true

Управление привязкой

И мы получим следующий вывод:



Управление привязкой

Это вполне валидный запрос, при обработке которого создается объект User. Для тех свойств, для которых не переданы значения, устанавливаются значения по умолчанию, например, для строковых свойств - пустые строки, для числовых свойств - число 0.

Управление привязкой

Но вряд ли подобный объект User можно считать удовлетворительным, поскольку, у него должно быть установлено, как минимум, имя - свойство Name. То есть имя выступает в качестве обязательного критерия, и чтобы это указать, используем атрибут BindRequired. А, к примеру, свойство HasRight не должно устанавливаться напрямую. Поэтому для него можно применить атрибут **BindNever**

Управление привязкой

```
using Microsoft.AspNetCore.Mvc.ModelBinding;

public class User
{
    public int Id { get; set; }

    [BindRequired]
    public string Name { get; set; } = "";

    public int Age { get; set; }
    [BindNever]
    public bool HasRight { get; set; }
}
```

Управление привязкой

Метод, который в качестве параметра принимает объект модели User:

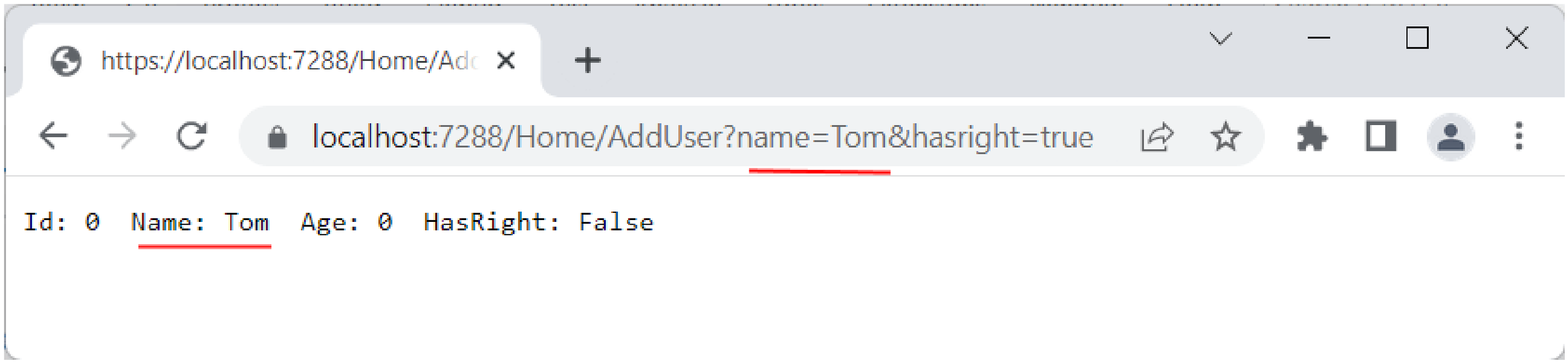
```
public string AddUser(User user)
{
    if (ModelState.IsValid)
    {
        return $"Id: {user.Id}  Name: {user.Name}  Age: {user.Age}  HasRight: {user.HasRight}";
    }
    string errors = $"Количество ошибок: {ModelState.ErrorCount}. Ошибки в свойствах: ";
    foreach(var prop in ModelState.Keys)
    {
        errors = $"{errors}{prop}; ";
    }
    return errors;
}
```

Управление привязкой

Если для свойства с атрибутом *BindRequired* не будет передано значение, то в объект **ModelState**, который представляет словарь, будет помещена информация об ошибках, а свойство *ModelState.IsValid* возвратит *false*. И в данном случае, проверяя значение *ModelState.IsValid*, мы можем проверить корректность создания объекта User. При этом все ключи в словаре *ModelState* будут представлять свойства, в которых произошли ошибки.

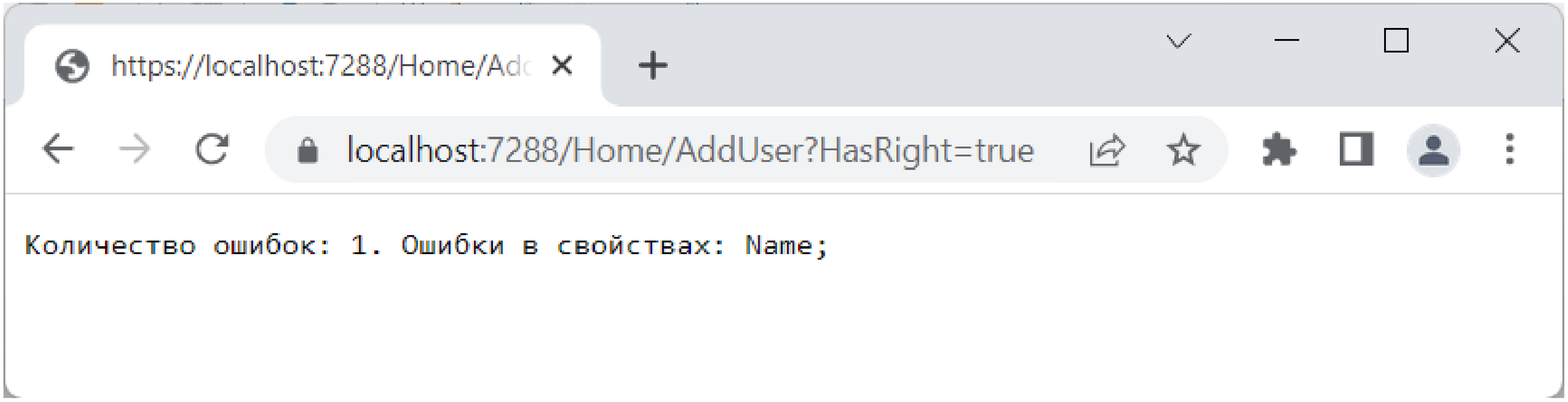
Управление привязкой

Теперь нам обязательно надо будет указать значение для свойства Name, а свойство HasRight будет исключено из привязки:



Управление привязкой

Если же для свойства Name не передать значение, то словарь ModelState будет содержать информацию об ошибке для этого свойства:



Управление привязкой

Кроме того, мы можем применять атрибут **BindingBehavior**, который устанавливает поведение привязки с помощью одно из значений одноименного перечисления **BindingBehavior**:

- **Required**: аналогично применению атрибута *BindRequired*
- **Never**: аналогично применению атрибута *BindNever*
- **Optional**: действие по умолчанию, мы можем передавать значение, а можем и не передавать, тогда будут применяться значения по умолчанию

Управление привязкой

Например, мы могли бы изменить модель User так:

```
public class User
{
    public int Id { get; set; }
    [BindingBehavior(BindingBehavior.Required)]
    public string Name { get; set; } = "";
    [BindingBehavior(BindingBehavior.Optional)]
    public int Age { get; set; }
    [BindingBehavior(BindingBehavior.Never)]
    public bool HasRight { get; set; }
}
```

Атрибут Bind

Атрибут **Bind** позволяет установить выборочную привязку отдельных значений. Так, уберем из модели User атрибуты привязки:

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public int Age { get; set; }
    public bool HasRight { get; set; }
}
```

Атрибут Bind

И применим атрибут *Bind* в методе *AddUser*:

```
public string AddUser([Bind("Name", "Age", "HasRight")] User user)
{
    return $"Name: {user.Name}  Age: {user.Age}  HasRight: {user.HasRight}";
}
```

В качестве параметра в атрибут Bind передается набор свойств объекта User, которые будут участвовать в процессе привязки. Здесь перечислены все свойства кроме Id.

Атрибут Bind

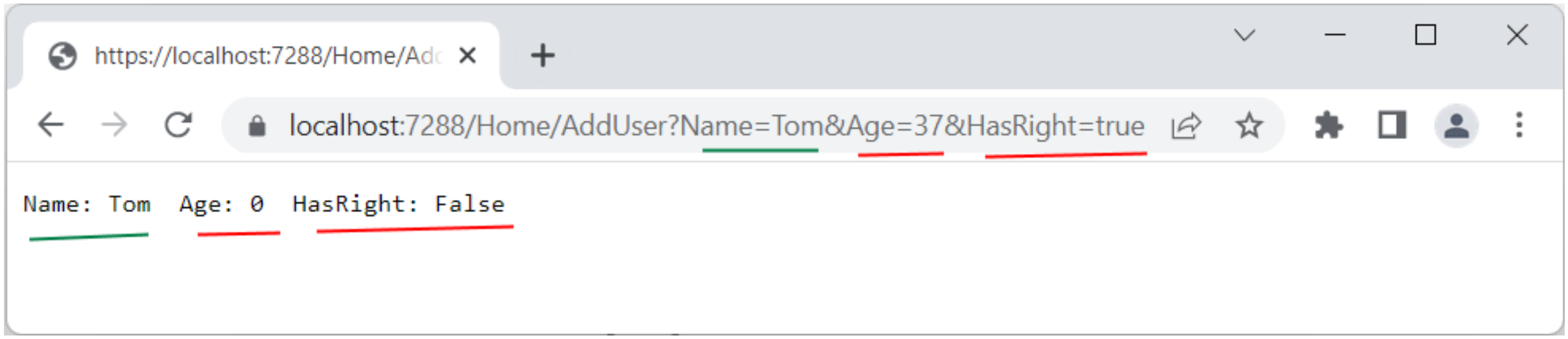
Но, допустим, уберем пару свойств:

```
public string AddUser([Bind("Name")] User user)
{
    return $"Name: {user.Name}  Age: {user.Age}  HasRight: {user.HasRight}";
}
```

Атрибут Bind

Теперь в привязке участвует только свойство Name, поэтому даже если в запросе мы передадим значения для всех остальных свойств, эти значения учитываться не будут, а для соответствующих свойств, не участвующих в привязке, будут применяться значения по умолчанию.

Атрибут Bind



Атрибут Bind

Так же, мы можем глобально переопределить привязку для модели User во всех методах. Для этого атрибут Bind применяется в целом к модели:

```
using Microsoft.AspNetCore.Mvc;

[Bind("Name")]

public class User
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public int Age { get; set; }
    public bool HasRight { get; set; }
}
```

Атрибут Bind

В этом случае в методе контроллера можно не применять данный атрибут к параметру:

```
public string AddUser(User user)
{
    return $"Name: {user.Name}  Age: {user.Age}  HasRight: {user.HasRight}";
}
```

Источники привязки

Существует группа атрибутов, которая позволяет переопределить поведения привязки, указав один целевой источник для поиска значений:

- **[FromHeader]:** данные извлекаются из заголовков запроса
- **[FromQuery]:** данные извлекаются из строки запроса
- **[FromRoute]:** данные извлекаются из значений маршрута
- **[FromForm]**: данные извлекаются из полученных форм
- **[FromBody]:** данные извлекаются из тела запроса.

Источники привязки

Например, получим данные о юзер-агенте из запроса:

```
public IActionResult GetUserAgent([FromHeader(Name="User-Agent")] string userAgent)
{
    return Content(userAgent);
}
```

В атрибут FromHeader передается строковый параметр, который указывает нужный заголовок.

HTML-хелперы

HTML-хелперы

Для вывода содержимого в представлении можно применять стандартные html-элементы, которые позволяют создавать блоки, списки, таблицы и т.д. Но кроме собственно html-элементов в ASP.NET Core MVC для создания разметки можно использовать специальные методы — **html-хелперы**. Html-хелперы представляют собой вспомогательные методы, цель которых - генерация html-разметки.

Создание HTML-хелперов

```
using Microsoft.AspNetCore.Html;          // для HtmlString
using Microsoft.AspNetCore.Mvc.Rendering;   // для IHtmlHelper

namespace MvcApp
{
    public static class ListHelper
    {
        public static HtmlString CreateList(this IHtmlHelper html, string[] items)
        {
            string result = "<ul>";
            foreach (string item in items)
            {
                result = $"{result}<li>{item}</li>";
            }
            result = $"{result}</ul>";
            return new HtmlString(result);
        }
    }
}
```

Создание HTML-хелперов

Так как данный метод расширяет функциональность html-хелперов, которые представляет интерфейс ***Microsoft.AspNetCore.Mvc.Rendering.IHtmlHelper***, то именно объект этого типа и передается в данном случае в качестве первого параметра. Второй параметр метода CreateList - массив строк-значений, которые потом будут выводиться в списке.

Использование HTML-хелперов

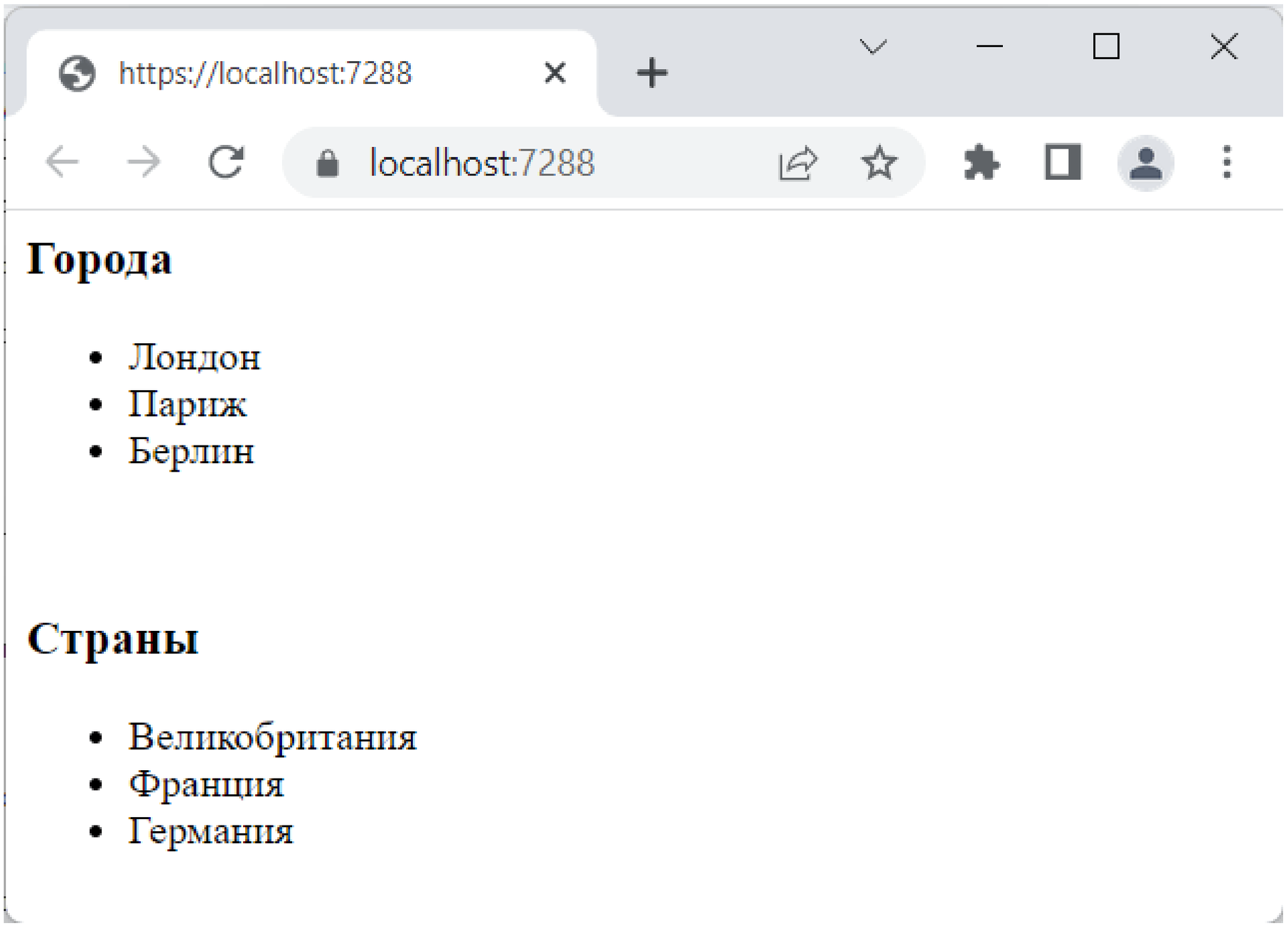
```
@{
    string[] cities = new string[] { "Лондон", "Париж", "Берлин" };
    string[] countries = new string[] { "Великобритания", "Франция", "Германия" };
}
@using MvcApp

<h3>Города</h3>
@Html.CreateList(cities)
<br />
<h3>Страны</h3>
<!-- или можно вызвать так -->
@ListHelper.CreateList(Html, countries)
```

Использование HTML-хелперов

Поскольку html-хелпер представляет метод расширения для объекта *IHtmlHelper*, то для его применения нам достаточно написать *Html.CreateList* и передать в метод необходимые параметры. Либо мы можем вызвать его как метод класса, в котором он определен: *ListHelper.CreateList*

Использование HTML-хелперов



TagBuilder

Для создания html-тегов в хелпере мы можем использовать класс **Microsoft.AspNetCore.Mvc.Rendering.TagBuilder.**

TagBuilder

```
public static class ListHelper
{
    public static HtmlString CreateList(this IHtmlHelper html, string[] items)
    {
        TagBuilder ul = new TagBuilder("ul");
        foreach (string item in items)
        {
            TagBuilder li = new TagBuilder("li");
            // добавляем текст в li
            li.InnerHtml.Append(item);
            // добавляем li в ul
            ul.InnerHtml.AppendHtml(li);
        }
        ul.Attributes.Add("class", "itemsList");
        using var writer = new StringWriter();
        ul.WriteTo(writer, HtmlEncoder.Default);
        return new HtmlString(writer.ToString());
    }
}
```

TagBuilder

В конструктор *TagBuilder* передается элемент, для которого создается тег. *TagBuilder* имеет ряд свойств и методов, которые можно использовать:

- Свойство **InnerHtml** позволяет установить или получить содержимое тега в виде строки.
- Свойство **Attributes** позволяет управлять атрибутами элемента.
- Метод **MergeAttribute()** позволяет добавить к элементу один атрибут.
- Метод **AddCssClass()** позволяет добавить к элементу класс css.
- Метод **WriteTo()** позволяет создать из элемента и его внутреннего содержимого строку при помощью объектов *TextWriter* и *HtmlEncoder*.

InnerHtml

Чтобы манипулировать свойством *InnerHtml*, можно вызвать один из методов:

- **Append(string text)**: добавление строки теста внутрь элемента
- **AppendHtml(IHtmlContent html)**: добавление в элемент кода html в виде объекта IHtmlContent - это может быть другой объект TagBuilder
- **Clear():** очистка элемента
- **SetContent(string text):** установка текста элемента
- **SetHtmlContent(IHtmlContent html):** установка внутреннего кода html в виде объекта IHtmlContent

КОНЕЦ