Маршрутизация. Контроллеры. Представления. Хелперы.

Маршрутизация

За сопоставление запросов с конкретными адресами внутри приложения в ASP.NET Core отвечает система маршрутизации.

Маршрутизация в MVC это процесс соответствия входящего запроса и обработчика запроса. Обработчиками запросов в MVC обычно выступают *действия контроллера*.

Запрос поступающий к серверу ASP.NET Core - это запрос **к файлу** или **к маршруту**.

Чтобы включить запрос *к статическим файлам* нужно добавить соответствующее промежуточное ПО в конвейер запросов. По умолчанию любой шаблон приложения в Visual Studio прописывает этот метод расширения в методе Configure.

```
app.UseStaticFiles();
```

Если это **middleware** не включено в конвейер обработки запроса или запрашиваемого файла не существует, то при запросе система сразу будет искать обработчик маршрута. Типичный обработчик маршрута – это контроллер.

Для направления запросов на соответствующие действия контроллера в MVC существует таблица маршрутизации. Добавление маршрута в таблицу маршрутизации называется ***регистрацией маршрута***.

Есть два API, которые можно использовать для регистрации маршрутов:

- Fluent API;
- атрибуты маршрутизации.

MVC поддерживает 2 типа маршрутизации:

- на основе соглашений (**Convention-Based Routing**);
- на основе атрибутов (**Attribute-Based Routing**).

Маршрутизация к действиям должна быть добавлена в конвейер обработки запросов.

В MVC можно использовать два подхода к определению системы маршрутизации:

- Определение и использование конечных точек с помощью компонентов **EndpointRoutingMiddleware** и **EndpointMiddleware (**применяется по умолчанию при создании нового проекта MVC).
- Использование **RouterMiddleware** и метода **UseMvc**.

Чтобы задействовать систему маршрутизации на основе конечных точек в приложении MVC, надо:

- добавить в классе **Startup** в методе ConfigureServices() сервисы MVC
- с помощью соответствующего middleware в методе Configure() определить как минимум один маршрут

```
app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern:
"{controller=Home}/{action=Index}/{id?}");
});
```

Добавляет в конвейер обработки запроса компонент **EndpointRoutingMiddleware**.

Встраивает в конвейер обработки запроса компонент **EndpointMiddleware**.

Компонент EndpointRoutingMiddleware позволяет определить маршрут, который соответствует запрошенному адресу, установить для его обработки конечную точку в виде объекта **Microsoft.AspNetCore.Http.Endpoint**, а также определить данные маршрута.

Компонент EndpointMiddleware определяет набор конечных точек, которые будут сопоставляться с определенными маршрутами и будут обрабатывать соответствующие маршрутам входящие запросы.

Метод app.UseEndpoints в качестве параметра принимает делегат Action с параметром **IEndpointRouteBuilder**, который добавляет конечные точки.

```
app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern:
"{controller=Home}/{action=Index}/{id?}");
            });
```

Шаблоны маршрутов

Шаблоны маршрутизации используют литералы и маркеры (для параметров маршрутизации). Литералы соответствуют точно тексту в URL, тогда как маркеры заменяются при сопоставлении маршрута.

public/{controller=Home}/{action=Index}/{id?}

Чтобы соответствовать шаблону, URL должен содержать маркеры контроллера и действия, так как это ключевая информация, которую MVC использует для поиска контроллера/действия. Другие маркеры в URL-адресе сопоставляются с параметрами методов действий **с помощью привязки модели**.

При использовании шаблона маршрута:

public/{controller=Home}/{action=Index}/{id?}

следующие URL сработают:

public/Home/Index/2
Имеется полное соответствие.

public/Home/Index
Тоже полное соответствие, id объявлен необязательным.

public/Home

Неполное соответствие. У третьего сегмента шаблона есть значение по умолчанию – Index.

public

У второго и третьего сегмента есть значения по умолчанию

Методы IEndpointRouteBuilder для добавления маршрутов:

MapControllerRoute(

string name, string pattern, [object defaults = null],
[object constraints = null], [object dataTokens = null]

) определяет произвольный маршрут .

name: название машрута

pattern: шаблон машрута

defaults: значения параметров маршрутов по умолчанию

constraints: ограничения маршрута

dataTokens: определения токенов маршрута

MapDefaultControllerRoute() определяет стандартный маршрут, эквивалентен вызову

```
endpoints.MapControllerRoute(
                    name: "default",
                    pattern:
"{controller=Home}/{action=Index}/{id?}");
```

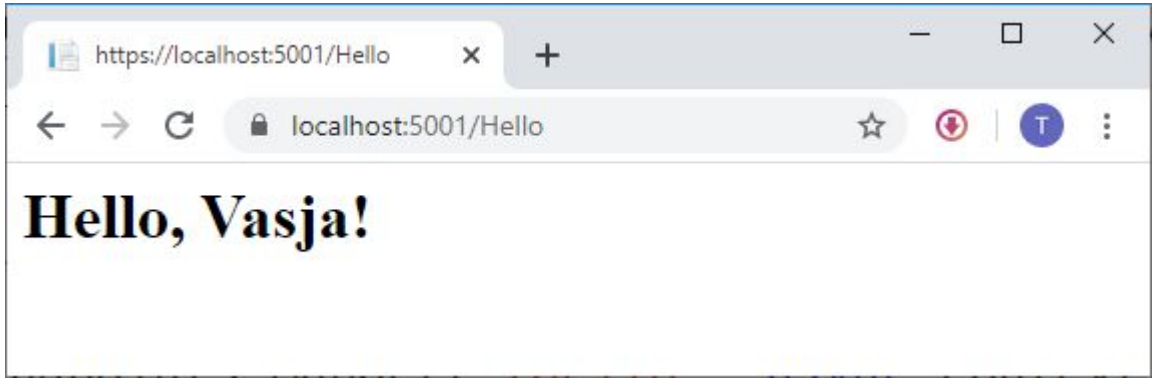
MapAreaControllerRoute(string name, string areaName, string pattern, [object defaults = null], [object constraints = null], [object dataTokens = null] **)** определяет маршрут, который также учитывает область приложения.

MapControllers() сопоставляет действия контроллера с запросами, используя маршрутизацию на основе атрибутов.

MapFallbackToController(string action, string controller) определяет действие контроллера, которое будет обрабатывать запрос, если все остальные определенные маршруты не соответствуют запросу.

MapGet() добавляет конечную точку для определенного маршрута по запросу типа GET и ее обработчик.

```
app.UseEndpoints(endpoints =>
    {
      endpoints.MapGet("/Hello", async context =>
        {
          await context.Response.WriteAsync("<h1>Hello, Vasja!</h1>");
        });
      endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
      });
```



Маршрутизация с помощью RouterMiddleware

Это старая форма маршрутизации. Была в Core 2.

Необходимо при подключении сервисов MVC в методе ConfigureServices *явным образом отключить использование конечных точек*.

Все методы для подключения сервисов MVC в качестве параметра могут принимать делегат **Action<MvcOptions>**. С помощью свойства **EnableEndpointRouting** объекта MvcOptions можно отключить использование конечных точек:

```
services.AddControllersWithViews(mvcOtions=>
    {
        mvcOtions.EnableEndpointRouting = false;
    });
```

Для определения маршрутов в методе Configure применяется метод UseMvc():

```
app.UseMvc(routes =>
{
    routes.MapRoute("api", "api/get", new { controller =
"Home", action = "About" });

    routes.MapRoute(
        name: "default",
        template:
"{controller=Home}/{action=Index}/{id?}");
});
```

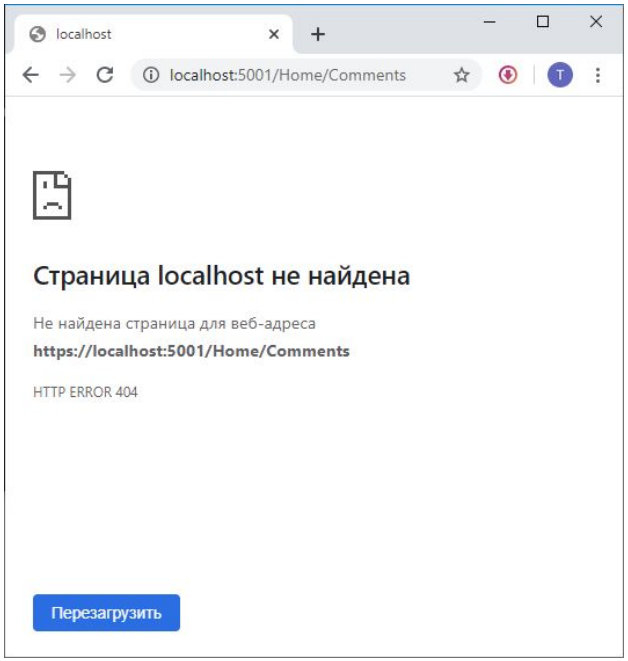
При составлении таблиц маршрутизации важно, чтобы самые специфичные были зарегистрированы первыми, поскольку, как только будет найдено первое соответствие входящему URL поиск в таблице завершится.

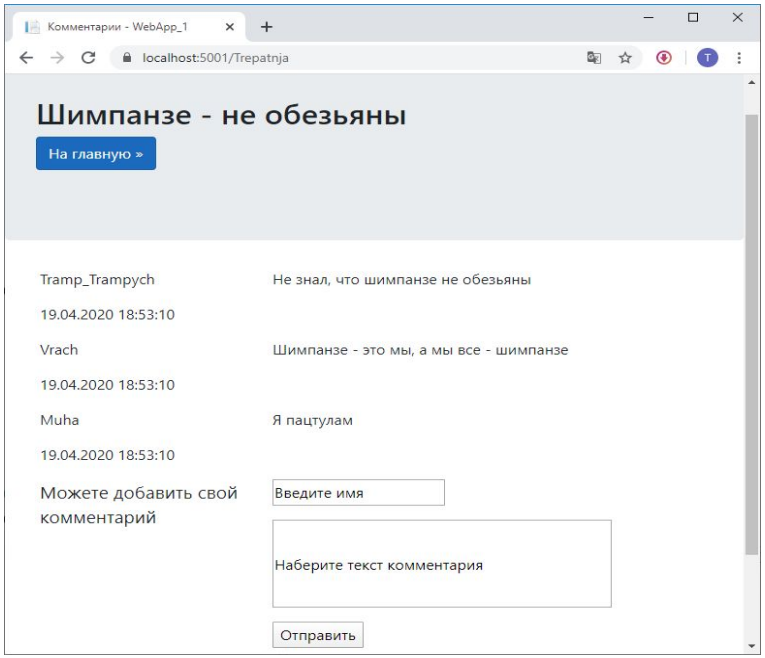
Маршрутизация на основе атрибутов

Маршруты, определенные с помощью атрибутов, имеют приоритет по сравнению с маршрутами, определенными в классе Startup.

Для определения маршрута необходимо использовать атрибут **[Route].** В качестве параметра атрибут Route принимает шаблон URL, с которым будет сопоставляться запрошенный адрес. Этот атрибут применяется к действиям контроллера.

```
[Route("Trepatnja")]
    [HttpGet]
    public ActionResult Comments()
    {
       var comments = CommentsDb.Comments;
       return View(comments);
    }
```





Если в проекте планируется использовать только маршрутизацию на основе атрибутов, то в классе Startup можно не определять никаких маршрутов. Например, при использовании EndpointMiddleware и конечных точек можно применить метод **MapControllers**.

Контроллеры

Контроллер представляет обычный класс, который наследуется от базового класса **Microsoft.AspNetCore.Mvc.Controller**.

В свою очередь класс Controller реализует абстрактный базовый класс ControllerBase.

Таким образом, чтобы создать свой класс контроллера, достаточно создать класс, наследующий от Controller и имеющий в имени суффикс *Controller*.

Методы действий (action methods) представляют такие методы контроллера, которые обрабатывают запросы по определенному URL.

Так как запросы бывают разных типов, например, GET и POST, ASP.NET MVC позволяет определить тип обрабатываемого запроса для действия, применив к нему соответствующий атрибут: [HttpGet], [HttpPost], [HttpDelete] или [HttpPut].
http://datanets.ru/tipy-zaprosov-http-protokola-get-post-put.html
https://habr.com/post/50147/

Не все методы контроллера являются методами действий. Методы действий всегда имеют модификатор **public**. Закрытых приватных методов действий не бывает.

Но контроллер может также включать и обычные методы, которые могут использоваться в вспомогательных целях.

Для контроллеров предназначена папка **Controllers**. По умолчанию при создании проекта в нее добавляется контроллер **HomeController.**

Чтобы указать, что класс не является контроллером (т.е. не может обрабатывать запрос), нам надо использовать над ним атрибут **[NonController]**

Если нужно, чтобы какой-то public метод контроллера не рассматривался как действие, то можно применить к нему атрибут [**NonAction]**.

Атрибут **[ActionName]** позволяет для метода задать другое имя действия.

```
[HttpPost]
        [ActionName("InsertComments")]
        public ActionResult AddComments()
        {

            Comment comment = new Comment();
            comment.Nik = Request.Form["Nik"];
            comment.Text = Request.Form["Text"];
            comment.Time = DateTime.Now;
            CommentsDb.Comments.Add(comment);

            return RedirectToAction("Comments");
        }
```

Результаты действий

https://metanit.com/sharp/aspnet5/5.3.php

Метод контроллера формирует некоторый ответ в виде результата действия, *как правило*, это объект класса, реализующего интерфейс **IActionResult.**

Интерфейс IActionResult находится в пространстве имен *Microsoft.AspNetCore.Mvc*

```
public interface IActionResult
{
    Task ExecuteResultAsync(ActionContext context);
}
```

Этот интерфейс реализуется абстрактным базовым классом **ActionResult**. В нем есть синхронный метод **ExecuteResult**, который выполняется в асинхронном.

Для создания своего класса результата действий нужно либо унаследовать его от ActionResult, либо реализовать интерфейс IActionResult.

```
public class MyActionResult:ActionResult
    {
        private string text;
        private string action;
        public MyActionResult(string text, string action)
        {
            this.text = text;
            this.action = action;
        }
public override async Task ExecuteResultAsync(ActionContext context)
        {
            string fullCode = "<!DOCTYPE html><html><head>";
            fullCode += "<title>Суперпроект</title>";
            fullCode += "<meta charset=utf-8 />";
            fullCode += "</head> <body>";
            fullCode += "<h2>" + text + "</h2>";
            fullCode += "<a href=\"/Home/\"" + action + ">Куда-то</a>";
            fullCode += "</body></html>";
            await context.HttpContext.Response.WriteAsync(fullCode);
        }
```

Метод действия в контроллере:

```
[Route("/")]
    public IActionResult GetResult()
    {
        return new MyActionResult("Привет, Вася!", "Index");
    }
```

Стандартные результаты действий:

ContentResult: пишет указанный контент напрямую в ответ в виде строки

EmptyResult: отправляет пустой ответ в виде статусного кода 200

FileResult: является базовым классом для всех объектов, которые пишут набор байтов в выходной поток. Предназначен для отправки файлов

И т.д. https://metanit.com/sharp/aspnet5/5.3.php

Например,

```
public IActionResult Index()
        {
        return NotFound("Ничего не найдено");
        }
```

ViewResult: производит рендеринг представления и отправляет результаты рендеринга в виде html-страницы клиенту

Чтобы возвратить объект ViewResult используется метод **View**

```
return View();
```

По умолчанию контроллер производит поиск представления в проекте по следующему пути:
/Views/Имя_контроллера/Имя_представления.cshtml

В качестве представления будет использоваться то, имя которого совпадает с именем действия.

Можно также задать имя представления явным образом:

```
return  View("Index");
```

Можно полностью переопределить путь, по которому система будет искать представление:

```
return View("~/Views/Some/Index.cshtml");
```

Передать данные из контроллера в представление можно, используя объекты **ViewData и ViewBag.**

```
ViewBag.User = "Вася";
ViewData["User"] = "Вася";
```

В представлении

```
@Html.TextBox("User",ViewData["User"])
```

Так нельзя: `@Html.TextBox("User",ViewBag.User)`

Так можно: @Html.TextBox("User",(string)ViewBag.User)

RedirectResult: перенаправляет пользователя по другому адресу URL

Для временной переадресации применяется метод **Redirect**:

```
return Redirect("/Home/Index");
```

RedirectToRouteResult: перенаправляет пользователя по определенному адресу URL, указанному через параметры маршрута

Метод **RedirectToAction** позволяет перейти к определенному действию определенного контроллера, а также позволяет задать передаваемые параметры:

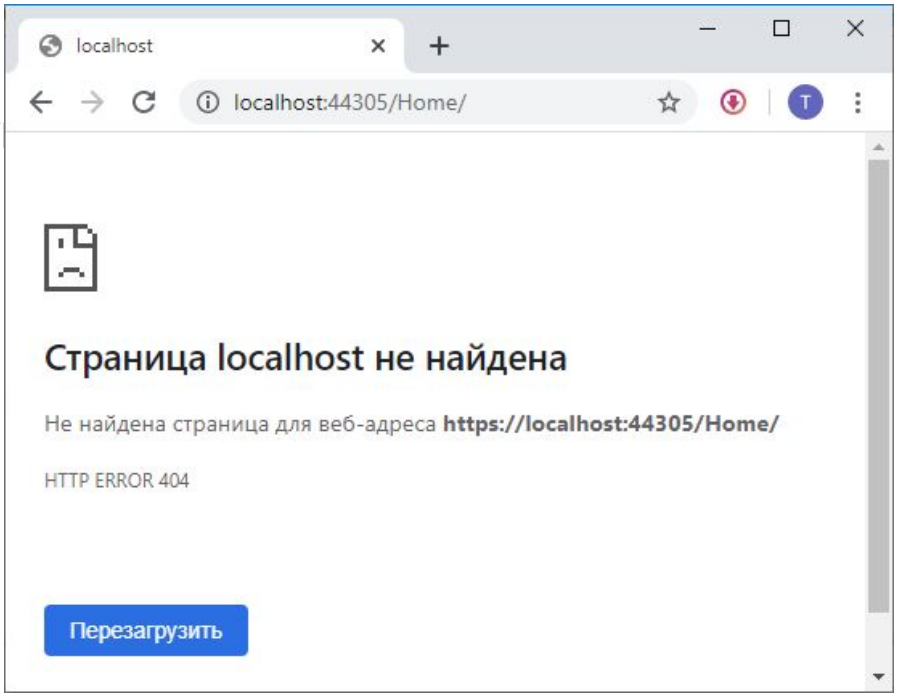
Например,

```
return RedirectToAction("Index");
return RedirectToAction("Square", "Home", new { a=10,h=12});
```

HttpNotFoundResult: возвращает клиенту ответ в виде статусного кода HTTP 404, указывая, что запрошенный ресурс не найден.

```
public ActionResult Buy(int? id)
        {
            if (id == null)
                return NotFound();
            ViewBag.ProductId = id;
    ViewBag.Title = "Оформление поку

            return View();
        }
```



Передача данных в контроллер

При отправке **GET-запроса** значения передаются через строку запроса.

Стандартный get-запрос:

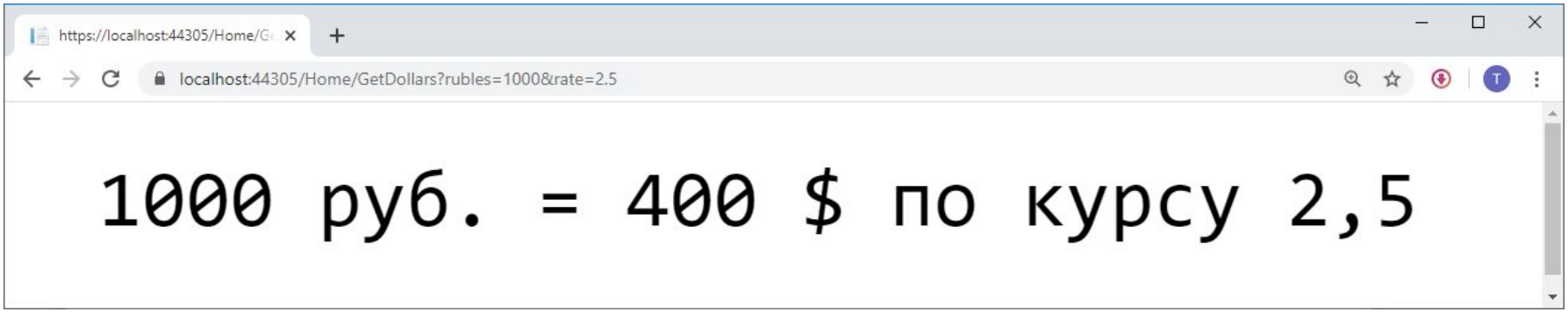
название_ресурса?параметр1=значение1&параметр2=значение2

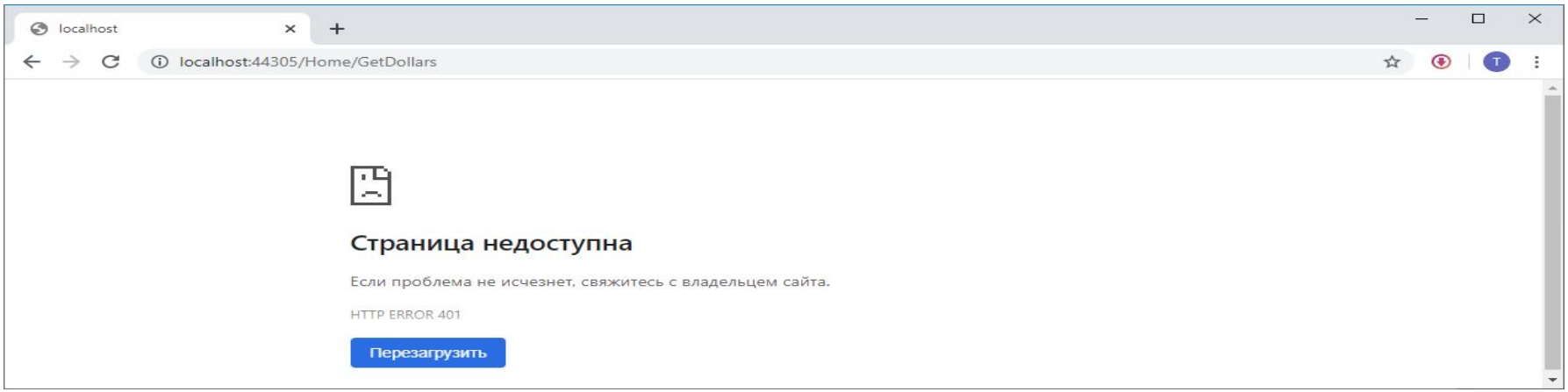
Например, определим следующий метод:

```
public IActionResult GetDollars(double? rubles,double? rate)
        {
            if(rubles!=null && rate!=null)
            {
                double dollars = rubles.Value / rate.Value;
                return Content(
   $" {rubles} руб. = {dollars} $ по курсу {rate}");
            }
            return StatusCode(401);
        }
```

Запрос этого действия:

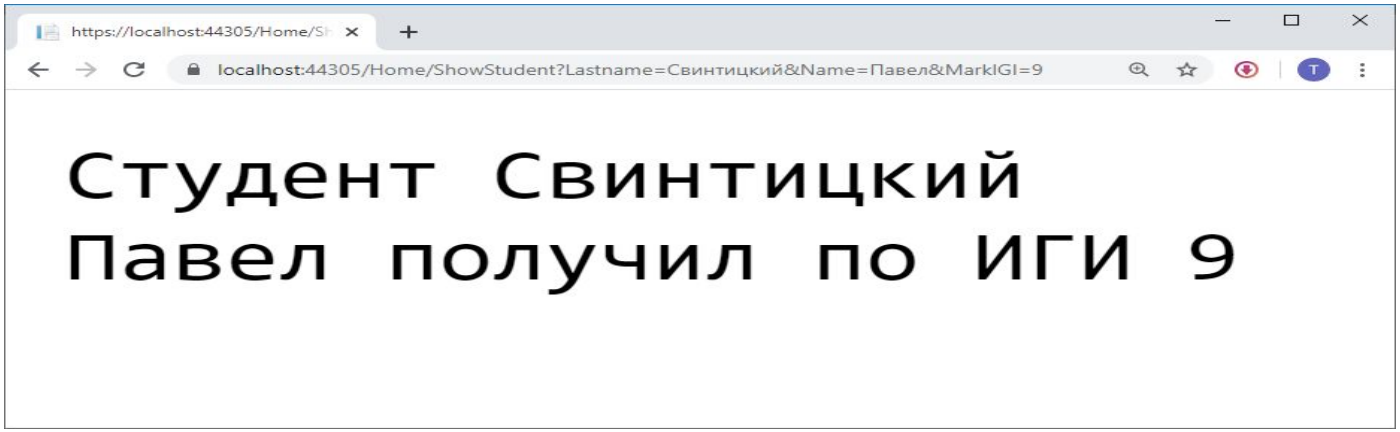
https://localhost:44305/Home/GetDollars?rubles=1000&rate=2.5





Передача в действие контроллера объекта класса

```
public IActionResult ShowStudent(Student student)
        {
            return Content($"Студент {student.Lastname}
{student.Name} получил по ИГИ {student.MarkIGI}");
        }
```



Передача данных в запросе POST

Чтобы система могла связать параметры метода и данные формы, необходимо, чтобы атрибуты name у полей формы соответствовали названиям параметров.

```
<form method="post" action="~/Home/ShowStudent">
        <label>Фамилия:</label><br />
        <input type="text" name="Lastname" /><br />
        <label>Имя:</label><br />
        <input type="text" name="Name" /><br />
        <label>Оценка по ИГИ:</label><br />
        <input type="number" name="MarkIGI" /><br />
        <input type="submit" value="Отправить" />
    </form>
```

Данные в действии можно получать, не используя параметры, через свойство Request контроллера.

```
public ActionResult AddComments()
        {

                Comment comment = new Comment();
                comment.Nik = Request.Form["Nik"];
                comment.Text = Request.Form["Text"];
                comment.Time = DateTime.Now;
                CommentsDb.Comments.Add(comment);

            return RedirectToAction("Comments");
        }
```

```
comment.Nik = Request.Form.FirstOrDefault(p=>p.Key=="Nik").Value;
```

Представления

Представление не является html-страницей. При компиляции приложения на основе требуемого представления сначала генерируется класс на языке C#, наследующий от класса **Microsoft.AspNetCore.Mvc.Razor.RazorPage<T>**
, где T - это класс модели, которая будет использоваться, или ключевое слово **dynamic**. Затем этот класс компилируется.
https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.razor.razorpage-1?view=aspnetcore-3.1

Все добавляемые представления, как правило, группируются *по контроллерам* в соответствующие папки в каталоге **Views**.

Для рендеринга представления в выходной поток, используется метод View().

Контроллер готовит данные и выбирает представление, а затем объект ViewResult обращается к *движку представления* **Razor** для рендеринга представления в выходной результат.
https://habr.com/post/98241/

Способы передачи данных из контроллера в представление:

- **ViewData**
- **ViewBag**
- **Модель представления**

ViewData представляет словарь из пар ключ-значение.

```
ViewData["Title"]= "Шимпанзе - не обезьяны";
```

В качестве значения может выступать любой объект.

ViewBag позволяет определить различные свойства и присвоить им любое значение.

```
ViewBag.Title = "Шимпанзе - не обезьяны";
```

Строго типизированные представления

Позволяют передавать данные не через объект ViewBag и т.п., а напрямую в представление *через параметр* метода View.

Чтобы связать представление с передаваемым параметром, надо добавить в представление директиву **@model** с указанием **типа передаваемых данных**

Объект **Model** представляет тип модели и будет содержать переданные из контроллера данные

```
@using MvcSimple.Models;
@model IEnumerable<Product>
    <div>
        <table border="1">
            <tr>
                <td><p>Товар</p></td>
                <td><p>Категория</p></td>
                <td><p>Цена</p></td>
                <td><p>Выбор</p></td>
            </tr>
            @foreach(var b in Model)
            {
                <tr>
                    <td><p>@b.Name</p></td>
                    <td><p>@b.ProductCategory</p></td>
                    <td><p>@b.Price</p></td>
                    @*адрес, по которому будет размещаться форма
оформления покупки*@
                    <td><p><a
href="/Home/Buy?id=@b.id&item=@b.Name">Купить</a></p></td>
                </tr>
            }
        </table>
    </div>
```

```
public ActionResult Index()
        {
            // получаем из бд данные
            ViewBag.Title = "Магазин";
            var products = db.Products;

            // возвращаем представление

            return View(products);
        }
```

Можно автоматически создавать строго типизированное представление, указав в диалоговом окне при создании представления соответствующие параметры.

Мастер-страница может иметь несколько секций, куда представления могут поместить свое содержимое.

Добавим к мастер-странице секцию footer

```
<body>
    @Html.ActionLink("Главная", "Index", "Home")

    @RenderBody()

    <footer>
        @RenderSection("Footer", false)
    </footer>

</body>
```

В представлении

```
@section Footer{
    Разаработчик Вася
    }
```

Код страницы **_ViewStart.**cshtml выполняется до кода любого из представлений, расположенных в том же каталоге. Этот файл последовательно применяется к каждому представлению, находящемуся в одном каталоге.

HTML-хелперы

Позволяют генерировать html-код.

Для создания хелпера в проекте нужно создать папку под названием **App_Code**. Это специальное зарезервированное имя для папки, которая содержит html-хелперы, поэтому нужно использовать данное название.

Хелперы представляют собой методы расширения, поэтому в папку **App_Code** для создания хелперов нужно добавить статический класс.

```
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);
        }
    }
```

Пример использования:

```
@{
    ViewData["Title"] = "Home Page";
    string[] students = { "Вася", "Петя", "Маша", "Даша" };
}
@using WebApplication1.App_Code

@Html.CreateList(students)
```

Встроенные html-хелперы

Встроенный хелпер **BeginForm** позволяет создать форму

Принимает в качестве параметров имя метода действия и имя контроллера, а также тип запроса

```
@using(Html.BeginForm("Buy","Home",FormMethod.Post))
        {
            ...
        }
```

Хелпер Html.**TextBox** генерирует тег **input** со значением атрибута type равным text. Перегружен 7 раз.

```
@Html.TextBox("Address")
```

Html.Hidden генерирует тег input type="hidden"

```
@Html.Hidden("ProductId",(int) ViewBag.ProductId)
```

Один из параметров хелперов позволяет задать атрибуты создаваемого элемента формы, в том числе и классы css.

```
@Html.TextBox("Text", "Наберите текст комментария",
                        new { style = "width:350px;height:100px",
                            maxlength = 250, minlengh = 5, required
= true })
```

Этот тег генерирует тег:

```
<input id="Text" maxlength="250" minlengh="5" name="Text"
required="True" style="width:350px;height:100px" type="text"
value="&#x41D;&#x430;&#x431 … " />
```

Html.Password
Html.RadioButton
Html.CheckBox
Html.Label
Html.DropDownList
Html.ListBox

Строго типизированные хелперы

Принимают в качестве параметра лямбда-выражение, в котором указывается то свойство модели, к которому должен быть привязан данный хелпер.

Могут использоваться только в строго типизированных представлениях

Тип модели, которая передается в хелпер, должен быть тем же самым, что указан в директиве **@model**.

Для каждого базового встроенного хелпера имеется свой строго типизированный хелпер, только в имени добавлен суффикс **For.**

```
@using MvcSimple.Models
@model Purchase
    <div>
        @using(Html.BeginForm("Buy","Home",FormMethod.Post))
        {
          @Html.HiddenFor(m=>m.ProductId,(int) ViewBag.ProductId)
            <table>
                <tr>
                    <td><p>Введите  имя </p></td>
                    <td>@Html.TextBoxFor(m=>m.Client)</td>
                </tr>
                <tr>
                    <td><p>Введите адрес :</p></td>
                       <td>@Html.TextBoxFor(m => m.Address)</td>
                </tr>
            <tr><td><input type="submit" value="Купить" /></td><tr>
            </table>
      }
    </div>
```

Шаблонные хелперы

Не генерируют *определенный* элемент html.
Генерируют тот элемент html, который наиболее подходит данному свойству, исходя из его типа и метаданных.

Display (DisplayFor)

Создает элемент разметки для отображения значения указанного свойства модели

Html.Display("Name") (Html.DisplayFor(m => m.Name))

Editor

Создает элемент разметки для редактирования указанного свойства модели: *Html.Editor("Name")*

DisplayForModel

Создает поля для чтения для всех свойств модели:
Html.DisplayForModel()

EditorForModel

Создает поля для редактирования для всех свойств модели:
Html.EditorForModel()

Для хелпера можно задавать в качестве параметра имя **шаблона** для отображения. Для этого в папке с представлениями нужно создать подпапку **DisplayTemplates**. В эту папку нужно добавить частичное представление с моделью, тип которой совпадает с типом параметра в шаблонном хелпере.

Например, создадим шаблон для отображения списка строк

```
@* Шаблон для списка строк *@
@model List<String>

<ul>
    @foreach (var item in Model)
    {
        <li>@item</li>
    }
</ul>
```

Тогда имя этого шаблона можно указывать в хелпере:

```
@Html.DisplayFor(m => m.Readers, "LisstTemplate")
```

Также шаблон можно указать прямо в определении модели, используя атрибут **UIHint**

```
[UIHint("LisstTemplate")]
    public List<string> Readers { get; set; }
```

Хелпер **ActionLink** создает гиперссылку на действие контроллера. Если создается ссылка на действие, определенное в том же контроллере, можно указать только имя действия.

Если нужно создать ссылку на действие из другого контроллера, то в хелпере ActionLink в качестве третьего аргумента указывается имя контроллера.

Параметры для действия передаются через параметр routeValues (можно в виде объекта анонимного класса).

Также можно установить значения атрибутов элемента HTML через параметр htmlAttributes.

```
@Html.ActionLink("Перейти к комментариям", "Comments",
            "Home", new { header = Model.Title },
            new { @class = "btn btn-primary btn-large" })
```

Tag-хелперы

Tag-хелперы – это специальные элементы в представлении, выглядят как обычные html-элементы или атрибуты, но при работе приложения они обрабатываются движком Razor на стороне сервера и преобразуются в стандартные html-элементы. Например,

```
<a asp-controller="Home" asp-action="Comments"
                asp-route-header="@Model.Title"
                class="btn btn-primary btn-large">
                 Перейти к комментариям
            </a>
```

ScriptTagHelper применяется для подключения внешних файлов скриптов.

Атрибуты:

- **asp-append-version**: если true, то к пути к файлу скрипта добавляется номер версии
- **asp-fallback-src**: указывает вспомогательный путь к скрипту, если загрузка скрипта, указанного в атрибуте src пройдет неудачно
- **asp-fallback-test**: определяет выражение, которое тестирует загрузку основного скрипта из атрибута src
- **asp-src-include**: определяет шаблон подключаемых файлов, через запятую можно задать несколько шаблонов
- **asp-src-exclude**: определяет через запятую набор шаблонов для тех файлов, которые следует исключить из загрузки
- **asp-fallback-src-include**: определяет набор шаблонов файлов, которые подключаются, если загрузка основного скрипта из атрибута src прошла неудачно
- **asp-fallback-src-exclude**: определяет набор шаблонов файлов, которые следует исключить из загрузки, если загрузка основного скрипта из атрибута src прошла неудачно

Символы подстановки в шаблонах :

? любой одиночный символ кроме слеша.
js/script?.js будет соответствовать **js/script1.js** или **js/scriptX.js**, но не **js/script35.js**

***** любое количество символов кроме слеша.
js/*.js будет соответствовать **js/script.js** или **js/scriptX25.js**, но не **js/bootstrap/script.js**

****** любое количество символов, в том числе и слеш.
js//script.js** будет соответствовать **js/script.js** или **js/bootstrap/script.js**, но не **js/script35.js**

```
<script
src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.17.0/jquery.validate.min.js"
 asp-fallback-src="~/lib/jquery-validation/dist/jquery.validate.min.js"
 asp-fallback-test="window.jQuery && window.jQuery.validator">
```

LinkTagHelper определяет тег **link**, который используется для подключения файлов стилей.

Атрибуты:

- **asp-append-version**: если true, то к пути к файлу скрипта добавляется номер версии
- **asp-fallback-href**: указывает вспомогательный путь к файлу стиля, если загрузка файла, указанного в атрибуте href пройдет неудачно
- **asp-href-include**: определяет шаблон подключаемых файлов, через запятую можно задать несколько шаблонов
- **asp-href-exclude**: определяет через запятую набор шаблонов для тех файлов, которые следует исключить из загрузки
- **asp-fallback-href-include**: определяет набор шаблонов файлов, которые подключаются, если загрузка основного файла стиля прошла неудачно
- **asp-fallback-href-exclude**: определяет набор шаблонов файлов, которые следует исключить из загрузки, если загрузка основного файла стиля прошла неудачно

```
<link rel="stylesheet"
href=https://stackpath.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css
  asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
  asp-fallback-test-class="sr-only"
  asp-fallback-test-property="position"
  asp-fallback-test-value="absolute" />
```

Tag-хелперы форм

```
<form asp-action="Order" asp-antiforgery="true">
    …
</form>
```

Основные атрибуты:

- **asp-controller**: указывает на контроллер, которому предназначен запрос
- **asp-action**: указывает на действие контроллера
- **asp-antiforgery**: если имеет значение true, то для этой формы будет генерироваться antiforgery token

- **asp-route**: указывает на название маршрута
- **asp-all-route-data**: устанавливает набор значений для параметров
- **asp-route-[название параметра]**: определяет значение для определенного параметра
- **asp-page**: указывает на страницу RazorPage, которая будет обрабатывать запрос

В тегах

LabelTagHelper
InputTagHelper
SelectTagHelper
TextAreaTagHelper

атрибут **asp-for** указывает, для какого свойства модели создается элемент.

```
<label asp-for="Address" class="control-label"/>
<input asp-for="Address" class="form-control" />
```

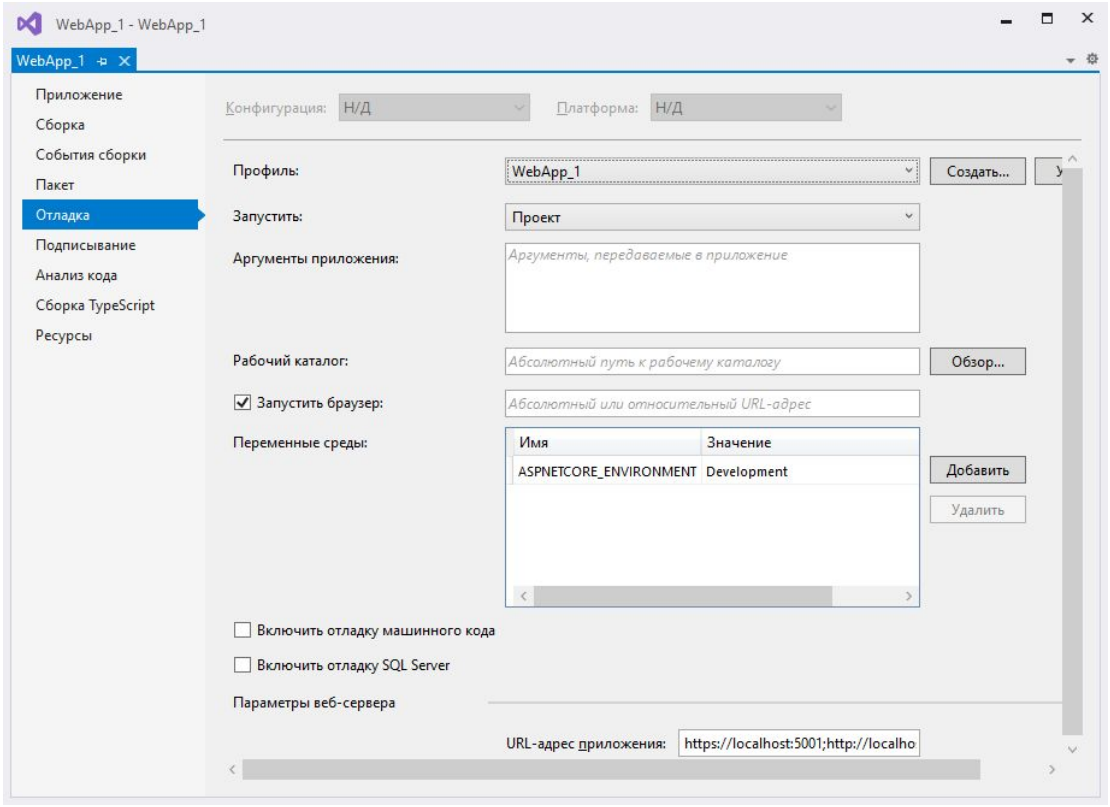
EnvironmentTagHelper используется для генерации определенной разметки html в зависимости *от состояния приложения*: приложение находится в процессе разработки, тестирования или опубликовано на сервере.

Состояние проекта задается с помощью среды окружения **ASPNETCORE_ENVIRONMENT**. Часто данный тег-хелпер используется совместно с LinkTagHelper и ScriptTagHelper.

Атрибут **names** позволяет установить названия состояний среды, при которых применяется данный тег.

```
<environment include="Development">
        <link rel="stylesheet" href="~/css/site.min.css" />
</environment>

<environment exclude="Development">
     <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.
min.css"
        asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
        asp-fallback-test-class="sr-only"
    asp-fallback-test-property="position"
    asp-fallback-test-value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css"
    asp-append-version="true" />
</environment>
```


WebApp_1 - WebApp_1
WebApp_1
Приложение
Сборка
События сборки
Пакет
Отладка
Подписывание
Анализ кода
Сборка TypeScript
Ресурсы
Конфигурация: Н/Д
Платформа: Н/Д
Профиль:
WebApp_1
Создать...
Запустить:
Проект
Аргументы приложения:
Аргументы, передаваемые в приложение
Рабочий каталог:
Абсолютный путь к рабочему каталогу
Обзор...
Запустить браузер:
Абсолютный или относительный URL-адрес
Переменные среды:
Имя
Значение
ASPNETCORE_ENVIRONMENT
Development
Добавить
Удалить
Включить отладку машинного кода
Включить отладку SQL Server
Параметры веб-сервера
URL-адрес приложения:
https://localhost:5001;http://localho

Создание tag-хелперов

https://dotnet.today/ru/aspnet5-vnext/mvc/views/tag-helpers/authoring.html

Для создания тег-хелпера нужно создать класс, наследующий от **TagHelper**.

В классе TagHelper определен метод **Process**(), который переопределяется производными классами для реализации поведения, трансформирующего элементы. Имя дескрипторного вспомогательного класса (тег-хелпера) образуется из имени трансформируемого элемента и суффикса TagHelper .

Метод Process принимает два параметра: объект **TagHelperContext**, представляющий контекст тега (его содержимое, атрибуты), и объект **TagHelperOutput**, отвечающий за генерацию выходного элемента html на основе тега.

Свойства класса TagHelperContext:

- **AllAttributes** возвращает словарь (только для чтения) с атрибутами, примененными к элементу, подвергающемуся трансформации, который индексирован по имени и по числовому индексу
- **Items** возвращает словарь, который используется для согласования дескрипторных вспомогательных классов
- **Uniqueid** возвращает уникальный идентификатор для трансформируемого элемента

Свойства и метод класса TagHelperOutput

- **TagName** применяется для получения и установки имени дескриптора в выходном элементе
- **Attributes** возвращает словарь, содержащий атрибуты для выходного элемента
- **Content** возвращает объект TagHelperContent, который используется для установки содержимого элемента

TagMode указывает, как будет записываться выходной элемент, с применением значения из перечисления TagMode.

```
public class ListTagHelper:TagHelper
    {
            public List<String> Info { get; set; }
        public override void Process(TagHelperContext context,
TagHelperOutput output)
        {
            output.TagName = "ul";
            string temp = "";
            foreach (string item in Info)
            {
                temp = $"{temp}<li>{item}</li>";
            }
            output.Content.SetHtmlContent(temp);
            output.TagMode = TagMode.StartTagAndEndTag;
        }
    }
```

Чтобы задействовать класс хелпера в представлении, нам надо подключить его функциональность в представление. Например, использовать директиву addTagHelper в файле *Views/_ViewImports.cshtml*:

```
@addTagHelper "*, WebApp_1"
```

Применение:

```
<list info="@Model.Readers"></list>
<list info='new List<string>() { "Вася","Петя","Коля"}'></list>
```

В следующем примере создается tag-хелпер для создания атрибута.

```
[HtmlTargetElement("a")]
    public class ButtonTagHelper:TagHelper
    {
        public string BsButtonColor { get; set; }
        public override void Process(TagHelperContext context,
                                              TagHelperOutput output)
        {
   output.Attributes.SetAttribute("class",
                $"btn  btn-large btn-{BsButtonColor}");
        }
    }
```

Применение:

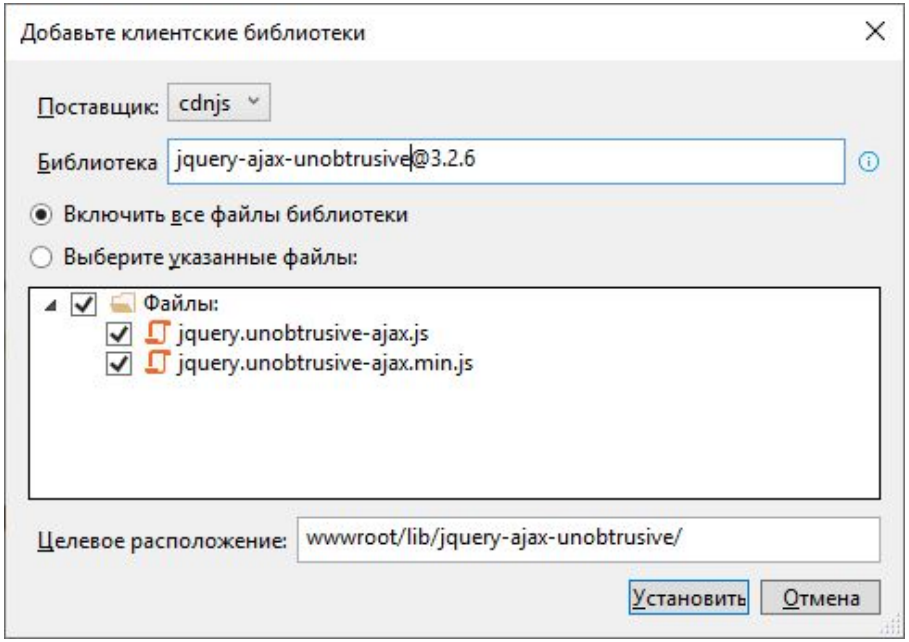
```
      <a href="/Home/Index" bs-button-color="primary">
                              На главную &raquo;
       </a>
```

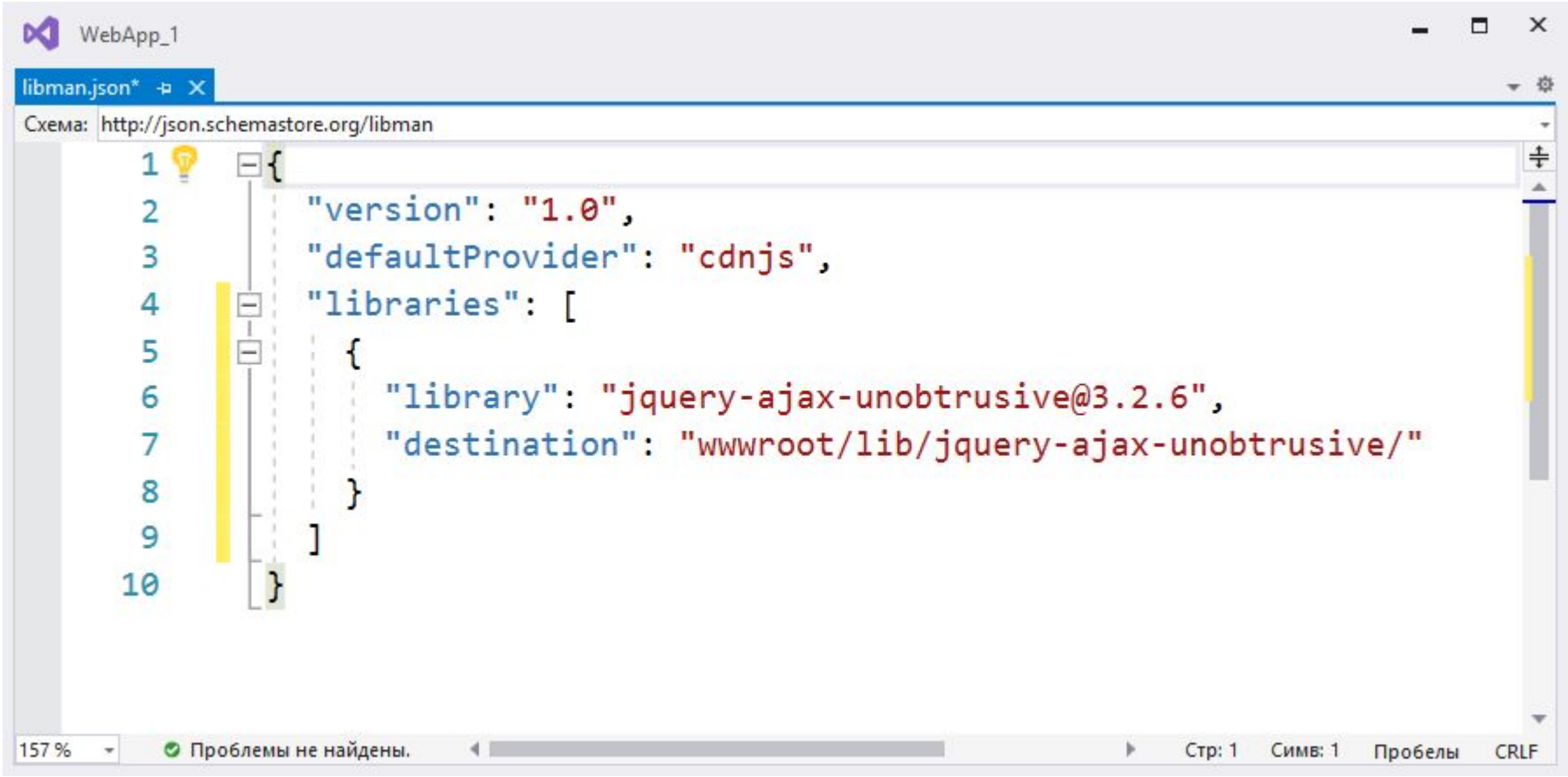
```
<a asp-controller="Home" asp-action="Comments"
                asp-route-header="@Model.Title"
                bs-button-color="danger">
                 Перейти к комментариям
</a>
```

Частичные представления

Частичные представления можно встраивать в другие представления. Они часто используются для рендеринга результатов AJAX-запроса.

Для того, чтобы использовать **ajax**, нужно добавить в проект библиотеку **jquery-ajax-unobtrusive**






```
<script
     src="~/lib/jquery-ajax-unobtrusive/jquery.unobtrusive-ajax.js">
</script>
```

Если есть частичное представление **GetComments**

```
@model List<Comment>
@foreach (var comment in Model)
{
    <div class="row">
        <div class="col-md-4">
            <p>@comment.Nik</p>
            <p>@comment.Time</p>
        </div>
        <div class="col-md-8">
            <p>@comment.Text</p>
        </div>

    </div>
}
```

его можно встроить в представление с помощью вызова метода Html.PartialAsync()

```
@await Html.PartialAsync("GetComments",Model.Comments)
```

За рендеринг частичных представлений отвечает объект **PartialViewResult**, который возвращается методом **PartialView**.

```
<form asp-controller="Home"
      asp-action="InsertComments"
      method="post"
      data-ajax="true"
      data-ajax-update="#com"
      data-ajax-mode="replace"
      data-ajax-method="post">
  .....
</form>
```

```
[HttpPost]
        [ActionName("InsertComments")]
        public ActionResult AddComments(string title)
        {

                Comment comment = new Comment();
comment.Nik = Request.Form.FirstOrDefault(p=>p.Key=="Nik").Value;
                comment.Text = Request.Form["Text"];
                comment.Time = DateTime.Now;
                CommentsDb.Comments.Add(comment);
        return RedirectToAction("GetComments");
        }
```

```
public ActionResult GetComments()
        {
            return PartialView(CommentsDb.Comments);
        }
```

Валидация на стороне клиента

Для валидации модели на стороне клиента нужно добавить скрипты

```
<script src=
"~/lib/jquery-validation/dist/jquery.validate.min.js">
</script>
<script
src=
"~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min
.js">
</script>
```

Это уже есть в файле **_ValidationScriptsPartial.cshtml** Поэтому вместо прямого подключения скриптов можно подключать данное частичное представление. *Перед подключением скриптов валидации должно идти подключение jquery.*

При объявлении модели нужно использовать атрибуты валидации.

В представлении нужно использовать tag-хелперы валидации. При каждом свойстве нужно использовать хелпер валидации (**ValidationMessageTagHelper**)

****

```
public class LoginModel
    {
        [Required]
        [Display(Name = "Логин")]
        public string UserName { get; set; }

        [Required]
        [DataType(DataType.Password)]
        [Display(Name = "Пароль")]
        public string Password { get; set; }
        [Display(Name = "Запомнить?")]
        public bool RememberMe { get; set; }

        public string ReturnUrl { get; set; }
    }
```

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
 <div class="form-group">
       <label asp-for="UserName" class="control-label"></label>
       <input asp-for="UserName" class="form-control" />
       <span asp-validation-for="UserName" class="text-danger"></span>
  </div>

  <div class="form-group">
        <label asp-for="Password" class="control-label"></label>
        <input asp-for="Password" class="form-control" />
       <span asp-validation-for="Password" class="text-danger"></span>
  </div>
```

Для отображения сводки ошибок валидации применяется **ValidationSummaryTagHelper**. Он применяется к элементу **div** в виде атрибута **asp-validation-summary**

В качестве значения атрибут asp-validation-summary принимает одно из значений перечисления **ValidationSummary**:

- **None**: ошибки валидации не отображаются
- **ModelOnly**: отображаются только ошибка валидации уровня модели, ошибки валидации для отдельных свойств не отображаются
- **All**: отображаются все ошибки валидации

Хелпер валидации **Html.ValidationMessageFor** для свойств **модели**

```
@Html.EditorFor(m => m.Client)

@Html.ValidationMessageFor(m => m.Client)
```

Для отображения сводки сообщений об ошибках при валидации модели можно использовать хелпер **Html.ValidationSummary**

Атрибуты:

Required

StringLength

RegularExpression

```
[RegularExpression (@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
                    ErrorMessage = "Некорректный адрес")]
public string Email { get; set; }
```

Range

```
[Range(1700,2000,ErrorMessage="Недопустимый год")]
public int Year { get; set; }

[Range(typeof(decimal), "0.00", "49.99")]
public decimal Price { get; set; }
```

Compare

```
[DataType(DataType.Password)]
public string Password { get; set; }

[Compare("Password",ErrorMessage="Пароли не совпадают")]
[DataType(DataType.Password)]
public  string PasswordConfirm { get; set; }
```

Для проверки на корректность электронной почты определены специальные атрибуты:

[CreditCard]

[EmailAddress]

[Phone]

[Url]

Remote

Для валидации свойства выполняет запрос на сервер к определенному методу контроллера. И если требуемый метод контроллера вернет значение false, то валидация не пройдена. Например:

```
[Remote(action: "CheckEmail", controller: "Home",
                    ErrorMessage ="Email уже используется")]
public string Email { get; set; }
```

```
[AcceptVerbs("Get", "Post")]
    public IActionResult CheckEmail(string email)
    {
        if (email == "admin@mail.ru" || email == "aaa@gmail.com")
            return Json(false);
        return Json(true);
    }
```

Все атрибуты валидации образованы от базового класса **ValidationAttribute**, который находится в пространстве имен **System.ComponentModel.DataAnnotations**

Можно создавать свои атрибуты для свойств и для модели.

```
public class PurschaseAllowedAttribute:ValidationAttribute
   {
       public override bool IsValid(object value)
       {
           Purchase p = value as Purchase;
           if (p.Client == "Вася" && p.Address.Contains("Гомель"))
           {
               return false;
           }
           return true;
       }
   }
```

```
[PurschaseAllowed(ErrorMessage = "Вася из Гомеля покупать не может")]
    public class Purchase
    {
        // ID покупки
        public int Id { get; set; }
        // имя и фамилия покупателя
   [StringLength(50, MinimumLength = 3, ErrorMessage = "Мало букв")]
          [Required(ErrorMessage = "Введите свое имя")]

        public string Client { get; set; }
        // адрес покупателя
        public string Address { get; set; }
        // ID книги
        public int ProductId { get; set; }
        // дата покупки
        public DateTime Date { get; set; }
    }
```

Валидация на стороне сервера

Если в браузере отключен javascript , то сообщение с ошибками не отображается, а форма отправляется на сервер.

Узнать, проходит модель валидацию или нет, можно с помощью свойства **ModelState.IsValid**

```
[HttpPost]
  public ActionResult Buy(Purchase purchase)
  {
      purchase.Date = DateTime.Now;
      if (ModelState.IsValid)
      {
          db.Purchases.Add(purchase);
          db.SaveChanges();
          return new SubmitResult(" Спасибо за покупку ",
           "Info?ProductId=" + purchase.ProductId);
      }
       return new SubmitResult(" Ошибка ", "Index" );
  }
```

Объект **ModelState** сохраняет все значения, которые пользователь ввел для свойств модели, а также все ошибки, связанные с каждым свойством и с моделью в целом.

Метод **ModelState.AddModelError** добавляет для свойства, указанного в качестве первого параметра, ошибку, сообщение для которой указано в качестве второго параметра.

```
if (purchase.Client == "Вася" && purchase.Address.Length < 3)
 ModelState.AddModelError("", "Вася с коротким адресом!");//ошибки
                                                          //на уровне модели
```

Html.ValidationSummary()	Отображает общий список ошибок сверху
Html.ValidationSummary(bool)	Если параметр равен **true**, то вверху будут отображаться только сообщения об ошибках уровня модели, а специфические ошибки будут отображаться рядом с полями ввода. Если же параметр равен **false**, то вверху отображаются все ошибки.
Html.ValidationSummary(string)	Данная перегруженная версия хелпера отображает перед списком ошибок сообщение, которое передается в параметр string
Html.ValidationSummary(bool, string)	Сочетает две предыдущие перегруженные версии

Внедрение зависимостей в контроллер

Это можно делать следующими способами:

- Через конструктор
- Через параметр метода, к которому применяется атрибут FromServices
- Через свойство HttpContext.RequestServices

Добавление новых классов в контейнер происходит в методе ***ConfigureServices()*** класса ***Startup***. Для этого у *IServiceCollection* существует метод ***Add(ServiceDescriptor serviceDescriptor)*.**

Класс *ServiceDescriptor* описывает параметры зависимости и содержит следующие свойства:

- ***ServiceLifetime Lifetime*** – Время жизни экземпляра класса:

- *Transient* – Новый экземпляр будет создан при каждом обращении к контейнеру.
- *Scoped* – Новый экземпляр будет создан для каждого запроса к серверу. При этом в рамках самого запроса всегда будет передаваться один и тоже экземпляр.
- *Singleton* – Всегда будет использоваться один и тоже экземпляр данного типа

- *Type* ***ServiceType*** – Тип регистрируемого сервиса, как правило интерфейса.

- Один из вариантов, указывающих на способ создания реализации *ServiceType:*

  Type ***ImplementationType*** – Тип реализации. Будет создан в контейнере при обращении.

  object ***ImplementationInstance*** – Созданный заранее объект типа *ServiceType* (только для *Singleton*);

  public Func<IServiceProvider, object> ***ImplementationFactory*** – Фабричный метод для создания

Для более компактной записи существуют методы-расширения с различными перегрузками для разных вариантов сочетания описанных выше значений:

AddSingleton(…)
AddScoped(…)
AddTransient(…)