**Виды маршрутизации**

* **Маршрутизация на основе условностей / соглашений** — это подход, при котором маршруты определяются в конфигурации приложения на основе заранее установленных правил. MapControllerRoute.
* **Маршрутизация на основе атрибутов** позволяет разработчикам определять маршруты непосредственно в контроллерах и методах действий с использованием атрибутов.
* **Таблица маршрутов (routing table)** — это набор правил, которые определяют, как запросы URL-адресов сопоставляются с действиями контроллеров.

**Маршрутизация** — это процесс сопоставления URL-запросов с определёнными контроллерами и действиями (методами) в приложении.

**Как работает маршрутизация**

При запуске приложения ASP.NET Core регистрируются маршруты и настраивается конвейер обработки запросов в классе Startup.

Когда клиент отправляет HTTP-запрос, запрос проходит через конвейер промежуточного ПО.

Как только запрос поступает в приложение, Routing Middleware анализирует входящий URL и пытается сопоставить его с маршрутами в приложении.

Если совпадение найдено, извлекаются параметры из URL и сохраняются в контексте запроса.

После успешного сопоставления маршрута запрос передаётся в промежуточное ПО конечных точек. UseEndpoints определяет, какой обработчик должен быть вызван для обработки запроса.

Поскольку обработчик был определён для данного маршрута, он вызывается с передачей извлечённых параметров из URL и выполняет свою логику, обрабатывая запрос.

После завершения выполнения обработчика формируется ответ для клиента, который помещается в контекст HTTP-ответа.

Как только ответ сформирован, он отправляется обратно клиенту через HTTP, завершая процесс обработки запроса.

**У кого больше приоритет среди MapControllerRoute**

* Маршруты, определенные с помощью атрибутов в контроллерах и методах действий, будут обрабатываться раньше определенных с помощью обычной маршрутизации.
* Если два маршрута имеют одинаковую относительную спецификацию (например, оба определяют один и тот же путь), тот, который был определен первым, будет иметь более высокий приоритет.
* При наличии маршрутов, использующих параметры, маршруты с более конкретными или точными параметрами имеют приоритет перед более общими.

**У кого больше приоритет среди атрибутов Route**

* Более конкретные маршруты имеют более высокий приоритет.
* Маршруты, определенные первыми, имеют более высокий приоритет при равенстве спецификаций.
* Определяющие методы HTTP, влияют на приоритет; метод, соответствующий запросу, имеет приоритет. Если запрос поступает с определенным методом HTTP, ASP.NET Core будет искать маршруты, соответствующие этому методу. Например, если запрос отправлен с методом GET, маршруты, помеченные атрибутом [HttpGet], будут обрабатываться, а маршруты с атрибутами [HttpPost] или другими методами игнорируются.

**Порядок, в котором роуты добавляются в таблицу маршрутизации, определяет порядок, в котором они будут найдены при поиске соответствий**. Это значит, что роуты в исходном коде должны быть перечислены по приоритету: от высшего, с наиболее конкретными условиями, до самого низкого, или *catchall* роута.

Сейчас мы добавим в пример приложения *catchall* роут, предназначенный для URL, которые не соответствует никакому другому роуту. Цель этого роута - показать сообщение об ошибке HTTP 404. Глобальные *catchall* роуты будут соответствовать любому роуту, и они должны быть определены в *последнюю очередь*:

Значение catchall заменяет значение, которое было передано в *catchall* роут. В отличие от обычных параметров роутов, *catchall* параметры (с префиксом звездочкой) охватывают целую часть URL, включая слеши, которые обычно используются для разделения параметров роута.

**Как определяется порядок маршрутов с атрибутом Route:**

1. Литеральные сегменты – это части маршрута, которые являются статическими и явно указаны в URL.
2. Параметры маршрута с ограничениями.
3. Параметры маршрута без ограничений.
4. Сегменты параметров с catch all с ограничениями.
5. Сегменты параметров с catch all без ограничений.

Маршруты атрибутов могут настроить порядок с помощью [Order](https://learn.microsoft.com/ru-ru/dotnet/api/microsoft.aspnetcore.mvc.routeattribute.order" \l "microsoft-aspnetcore-mvc-routeattribute-order) свойства.



В данном приложении контроллер TMResearchController с методами, которые соответствуют определённым URI (адресам) и возвращают заранее заданные ответы. Приложение настраивает маршрутизацию так, чтобы каждый запрос к определённому URI попадал в нужный метод контроллера.

**Основные аспекты работы приложения:**

**Контроллер** принимает HTTP-запросы (GET) на URI, указанные в таблице. Каждый метод формирует и возвращает текстовый ответ (строку), который отображается в браузере или используется клиентом.

**Маршрутизация:** Приложение связывает URI (например, /MResearch/M01) с соответствующими методами контроллера. Для этого используются атрибуты [HttpGet] в методах контроллера.

Общий маршрут GET:MXX настроен как fallback (резервный), чтобы обработать любой другой URI, не указанный явно.

**Ответы от сервера:**

Каждый метод возвращает сообщение, в котором указывается имя метода (например, GET:M01) и дополнительные данные (string - ..., если строка передана в URI).

**Как работает приложение?**

1. Пользователь отправляет HTTP-запрос:

Через браузер, Postman или другой клиент пользователь отправляет запрос на URI, например, /MResearch/M01.

1. Приложение обрабатывает запрос:

В зависимости от URI маршрутизация выбирает подходящий метод в контроллере TMResearchController.

1. Контроллер формирует ответ:

Например, запрос /MResearch/M01/1 обрабатывается методом M01, который возвращает текст GET:M01\nstring - null.

1. Ответ отправляется клиенту:

Ответ отображается в браузере, Postman или любом другом клиенте, который отправлял запрос.

### 5 Лабораторная

**В ASP.NET MVC общий макет (layout)** — это шаблон, который используется для создания общего интерфейса и структуры страниц приложения.

Главное же отличие от обычных представлений состоит в использовании метода **@RenderBody()**, который является плейсхолдером и на место которого потом будут подставляться другие представления, использующие данную мастер-страницу.

то есть если я хочу сделать общий макет с шапкой и футером. То я создаю представление с любым названием, закидываю в него код шапки и футера. Между ними вставляю @RenderBody(). Потом в обычном представлении прописываю @{ Layout = "~/Views/...путь\_к\_макету"; }.

Если в представлении не указать свойство Layout, то б**удет использован макет по умолчанию**, если он задан в настройках приложения.

В ASP.NET MVC можно указать глобальный макет в файле ViewStart.cshtml, который обычно находится в папке Views. Этот файл применяется ко всем представлениям, которые не задают собственный макет.

\_**ViewStart.cshtml** используется для настройки представлений перед их рендерингом, например, для указания общего макета (Layout) и задания глобальных параметров (например, ViewBag или ViewData). Этот файл выполняется **перед каждым представлением**, а его код интерпретируется и выполняется при каждом запросе.

**\_ViewImports.cshtml** управляет **импортом пространств имен и тег-хелперов** и действует как статический импорт. Пространства имен, добавленные через \_ViewImports.cshtml, доступны **сразу для всех представлений в папке и подпапках** без необходимости их подключать заново.

Кроме обычных представлений и мастер-страниц в ASP.NET Core MVC также можно использовать частичные представления или **partial views**. Их отличительной особенностью является то, что их можно встраивать в другие обычные представления.

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

Метод **Html.PartialAsync()** встраивает код частичного представления в обычное. Он является асинхронным и возвращает объект IHtmlContent, который представляет html-содержимое и который обернут в объект **Task<TResult>**.

Кроме метода Html.PartialAsync() частичное представление можно встроить с помощью другого метода - **Html.RenderPartialAsync**. Этот метод также принимает имя представления, только он используется не в строчных выражениях кода Razor, а в блоке кода, то есть обрамляется фигурными скобками.

В частичных представлениях обычно **не указывается Layout**, так как они встроены в родительское представление и должны следовать его структуре.

Если в частичном представлении прописать Layout, это может вызвать конфликт, так как оно начнет пытаться рендериться как самостоятельная страница, а не как часть другой страницы.

Html.BeginForm создает форму. Она принимает параметры:

* action — имя действия контроллера, к которому отправится форма при POST.
* "Calc" — имя контроллера, где расположен обработчик запроса.
* FormMethod.Post — указывает, что данные отправляются методом POST.

**Какова разница между частичным представлением и полным представлением?**

Полное представление — это представление, которое включает в себя всю HTML-структуру документа (теги <html>, <head>, <body> и т. д.) и отображает страницу как таковую.

Частичное представление — это только часть страницы, которая может быть вставлена в другие представления. Оно не содержит основной структуры HTML и используется для повторного использования разметки.

**Как вы можете передать данные в частичное представление?**

Данные могут быть переданы в частичное представление через:

Модель: передавая модель как параметр при вызове частичного представления.

ViewBag: используя динамическую структуру ViewBag для передачи данных.

ViewData: используя словарь ViewData для передачи данных.

**Html helper виды:**

HTML Helpers в ASP.NET MVC — это методы, которые упрощают создание HTML-кода в представлениях. Они позволяют разработчикам генерировать элементы HTML, такие как формы, текстовые поля, ссылки и многое другое, с минимальным количеством кода, а также позволяют удобно связывать эти элементы с данными модели.

**Основные виды HTML Helpers**

Стандартные HTML Helpers Эти помощники встроены в ASP.NET MVC и предназначены для создания стандартных HTML-элементов. Например:

Html.TextBoxFor(): Создает текстовое поле для привязанного свойства модели.

Html.TextAreaFor(): Создает многострочное текстовое поле.

Html.DropDownListFor(): Создает выпадающий список.

Html.CheckBoxFor(): Создает элемент типа checkbox.

Html.RadioButtonFor(): Создает элемент типа radio.

**HTML Helper для форм** Эти помощники предназначены для работы с формами. Они упрощают создание форм, включая открытие и закрытие форм, а также обработку элементов внутри них. Например:

* **Html.BeginForm():** Создает начало формы, принимает параметры, такие как имя действия и контроллера, а также метод (GET/POST).
* **Html.BeginRouteForm():** Создает форму на основе маршрута.

**Взаимодействие шаблонизаторов и HTML-хелперов**

Шаблоны и HTML-хелперы в ASP.NET MVC часто работают вместе. Например, @Html.DisplayFor() и @Html.EditorFor() представляют собой комбинацию шаблонов и HTML-хелперов: эти методы определяют, какой шаблон отображения или редактирования использовать для данных, при этом вызов DisplayFor() или EditorFor() также обрабатывается как HTML-хелпером, так и шаблоном.

Пример взаимодействия:

1. Вызов @Html.EditorFor(model => model.BirthDate) задействует встроенный HTML-хелпер EditorFor, который:
   * Сначала находит тип данных (DateTime).
   * Далее ищет подходящий шаблон для редактирования типа DateTime в Views/Shared/EditorTemplates.
2. Если пользователь создал шаблон DateTime.cshtml в EditorTemplates, то ASP.NET MVC использует именно его для рендеринга данных. В противном случае применяется стандартный шаблон.

**Пример: Создание пользовательского шаблона и хелпера**

1. **Создание шаблона**: Допустим, нужно создать шаблон для отображения денежного значения.
   * В Views/Shared/DisplayTemplates создается файл Currency.cshtml:

@model decimal

<span>@Model.ToString("C")</span> <!-- Форматирование в виде валюты -->

1. **Создание HTML-хелпера для отображения валюты**: Можно также создать хелпер для вставки валютного значения.

public static class HtmlHelperExtensions

{

public static MvcHtmlString DisplayCurrency(this HtmlHelper html, decimal amount)

{

string formattedAmount = string.Format("{0:C}", amount);

return MvcHtmlString.Create(formattedAmount);

}

}

Использование в представлении:

@Html.DisplayCurrency(123.45M)

В ASP.NET MVC действительно можно создавать кастомные шаблоны для сложных типов данных или компонентов, чтобы использовать их в различных частях приложения.

Допустим, у нас есть модель Product, которая содержит основные данные о продукте:

csharp

Копировать код

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public decimal Price { get; set; }

public string ImageUrl { get; set; }

}

Теперь создадим шаблон для отображения карточки продукта. Этот шаблон мы можем разместить в Views/Shared/DisplayTemplates/ProductCard.cshtml:

razor

@model Product

<div class="product-card">

<img src="@Model.ImageUrl" alt="@Model.Name" class="product-image" />

<h3 class="product-name">@Model.Name</h3>

<p class="product-description">@Model.Description</p>

<span class="product-price">@Model.Price.ToString("C")</span>

</div>

**Разбор кода:**

**Методы HTTP GET и POST для каждой операции** Для каждой арифметической операции (сложение, вычитание, умножение и деление) существуют два метода: один для обработки запросов GET (для отображения калькулятора для этой операции) и один для обработки запросов POST (для выполнения вычисления).

GET-метод для сложения -  **Атрибут [HttpGet("Calc/Sum")]**: Указывает, что этот метод будет вызываться для HTTP GET запросов по маршруту Calc/Sum.

 **ViewBag**: Это динамическая структура для передачи данных между контроллером и представлением. В данном случае, она используется для хранения информации о текущей операции и для того, какая кнопка была нажата.

POST-метод для сложения - Э  **Атрибут [HttpPost("Calc/Sum")]**: Указывает, что этот метод будет обрабатывать HTTP POST запросы по маршруту Calc/Sum, что происходит при отправке формы.

#### **Как это работает?**

1. **Начало блока формы**: Когда вызывается Html.BeginRouteForm, он создает HTML-код для открытия формы. Этот код будет включать <form> с атрибутами, указывающими на маршрут, метод отправки и другие параметры. Например, это может выглядеть так:

<form action="/Calc" method="post">

1. **Передача параметров**: Так как в routeValues передается значение press, в маршруте будет содержаться информация о том, какая кнопка была нажата (например, "+"). Это позволяет контроллеру идентифицировать, какая операция должна быть выполнена, когда форма отправляется.
2. **Вложенные элементы формы**: Внутри блока using вы добавляете элементы, которые должны быть частью формы:
   * @await Html.PartialAsync("\_Calc"): Вставляет частичное представление, которое, вероятно, содержит текстовые поля для ввода чисел.
   * <input type="text" disabled value="@result" placeholder="Результат" />: Создает поле для отображения результата операции, но оно отключено для ввода (пользователь не может изменять его).
   * <input type="submit" name="press" value="@press" />: Создает кнопку отправки формы с названием press, которая будет отправлять значение, соответствующее нажатой операции (например, "+").
3. **Отображение ошибок**: <p> @(ViewBag.Error as string) </p> отображает сообщение об ошибке, если оно было установлено в контроллере, например, если пользователь ввел некорректные данные.
4. **Закрытие формы**: После завершения блока using с Html.BeginRouteForm, форма автоматически закрывается. Это эквивалентно добавлению </form> в конце блока.

**Шаблонизаторы** — это механизмы, которые позволяют ASP.NET MVC автоматически определять, как отображать или редактировать данные на основе их типа. Основная идея шаблонизаторов — централизовать логику отображения и редактирования, избегая дублирования кода и поддерживая единый стиль представления данных.

**Виды шаблонизаторов**

1. **Встроенные шаблоны (Default Templates)**:  
   ASP.NET MVC предоставляет шаблоны для базовых типов данных, таких как string, DateTime, decimal, bool и другие. Когда используется метод @Html.DisplayFor() или @Html.EditorFor(), MVC автоматически применяет эти шаблоны на основе типа данных свойства.
2. **Display Templates (Шаблоны отображения)**:  
   Шаблоны отображения предназначены для read-only отображения данных. Они помогают создать единый стиль отображения, например, даты в формате dd-MM-yyyy или цен с валютными символами. Разработчик может создать свои шаблоны и сохранить их в папке Views/Shared/DisplayTemplates, чтобы MVC автоматически применяла их к нужным типам данных.
3. **Editor Templates (Шаблоны редактирования)**:  
   Шаблоны редактирования управляют тем, как данные рендерятся в интерфейсе ввода (например, формах). Они хранятся в Views/Shared/EditorTemplates и применяются, когда вызывается @Html.EditorFor(). Например, для типа DateTime можно создать шаблон с календарем для выбора даты, который будет автоматически применяться ко всем полям типа DateTime.
4. **Пользовательские шаблоны**:  
   ASP.NET MVC позволяет создавать кастомные шаблоны для сложных типов данных или кастомных компонентов, таких как карточки продуктов, панели навигации и т.д. Их можно переиспользовать в разных местах приложения, улучшая структурность кода и удобство работы.