**ASP.NET Core** — это кросс-платформенный фреймворк для разработки веб-приложений, обладающий высокой производительностью и поддерживающий различные сценарии, такие как веб-сайты, API, микросервисы и веб-приложения в реальном времени.

**MVC** — это архитектурный паттерн, который разделяет приложение на три части:

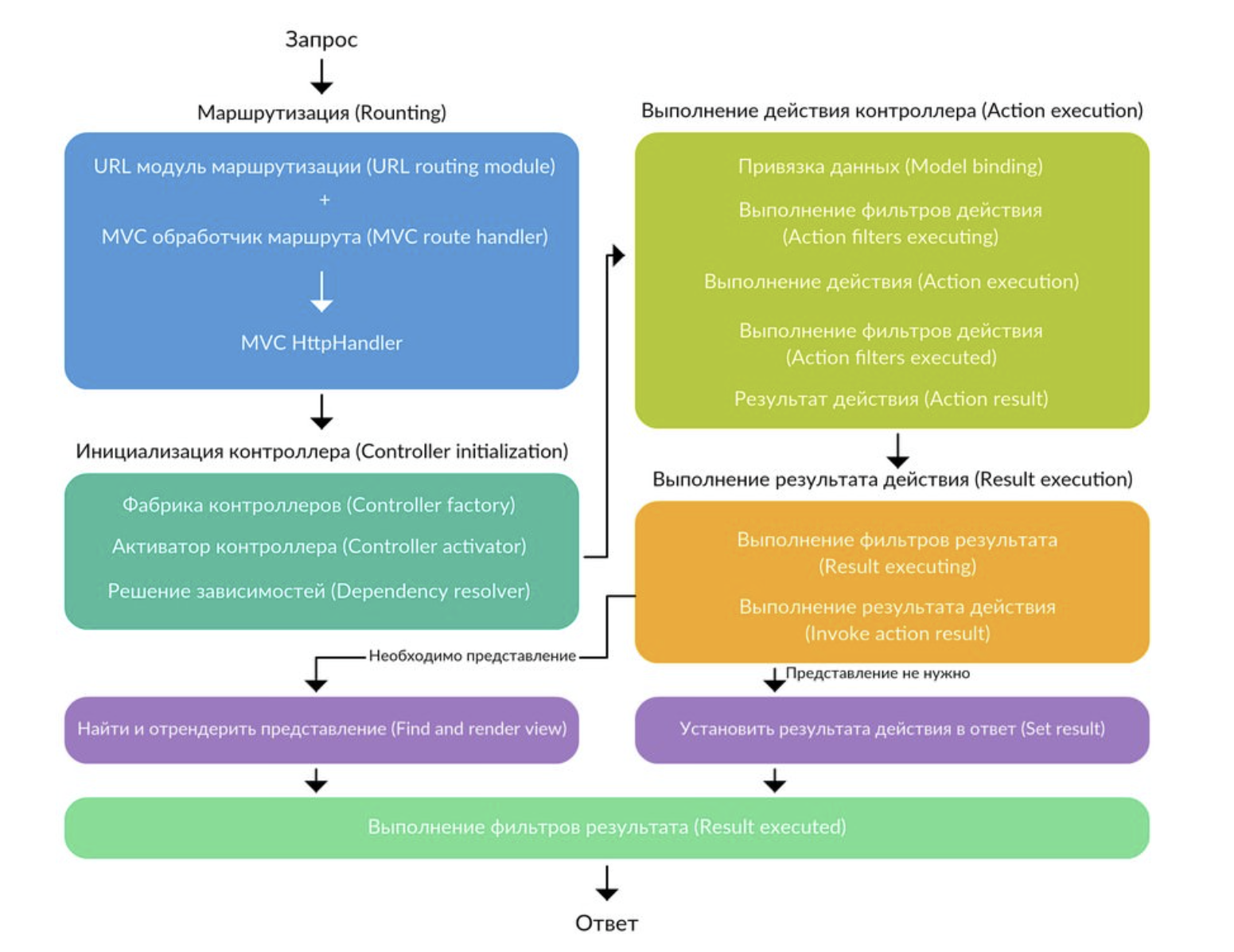
**Model (Модель)** — ответственна за управление данными. Это бизнес-логика приложения и взаимодействие с базой данных.

**View (Представление)** — это часть, которая отвечает за отображение данных пользователю. Представления обычно являются HTML-страницами, которые содержат динамически обновляемое содержимое.

**Controller (Контроллер)** — это компонент, который обрабатывает запросы пользователей, взаимодействует с моделью для получения данных и выбирает представление для отображения пользователю.

**MVC** помогает разделить логику представления, обработки данных и работы с пользователями, что улучшает модульность и тестируемость приложения.

**ASP.NET MVC жизненный цикл запроса**



**1.Routing (Маршрутизация)**

Первым делом запрос попадает в систему маршрутизации.

Система маршрутизации смотрит на URL и пытается понять:

* Какой контроллер (например, "HomeController") будет обрабатывать запрос?
* Какое действие этого контроллера (например, метод "About") нужно выполнить?

Иными словами, маршрутизация находит "маршрут" для запроса и передает его дальше.

Когда система маршрутизации определяет маршрут, запрос передается обработчику запроса (по сути, это программа, которая "берет запрос в руки"). Этот обработчик определяет, как с этим запросом дальше работать.

В обычном приложении ASP.NET MVC используется специальный обработчик под названием MvcRouteHandler. Его задача — передать управление следующему элементу, который называется HttpHandler.

HttpHandler — это важный элемент, он непосредственно занимается обработкой запроса и формирует для него ответ. Его ключевая задача — вызвать нужный контроллер и действие, чтобы выполнить нужную логику и вернуть данные.

Свойство IsReusable отвечает за то, нужно ли каждый раз создавать новый обработчик для каждого запроса или можно использовать один и тот же обработчик для нескольких запросов.

Метод ProcessRequest — это основной метод, который запускает обработку запроса. Он выполняет нужное действие и формирует ответ.

В стандартном ASP.NET MVC приложении ты получаешь специальный обработчик, который умеет работать с MVC архитектурой — это MvcRouteHandler. Он передает управление объекту MvcHandler, который запускает:

* Нужный контроллер (например, "HomeController").
* Внутри этого контроллера вызывается нужное действие (например, метод "About").

**2.Controller initialization (Инициализация контроллера)**

Когда запрос дошел до MvcHandler (это объект, который управляет обработкой запроса в MVC-приложении), он начинает свою работу. Главная его задача — найти нужный контроллер и передать ему управление для выполнения действия (метода). MvcHandler в методе ProcessRequest начинает процесс создания нужного контроллера и исполнение нужного действия.

Чтобы получить контроллер, MVC использует специальный объект — фабрику контроллеров (Controller Factory). Это нужно для того, чтобы правильно создать контроллер, который может зависеть от различных вещей (например, от данных или сервисов, которые нужно подтянуть для его работы).

По умолчанию, в ASP.NET MVC используется DefaultControllerFactory, который отвечает за создание контроллеров.

Объект, который реализует интерфейс IControllerFactory, отвечает за:

* Создание контроллеров: Когда нужно создать контроллер, фабрика отвечает за его создание.
* Уничтожение контроллеров: После завершения работы фабрика также может управлять тем, как контроллер "удаляется" из памяти.

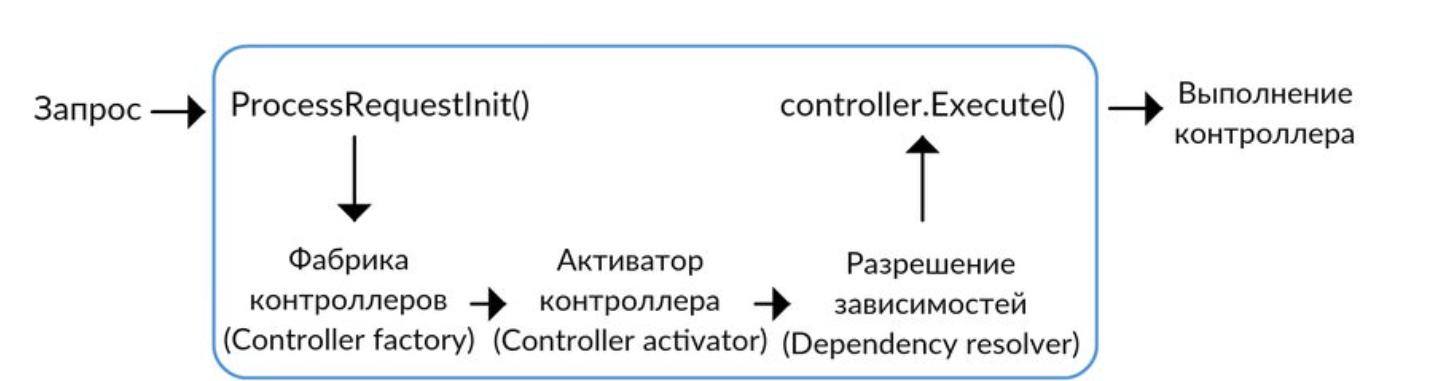
MVC использует еще один объект — активатор контроллеров (Controller Activator), который находит нужный контроллер и создает его.

Зачем это нужно? Потому что в сложных приложениях контроллеры могут зависеть от различных сервисов и данных, и для того чтобы "внедрить" эти зависимости, ASP.NET MVC использует механизм, который называется разрешение зависимостей (dependency resolver). Это делает процесс создания контроллеров гибким и расширяемым.

В ASP.NET MVC есть специальная система для того, чтобы управлять этими зависимостями — это dependency resolver. Он проверяет, какие объекты или данные нужны контроллеру, и автоматически "внедряет" их.

Когда нужный контроллер создан, MvcHandler передает ему управление. Он вызывает метод Execute у контроллера и передает ему всю информацию о запросе (в виде объекта RequestContext).

RequestContext — это объект, который содержит всю информацию о текущем запросе: URL, данные, параметры, маршруты и многое другое. Контроллер использует этот контекст для того, чтобы понять, что именно от него требуется, какие данные были переданы, и какое действие ему нужно выполнить.



**3.Action execution (Выполнение действия)**

При выполнении действий контроллера, так же выполняются фильтры.

Фильтры в ASP.NET MVC — это такие "точки расширения", которые позволяют добавить дополнительную логику перед или после выполнения действий контроллера. Они помогают выполнять какие-то общие задачи (например, проверку прав доступа или логирование), не изменяя основной код действий.

Когда в ASP.NET MVC вызывается метод контроллера, внутри все это обрабатывается через специальные классы и методы.

В классе контроллера (Controller), который мы пишем, реализовано поведение через наследование от базового класса ControllerBase. Этот базовый класс содержит метод Execute(), который отвечает за выполнение действий контроллера. Но он сам не делает всю работу — он вызывает другой метод ExecuteCore().

В методе ExecuteCore запускается выполнение действия, но уже через специальный объект ActionInvoker.

ActionInvoker — это компонент, который управляет всем процессом выполнения действия контроллера. Он отвечает за:

* Выполнение всех фильтров (аутентификации, авторизации и т.д.).
* Привязку параметров действия (например, данные, переданные через форму).
* Непосредственное выполнение действия.
* Формирование результата (например, HTML-страницы или JSON).

Когда MVC решает, какое действие контроллера нужно запустить, ActionInvoker обрабатывает запрос в несколько этапов:

* Фильтры аутентификации — сначала проверяется, является ли пользователь авторизованным (метод InvokeAuthenticationFilters).
* Фильтры авторизации — затем проверяются права доступа пользователя (метод InvokeAuthorizationFilters).
* Привязка параметров — данные, которые пользователь отправил (например, через форму или параметры URL), привязываются к параметрам действия (метод GetParameterValues).
* Выполнение действия с фильтрами — само действие контроллера выполняется с учетом всех фильтров действий (метод InvokeActionMethodWithFilters).
* Формирование результата — после выполнения действия система формирует результат (например, возвращает HTML-страницу) и обрабатывает фильтры результата (метод InvokeActionResultWithFilters).

Пример с HttpGet и HttpPost

ASP.NET MVC также позволяет выбирать действия в зависимости от типа запроса.

HttpGet — это обычный запрос (например, когда пользователь просто открывает страницу).

HttpPost — это запрос, который отправляется, когда пользователь отправляет данные (например, заполняет форму).

Эти типы запросов контролируются специальными атрибутами, такими как [HttpGet] и [HttpPost].

MVC выбирает действие, проверяя такие атрибуты с помощью метода IsValidForRequest. Например, если запрос типа POST, система найдет метод контроллера, помеченный атрибутом [HttpPost]. Если соответствующий метод не найден, будет ошибка.

**4. Result execution (Выполнение результата действия)**

После того, как действие контроллера завершилось, оно возвращает ActionResult. Это объект, который представляет результат действия, будь то HTML-страница, JSON-данные или что-то еще. Важно понять, что результат действия — это еще не готовый ответ. Это просто объект, который описывает, какой тип ответа нужно вернуть.

ASP.NET MVC поддерживает различные типы результатов, такие как:

ViewResult — возвращает HTML-страницу.

JsonResult — возвращает JSON-данные (удобно для API).

RedirectResult — делает перенаправление на другой URL.

Когда у нас уже есть результат действия (ActionResult), его нужно "превратить" в реальный ответ для клиента. Это делается через метод InvokeActionResultWithFilters, который выполняет результат с учетом всех фильтров результата. Этот метод:

* Рекурсивно применяет все фильтры результата (если они есть).
* Вызывает метод ExecuteResult, который и отвечает за непосредственное формирование ответа.

3. Что значит "выполняется сам результат"?

Выполнение результата — это процесс записи данных в HTTP-ответ. Например:

Если действие вернуло ViewResult, система находит нужное представление (HTML-страницу) и рендерит его в ответ.

Если это JsonResult, данные сериализуются в JSON и записываются в тело ответа.

Рассмотрим шаги для ViewResult:

Когда результатом действия является ViewResult (например, возврат HTML-страницы), вызывается метод ExecuteResult, который находится в базовом классе ViewResultBase. Он отвечает за то, чтобы найти представление и передать его на рендеринг.

ViewResult использует механизм поиска представлений через зарегистрированные ViewEngines. В процессе поиска MVC проходит по каждому ViewEngine (их может быть несколько), и первый, который находит подходящее представление, используется для рендеринга.

ViewEngine — это движок, который отвечает за нахождение и рендеринг представлений. По умолчанию в ASP.NET MVC используется RazorViewEngine, который ищет представления с расширениями .cshtml. Можно зарегистрировать свои движки (например, для поиска представлений в других директориях или с другими расширениями).

Когда нужное представление найдено, оно передается в движок рендеринга. В случае с Razor это шаблонный HTML-файл, который компилируется в HTML с учетом данных, переданных из контроллера (через модель).

После рендеринга HTML-страница (или результат другого типа) записывается в HTTP-ответ, который возвращается пользователю. Этот шаг включает отправку сгенерированного HTML в браузер, где пользователь видит результат.

Передача данных во view

В ASP.NET MVC существует несколько способов передачи данных из контроллера в представление (view). Данные, переданные во **view**, могут быть как простыми типами данных, так и сложными объектами. Основные способы передачи данных:

1. **Использование объекта модели (Model)**

Контроллер передает объект (или коллекцию объектов) модели в представление через метод View(). Модель может быть как простой (например, строка или число), так и сложной (класс, представляющий данные).

1. **Использование объекта ViewBag**

ViewBag — это динамическое свойство, которое позволяет передавать данные между контроллером и представлением. Его можно использовать для передачи простых данных без создания специальных классов для модели. Однако ViewBag не проверяется на этапе компиляции (из-за динамической природы), что делает его менее надежным.

Преимущество ViewBag заключается в простоте использования, но его следует использовать только для передачи небольших данных, таких как сообщения, заголовки и т.д.

1. **Использование объекта ViewData**

ViewData — это словарь типа Dictionary<string, object>, который также позволяет передавать данные между контроллером и представлением. В отличие от ViewBag, он не является динамическим, но предоставляет аналогичную функциональность.

ViewData удобен для передачи данных в виде ключ-значение, но требует приведения типов при использовании.

1. **Использование TempData**

TempData используется для передачи данных между запросами (например, при редиректе). В отличие от ViewBag и ViewData, данные в TempData сохраняются только на один запрос. Это полезно, когда нужно передать данные после выполнения редиректа (например, после сохранения формы).

TempData очищается после использования, поэтому данные доступны только в одном последующем запросе.

1. **Передача данных через параметры метода View()**



Контроллеры

Контроллеры — это один из основных компонентов архитектуры Model-View-Controller (MVC). Они играют ключевую роль в обработке запросов и являются связующим звеном между моделью (данными) и представлением (интерфейсом пользователя). Контроллеры получают входные данные от пользователя, обрабатывают их с помощью моделей и возвращают результат пользователю в виде представления или других форматов данных (например, JSON).

Основные функции контроллеров

* Получение запроса: Контроллер принимает входящие HTTP-запросы от пользователя (например, через браузер).
* Обработка данных: Контроллер взаимодействует с моделью для получения или изменения данных. Он может обращаться к базе данных, вызывать методы моделей или сервисов.
* Возвращение результата: Контроллер формирует ответ на запрос в виде одного из объектов ActionResult (HTML-страница, JSON-данные, перенаправление на другой URL и т.д.).

Контроллер в ASP.NET MVC — это обычный C# класс, который наследуется от базового класса Controller или ControllerBase. Стандартный контроллер содержит действия (методы), которые обрабатывают запросы. Каждое действие отвечает за конкретный тип запроса и возвращает результат.

Пример простого контроллера:

public class HomeController : Controller

{

// Действие для обработки запросов на главную страницу

public ActionResult Index()

{

// Возвращает представление Index.cshtml

return View();

}

// Действие для отображения страницы с информацией "О нас"

public ActionResult About()

{

ViewBag.Message = "Your application description page.";

return View();

}

// Действие для отображения страницы контактов

public ActionResult Contact()

{

ViewBag.Message = "Your contact page.";

return View();

}

}

Особенности:

Класс HomeController наследуется от Controller.

Каждое действие — это публичный метод, возвращающий ActionResult.

Действие Index() возвращает представление Index.cshtml.

Действие About() использует ViewBag для передачи данных в представление.

Способы маршрутизация

1. Традиционная маршрутизация (Conventional Routing)

Традиционная маршрутизация в ASP.NET MVC определяет правила маршрутизации в файле RouteConfig.cs (обычно в папке App\_Start). Эти правила задаются с помощью метода MapRoute, который указывает шаблон URL и сопоставляет его с контроллером, действием и параметрами.

public class RouteConfig

{

public static void RegisterRoutes(RouteCollection routes)

{

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

// Определение маршрута

routes.MapRoute(

name: "Default", // Имя маршрута

url: "{controller}/{action}/{id}", // Шаблон URL

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Значения по умолчанию

);

}

}

2. Атрибутивная маршрутизация (Attribute Routing)

Атрибутивная маршрутизация позволяет задавать маршруты прямо в контроллерах и их действиях с помощью атрибутов. Этот способ маршрутизации более гибкий и дает больше контроля над тем, как обрабатываются конкретные маршруты.

Включение атрибутивной маршрутизации:

Для использования атрибутивной маршрутизации нужно включить её в файле RouteConfig.cs:

public class RouteConfig

{

public static void RegisterRoutes(RouteCollection routes)

{

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

// Включаем атрибутивную маршрутизацию

routes.MapMvcAttributeRoutes();

}

}

### 

### 4 Лабораторная

**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.

Контроллеры ASP.NET Core используют промежуточное ПО маршрутизации для сопоставления URL-адресов входящих запросов и действий. Действия бывают двух типов:

* Условно-маршрутизированные (обычная маршрутизация)
* Атрибутно-маршрутизированные (маршрутизация на основе атрибутов)

Размещение маршрута на контроллере или действии делает его атрибутно-маршрутизированным.

**Маршрутизация на основе условностей / соглашений**(обычная) — это подход, при котором маршруты определяются в конфигурации приложения на основе заранее установленных правил. **MapControllerRoute**.

Он называется обычной маршрутизацией , поскольку устанавливает соглашение для путей URL:

* Первый сегмент пути, {controller=Home}, соответствует имени контроллера.
* Второй сегмент, {action=Index}, соответствует названию действия .
* Третий сегмент {id?}используется для необязательного id. ?In {id?}делает его необязательным. idиспользуется для сопоставления с сущностью модели.

**Маршрутизация на основе атрибутов** позволяет разработчикам определять маршруты непосредственно в контроллерах и методах действий с использованием атрибутов.

API REST должны использовать маршрутизацию атрибутов для моделирования функциональности приложения как набора ресурсов, где операции представлены HTTP-глаголами .

 Использует набор атрибутов для сопоставления действий напрямую с шаблонами маршрутов.

 Требует больше входных данных для указания маршрута.

 Позволяет точный контроль над шаблонами маршрутов для каждого действия.

 Имена контроллера и действия не играют роли в сопоставлении, если не используется замена токенов.

**Как работает маршрутизация**

При запуске приложения ASP.NET Core регистрируются маршруты и настраивается конвейер обработки запросов в классе Startup, где указываются необходимые сервисы и промежуточное ПО.

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 определёнными маршрутами, зарегистрированными 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 на основе сопоставленного маршрута.

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 маршрутов, определенных с помощью обычной маршрутизации (на основе соглашений).
* 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. Например, маршрут, определяющий конкретный идентификатор ресурса, будет иметь более высокий приоритет, чем маршрут, который определяет ресурс без идентификатора.

**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.

**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.

Наиболее конкретные маршруты имеют возможность выполняться до более общих маршрутов.

**У кого больше приоритет: Route или таблица маршрутизации**

Приоритет всегда отдается маршрутам, заданным с помощью атрибутов, если они есть. Таблица маршрутизации используется для обработки более общих маршрутов и работает, когда запросы не сопоставляются с атрибутной маршрутизацией.

**Как явно изменить приоритет атрибутов Route**

**Маршруты атрибутов** могут настроить порядок с помощью Order свойства. Все предоставленные атрибуты маршрута платформы включают Order . Маршруты обрабатываются в порядке возрастания значения свойства Order. Порядок по умолчанию — 0.

Свойство Order в атрибутах маршрутизации ASP.NET Core позволяет явно задавать порядок, в котором маршруты обрабатываются при сопоставлении входящих запросов. Это свойство может быть особенно полезным в случаях, когда у вас есть несколько маршрутов, которые могут совпадать с одним и тем же URL, и вы хотите контролировать, какой маршрут будет иметь приоритет при обработке запросов.

Пример: [HttpGet("{id:int}", Order = 1)]

**Как работают catch all параметры**

В ASP.NET Core маршрутизация поддерживает так называемые catch-all параметры, которые позволяют захватывать все оставшиеся части URL-адреса после определенного сегмента. Это полезно, когда вам нужно обработать динамические сегменты URL, которые не могут быть точно определены заранее.

Чтобы создать catch-all параметр, вы используете символ \* перед именем параметра в шаблоне маршрута.

* Объявление: Они объявляются в маршруте с использованием символа \*, который указывает на то, что будет захвачено все, что следует после определенного сегмента URL.
* Захват сегментов: При получении запроса catch-all параметры захватывают все оставшиеся части URL после указанного сегмента и передают их в метод контроллера как строку.
* Многоуровневость: Catch-all параметры могут захватывать многоуровневые пути, что позволяет получать полные пути или структуры из URL.
* Ограничения: В одном маршруте может быть только один catch-all параметр, и его размещение в маршруте, как правило, должно быть в конце, чтобы избежать путаницы с другими маршрутами.
* Обработка данных: Внутри метода контроллера значение catch-all параметра может быть обработано как обычная строка, что позволяет выполнять различные операции, такие как разбиение на сегменты или анализ содержимого.

 Единственный catch-all параметр: В одном маршруте может быть только один catch-all параметр.

 Расположение в маршруте: Catch-all параметры обычно размещаются в конце шаблона маршрута. Если он находится в середине или начале, это может привести к путанице и нежелательным результатам.

 Необходимость в обработке: В некоторых случаях вам может потребоваться специальная обработка для значения, захваченного catch-all параметром, чтобы правильно обработать разные типы запросов или ошибок.

**Как вообще работают параметры, какие есть ограничения**

Как работают параметры

Параметры маршрута объявляются в шаблонах маршрута с помощью фигурных скобок {}. Например, {id} обозначает параметр, который будет извлечен из URL.

Когда запрос поступает, ASP.NET Core анализирует URL и пытается сопоставить его с определенными маршрутами. Если маршрут содержит параметры, их значения извлекаются из URL и передаются в метод контроллера как аргументы.

Параметры могут быть различных типов, включая целые числа, строки и другие типы, которые могут быть автоматически преобразованы. Если преобразование не удается, ASP.NET Core возвращает ошибку.

Параметры могут быть объявлены как необязательные с помощью знака вопроса ?. Например, {id?} означает, что этот параметр может отсутствовать в URL.

Как упоминалось ранее, catch-all параметры обозначаются символом \*, и они могут захватывать все оставшиеся части URL после указанного сегмента.

Ограничения параметров

1. Тип параметра

int: Параметр должен быть целым числом.

float: Параметр должен быть числом с плавающей запятой.

bool: Параметр должен быть логическим значением (true или false).

alpha: Параметр должен содержать только буквенные символы (алфавитные).

regex: Параметр должен соответствовать заданному регулярному выражению.

string: Параметр — строка без ограничений (по умолчанию).

2. Ограничения по длине

minlength(n): Минимальная длина строки (параметр должен содержать не менее n символов).

maxlength(n): Максимальная длина строки (параметр не может превышать n символов).

3. Ограничения по диапазону

min(n): Минимальное значение для числовых параметров (параметр должен быть не менее n).

max(n): Максимальное значение для числовых параметров (параметр не может превышать n).

4. Обязательные и необязательные параметры

Обязательные параметры: Параметры по умолчанию являются обязательными; если они не указаны, будет возвращена ошибка 404 (Not Found).

Необязательные параметры: Можно сделать необязательными, добавив знак вопроса ? (например, {id?}).

5. Именование параметров

Имена параметров должны совпадать с именами параметров в методе контроллера, иначе значения не будут привязаны.

6. Порядок параметров

Необязательные параметры должны располагаться в конце маршрута, после обязательных.

7. Множественные параметры

В одном маршруте не может быть нескольких параметров с одинаковыми именами.

8. Чувствительность к регистру

Параметры могут быть чувствительными к регистру в зависимости от конфигурации.