Оглавление

[1 Лабораторная 2](#_Toc185237530)

[2 Лабораторная 15](#_Toc185237531)

[3 Лабораторная 24](#_Toc185237532)

[4 Лабораторная 38](#_Toc185237533)

[5 Лабораторная 45](#_Toc185237534)

[6 Лабораторная 53](#_Toc185237535)

[7 Лабораторная 62](#_Toc185237536)

### 1 Лабораторная

**Интернет**: Всемирная компьютерная сеть, построенная на основе стека протоколов TCP/IP.

**Из чего состоит Интернет?**

1. TCP/IP - сетевая модель передачи данных, представленных в цифровом виде

2. Службы (сервисы) Интернет:

DNS,

E-mail(STMP,POP3, IMAP),

IRC(обмен сообщений в реальном времени),

FTP,

Telnet (управление удаленным компьютером в терминальном режиме),

WWW.

3. Документация

Рабочее предложение (англ. Request for Comments, RFC) — документ из серии пронумерованных информационных документов Интернета, содержащих технические спецификации и стандарты, широко применяемые во всемирной сети.

4. Организация

Общество Интернета (англ. Internet Society, ISOC) — международная профессиональная организация, занимающаяся развитием и обеспечением доступности сети Интернет.

Распределенное приложение имеет архитектуру **клиент-сервер**, если все процессы распределенного приложения можно условно разбить на две группы. Одна группа процессов называется серверами другая – клиентами. Обмен данными осуществляется только между процессами-клиентами и процессами-серверами. Основное отличие процесса-клиента от процесса-сервера в том, что инициатором обмена данными всегда является процесс-клиент. Другими словами процесс-клиент обращается за услугой (сервисом) к процессу-серверу.

Существует несколько основных типов клиент-серверной архитектуры. Это нужно учитывать при построении собственного веб-приложения. От архитектуры зависит производительность и другие параметры работы системы.

**Один уровень взаимодействия** (1-Tier). В этой архитектуре клиент и сервер находятся на одном и том же уровне и выполняют какие-либо схожие функции. Однородная архитектура обычно используется в простых системах, где нет четкого разделения обязанностей между клиентом и сервером, и оба выполняют схожие функции для обмена данными и управления приложением.

**Два уровня** (2-Tier) В этом типе клиент и сервер составляют два уровня. Первый обычно отвечает за визуальное отображение данных и взаимодействие с пользователем. Сервер – за хранение массивов и выполнение бизнес-логики. Принцип работы двухуровневой архитектуры взаимодействия клиент-сервер заключается в том, что обработка запроса происходит на одном сервере без обращения к другим серверам в процессе этой обработки.

**Три уровня** (3-Tier) Клиент, сервер ПО и базы образуют три уровня. Первый отвечает за представление данных. Сервер приложений управляет бизнес-логикой и приемом запросов. База – содержит информацию.

**Много уровней в системе** (N-Tier) Эта архитектура имеет больше чем три уровня, что позволяет дополнительно разделять компоненты и функции приложения.Каждый уровень может быть разбит на подуровни в зависимости от сложности приложения.

**Взаимосвязь дуплекса и клиент-серверной архитектуры:**

**Полудуплекс** в клиент-серверной архитектуре: В режиме полудуплекса клиент и сервер могут передавать данные только поочередно. Например, клиент отправляет запрос, сервер обрабатывает его и только после этого отправляет ответ. В полудуплексном режиме клиент должен дождаться завершения обработки запроса на сервере, прежде чем снова отправить запрос или получить данные.

**Полнодуплекс** в клиент-серверной архитектуре: В полнодуплексном режиме клиент и сервер могут одновременно отправлять и получать данные, что особенно важно для приложений, требующих низких задержек и постоянного обмена данными. Полнодуплексная передача увеличивает скорость взаимодействия между клиентом и сервером, так как они могут обмениваться информацией параллельно, не дожидаясь завершения передачи данных одной из сторон.

**Модель TCP/IP** — это стек протоколов, которые задают правила передачи данных по Сети. Так как здесь правят протоколы TCP и IP, в честь них и назвали всю модель. Эти два протокола работают в связке: IP строит маршрут, а TCP контролирует, чтобы всё передавалось правильно.

TCP (Transmission Control Protocol) отвечает за обмен данными. Он управляет их отправкой и следит за тем, чтобы они дошли до получателя в целости.

IP (Internet Protocol) отвечает за адресацию. Его задача — связывать друг с другом устройства и нарезать данные на пакеты для удобной отправки.

**HTTP** (Hypertext Transfer Protocol) – это протокол Прикладного уровня, доставляющий информацию между различными гипер медийными системами. Под понятием гипермедийной системы понимается компьютерное представление системы данных, элементы которой представляются в различных форматах (гипертекст, графические изображения, видеоизображения, звук и т.д.) и обеспечивается автоматическая поддержка смысловых связей между представлениями элементов.

Запросы клиентов содержат URI (Uniform Resource Identifier) - универсальный идентификатор ресурса, позволяющий определить у сервера затребованный ресурс. URI представляет собой сочетание URL (Uniform Resource Locator) и URN (Uniform Resource Name).

URI – имя и адрес ресурса в сети, включает в себя URL и URN. Однако в реальности URI и URL частно отождествляют и используют взаимозаменяемо. Подобные адреса гораздно проще запомнить и использовать, нежели IP-адреса.

URN – имя ресурса в сети, определяет только название ресурса, но не говорит как к нему подключиться.

Общая форма адреса URL выглядит следующим образом:

scheme://username:password@host:port/path?query#fragment

**scheme**: Протокол (например, http, https, ftp).

**host**: Имя хоста или IP-адрес (например, www.example.com).

**port** (опционально): Номер порта (например, :80 для HTTP или :443 для HTTPS).

**path**: Путь к ресурсу (например, /page).

**query** (опционально): Параметры запроса (например, ?id=123&name=abc).

**fragment** (опционально): Фрагмент документа или якорь (например, #section1).

Символ @ отделяет учетные данные от домена. Параметр user\_id предоставляет логин, а password - пароль для доступа к ресурсу. Они отделяются друг от друга двоеточием.

В конце после символа решетки # может указываться фрагмент - необязательная строка для идентификации компонентов внутри URL. Обычно применяется в веб-браузерах для навигации по частям веб-страницы.

Но для протокола IP, через который идет взаимодействие, URI-адреса не существуют. Поэтому при отправке или передаче данных по доменному имени, компьютер еще обращается к системе доменных имен или Domain Name System (DNS), которая выполняют сопоставление между интернет-адресами в формате IPv4 или IPv6 и доменными названиями.

**Основные свойства HTTP**

версии HTTP/1.1 – действующий (текстовый), HTTP/2 – черновой (не распространен, бинарный);

два типа абонентов: клиент и сервер;

два типа сообщений: request и response;

от клиента к серверу – request;

от сервера к клиенту – response;

на один request всегда один response, иначе ошибка;

одному response всегда один request, иначе ошибка;

TCP-порты: 80, 443;

для адресации используется URI или URN;

поддерживается W3C, описан в нескольких RFC.

**Запросы** содержат следующие элементы:

HTTP-метод

Путь к ресурсу

Версию HTTP-протокола.

Заголовки (опционально), предоставляющие дополнительную информацию для сервера.

Параметры

Расширение

**Методы запроса:**

OPTIONS: Запрашивает информацию о поддерживаемых сервером методах для указанного ресурса.

GET: Получает данные с сервера, не изменяя ресурс.

HEAD: Похоже на GET, но возвращает только заголовки без тела ответа.

POST: Отправляет данные на сервер для создания или обработки ресурса.

PUT: Загружает данные на сервер, заменяя текущий ресурс.

DELETE: Удаляет указанный ресурс с сервера.

TRACE: Выполняет тестовый запрос, возвращая полный путь до ресурса через промежуточные узлы.

CONNECT: Устанавливает двустороннее соединение через прокси, обычно для работы с HTTPS.

Метод HTTP является **идемпотентным**, если повторный идентичный запрос, сделанный один или несколько раз подряд, имеет один и тот же эффект, не изменяющий состояние сервера. Другими словами, идемпотентный метод не должен иметь никаких побочных эффектов , кроме сбора статистики или подобных операций. Корректно реализованные методы GET, HEAD, PUT и DELETE идемпотентны, но не метод POST. Также все безопасные методы являются идемпотентными.

**Заголовки HTTP** позволяют клиенту и серверу отправлять дополнительную информацию с HTTP запросом или ответом. В HTTP-заголовке содержится не чувствительное к регистру название, а затем после (:) непосредственно значение. Пробелы перед значением игнорируются.

**Общие заголовки**

Эти заголовки могут присутствовать как в запросах, так и в ответах и касаются передачи сообщений. Применяется как к запросам, так и к ответам, но не имеет отношения к данным, передаваемым в теле.

Дата и время отправки сообщения.

Указывает, будет ли соединение оставаться открытым после завершения текущей передачи

Определяет параметры кэширования данных

**Заголовки запроса**

Используются клиентом для передачи дополнительной информации о запросе на сервер. Заголовки запроса содержит больше информации о ресурсе, который нужно получить, или о клиенте, запрашивающем ресурс.

Указывает доменное имя сервера и (необязательный) номер порта.

Информация о клиенте (например, браузер, версия, операционная система).

Указывает типы данных, которые клиент готов принять (например, text/html, application/json).

Данные для авторизации (например, токен или пара логин/пароль).

Передача данных cookie с клиента на сервер.

URL-адрес страницы, откуда был сделан запрос.

Тип данных, которые передаются в теле запроса

**Заголовки ответа**

Заголовки ответа содержат дополнительную информацию об ответе, например его местонахождение, или о сервере, предоставившем его.

Информация о сервере, который отвечает на запрос.

Устанавливает cookie в браузере клиента.

Указывает MIME-тип возвращаемых данных

Длина тела ответа в байтах.

Указывает URL для перенаправления

Указывает, какой тип авторизации требуется для доступа к ресурсу.

**Заголовки сущности**

Описывают тело сообщения (сущность) и могут присутствовать как в запросах, так и в ответах. Заголовки сущности содержат информацию о теле ресурса, например его длину содержимого или тип MIME.

Указывает, как данные в теле закодированы (например, gzip, compress).

Язык содержимого (например, en-US, ru-RU).

Определяет, как контент должен быть отображен (например, inline или attachment для скачивания файла).

Уникальный идентификатор версии ресурса для управления кэшированием.

Дата последнего изменения ресурса.

Заголовки также могут быть сгруппированы согласно тому, как прокси (proxies) обрабатывают их

**Передача параметров**

Строка запроса: ?параметр1=значение1&параметр2=значение2

string a = httpContext.Request.Query["ParmA"];

Описание: Данный код получает значение параметра ParmA из строки запроса. Строка запроса — это часть URL, которая идет после знака вопроса (?).

return Results.Content(result, "text/plain");

Описание: Этот оператор возвращает созданную строку result в ответ на запрос, при этом указывая, что MIME-тип контента — "text/plain". Это означает, что содержимое ответа — простой текст.

Данные передаются в теле запроса, обычно используется для POST, PUT, PATCH и других методов.

POST /submit-form

Content-Type: application/x-www-form-urlencoded

name=John&age=30

Cookies: Cookie: sessionId=abc123; token=xyz456

**Ответы** содержат следующие элементы:

Версию HTTP-протокола.

HTTP код состояния, сообщающий об успешности запроса или причине неудачи.

Сообщение состояния — краткое описание кода состояния.

HTTP заголовки, подобно заголовкам в запросах.

Расширение

Опционально: тело, содержащее пересылаемый ресурс.

**Код состояния ответа** HTTP показывает, был ли успешно выполнен определённый HTTP запрос.

Информационные ответы (100 – 199)

Успешные ответы (200 – 299)

Сообщения о перенаправлении (300 – 399)

Ошибки клиента (400 – 499)

Ошибки сервера (500 – 599)

**HTTPS** (HyperText Transfer Protocol Secure) – это безопасный протокол передачи данных, который поддерживает шифрование посредством криптографических протоколов SSL и TLS, и является расширенной версией протокола HTTP.

SSL и TSL – это криптографические протоколы, которые шифруют данные, передаваемые между сервером и браузером. Расшифровать их могут только сервер и браузер, поэтому на сайтах с HTTPS можно спокойно оплачивать покупки и вводить свои данные – они будут под надежной защитой.

**TLS** — это протокол шифрования и аутентификации, разработанный для защиты интернет-коммуникаций.

**TLS-рукопожатие** — это процесс, который запускает сеанс связи, использующий TLS. Во время TLS-рукопожатия две взаимодействующие стороны обмениваются сообщениями, чтобы подтвердить друг друга, проверить друг друга, установить алгоритмы шифрования, которые они будут использовать, и согласовать ключи сеанса.

В ходе TLS-рукопожатия клиент и сервер вместе выполняют следующие действия:

Указывают, какую версию TLS они будут использовать

Решают, какие наборы шифров они будут использовать

Проверяют подлинность сервера с помощью открытого ключа сервера и цифровой подписи центра сертификации SSL.

Генерируют сеансовые ключи, чтобы использовать симметричное шифрование после завершения рукопожатия

Этапы TLS-рукопожатия

Сообщение "hello" клиента: Клиент посылает сообщение "hello", содержащее версию протокола, random клиента и список наборов шифров.

Сервер генерирует главный секрет: На данный момент сервер получил клиентский пароль, а также параметры клиента и наборы шифров. У него уже есть случайное значение сервера, так как он может генерировать его самостоятельно.

Приветствие сервера и сообщение "Готово": Приветствие сервера включает сертификат сервера, цифровую подпись, пароль сервера и выбранный набор шифров. Поскольку у него уже есть главный секрет, он также отправляет сообщение "Готово".

Заключительные шаги и сообщение клиента "Готово": Клиент проверяет подпись и сертификат, генерирует главный секрет и отправляет сообщение "Готово".

Безопасность симметричного шифрования достигнута

**Сессия** (Session): серверный объект, хранящий информацию о соединении с клиентом, создается при первом обращении.

Время жизни: timeout (системный параметр, обычно равен 10 – 30 минутам) – максимальное время между запросами клиента. Если timeout превышен, то Session разрушается и при следующем запросе создается новый экземпляр.

Каждая сессия имеет собственный идентификатор (Session ID, 16 или более байт). Каждый Request (серверный объект, который образуется в результате обработки сервером http-запроса, поступающего от клиента и передается серверному программному коду для обработки) принадлежит, какой-то сессии (имеет ссылку на объект Session или содержит Session ID). Обычно объект Session предоставляет приложению возможность хранить данные в формате ключ/значение.

**Обработчик HTTP-запросов** - простейшее приложение ASP.NET Core

HTTP-обработчики (HTTP Handlers) используются для генерации содержимого ответа на HTTP-запрос. Мы можем использовать HTTP-обработчики для настройки обработчик запросов. При обработке одного запроса мы можем задействовать несколько различных модулей, но только один http-обработчик может быть сопоставлен с запросом.

**Веб-приложение** - сущность, расположенная на стороне сервера и имеющая URL/URI, к которой можно сделать http-запрос и получить http-ответ. Одно web-приложение представлено одним или более ресурсов.

**Cookie:** фрагмент данных, отправленный web-сервером и хранимый web-клиентом. Используется для аутентификации, хранения пользовательских предпочтений, статистики, информации о сеансе (обычно Session ID). Обычно имеет имя, содержащее URL, может иметь срок действия. Для создания и пересылки Cookie применяются заголовки.

**ASP.NET Core** представляет фреймворк для создания веб-приложений на платформе .NET, развиваемую компанией Microsoft. В качестве языков программирования для разработки приложений на ASP.NET Core используются C# и F#.ASP.NET Code является кроссплатформенным фреймворком с открытым кодом.

ё.NET – это платформа с открытым исходным кодом для создания настольных, мобильных и веб-приложений, которые могут работать в любой операционной системе.

Архитектура:

**1. Модели взаимодействия с пользователем** (Определить, как приложение выглядит и как с ним взаимодействует пользователь)

MVC (Model-View-Controller): Разделяет код на три части - модель (данные), представление (отображение), контроллер (обработка данных и действий пользователя). Позволяет создавать сложные, хорошо структурированные приложения.

Razor Pages: Упрощенный подход, где код обработки запроса и разметка страницы находятся вместе. Подходит для небольших и средних приложений.

SPA (Single Page Application): Подход, где основная часть логики выполняется на стороне клиента (браузер) с помощью JavaScript фреймворков (Angular, React, Vue). Обеспечивает высокую интерактивность и скорость работы.

Blazor: для создания интерактивных приложений, которые могут работать как на стороне сервера, так и на стороне клиента.

MVC, Razor Pages, SPA и Blazor не являются частью процесса обработки запроса на уровне ASP.NET Core и middleware. Они определяют, как именно будет сформирован запрос и как будет обработан полученный ответ.

**2. ASP.NET Core и Middleware:** Обработать запрос пользователя и вернуть ответ

Middleware:

Компоненты, выполняющие специфические задачи: аутентификация, авторизация, обработка cookie, логирование, кэширование и т.д.

Каждый компонент middleware может обработать запрос, изменить его или передать дальше по цепочке.

Порядок middleware в конвейере важен, так как он определяет порядок их выполнения.

**3. Веб-сервер:** Принимать HTTP-запросы от клиентов и передавать их приложению ASP.NET Core.

Kestrel: Кроссплатформенный веб-сервер, встроенный в ASP.NET Core. Быстрый и легковесный, но не предназначен для работы с большим количеством одновременных подключений.

IIS (Internet Information Services): Веб-сервер от Microsoft, доступный только на Windows. Мощный и функциональный, но более ресурсоемкий.

HTTP.sys: Низкоуровневый веб-сервер, встроенный в Windows. Обеспечивает высокую производительность и безопасность.

**Процесс обработки запроса:**

Браузер отправляет HTTP-запрос на веб-сервер.

Веб-сервер передает запрос приложению ASP.NET Core.

ASP.NET Core использует систему маршрутизации, чтобы определить, какой компонент (Controller в MVC, Razor Page или обработчик запроса API) должен обработать запрос.

При получении запроса, система маршрутизации ASP.NET Core проверяет URL запроса и сопоставляет его с настроенными маршрутами. Маршрутизатор определяет, каким обработчиком запрос должен обрабатываться.

Запрос проходит через цепочку middleware.

Middleware могут выполнять аутентификацию, авторизацию, логирование, кэширование и другие задачи.

Middleware применяются до того, как запрос достигнет вашего обработчика.

Выбранный обработчик выполняет логику приложения:

Взаимодействует с моделями данных.

Вызывает бизнес-логику.

Формирует данные для ответа.

В зависимости от выбранной модели взаимодействия с пользователем (MVC, Razor Pages, SPA, Blazor) происходит формирование ответа

Ответ проходит обратно через цепочку middleware.

Middleware могут модифицировать ответ, добавлять заголовки и т.д.

Веб-сервер отправляет ответ браузеру.

Браузер отображает полученный HTML, CSS и JavaScript код.

Одна из основных задач приложения - это обработка входящих запросов. Обработка запроса в ASP.NET Core устроена по принципу конвейера, который состоит из компонентов. Подобные компоненты еще называются middleware.

При получении запроса сначала данные запроса получает первый компонент в конвейере. После обработки запроса компонент middleware он может закончить обработку запроса - такой компонент еще называется терминальным компонентом. Либо он может передать данные запроса для обработки далее по конвейеру - следующему в конвейере компоненту и так далее. После обработки запроса последним компонентом, данные запроса возвращаются к предыдущему компоненту. Схематически это можно отобразить так:

Компоненты middleware встраиваются с помощью методов расширений Run, Map и Use интерфейса IApplicationBuilder. Класс WebApplication реализует данный интерфейс и поэтому позволяет добавлять компоненты middleware с помощью данных методов.

Для создания компонентов middleware используется делегат RequestDelegate, который выполняет некоторое действие и принимает контекст запроса - объект HttpContext:

public delegate Task RequestDelegate(HttpContext context);

При получении запроса сервер формирует на его основе объект HttpContext, которые содержит всю необходимую информацию о запросе. Эта информация посредством объекта HttpContext передается всем компонентам middleware в приложении.

Метод Run добавляет терминальный компонент - такой компонент, который завершает обработку запроса. Поэтому соответствено он не вызывает никакие другие компоненты и обработку запроса дальше - следующим в конвейере компонентам не передает. Поэтому данный метод следует вызывать в самом конце построения конвейера обработки запроса. До него же могут быть помещены другие методы, которые добавляют компоненты middleware. В качестве параметра метод Run принимает делегат RequestDelegate.

Компоненты middleware создаются один раз и существуют в течение всего жизненного цикла приложения. То есть для последующей обработки запросов используются одни и те же компоненты.

**Структура проекта ASP.NET Core**

Dependencies: все добавленные в проект пакеты и библиотеки, иначе говоря зависимости

Properties: узел, который содержит некоторые настройки проекта. В частности, в файле launchSettings.json описаны настройки запуска проекта, например, адреса, по которым будет запускаться приложение.

appsettings.json: файл конфигурации приложения в формате json

appsettings.Development.json: версия файла конфигурации приложения, которая используется в процессе разработки

Program.cs: главный файл приложения, с которого и начинается его выполнение. Код этого файла настраивает и запускает веб-приложение

**OWIN** (Open Web Interface for .NET) — это стандарт, созданный для отделения веб-сервера от серверного приложения в экосистеме .NET. Основная цель OWIN — обеспечить абстракцию, которая позволяет различным веб-серверам и приложениям взаимодействовать друг с другом без необходимости тесной привязки.

1. Host – приложение-процесс операционной системы, управляющий жизненным циклом OWIN Server.

2. OWIN Server - http-сервер, реализующий интерфейс OWIN.

3. OWIN: обеспечивает интерфейсы: между приложением и http-сервером, между http-сервером и Host, между http-сервером и middleware.

4. middleware – подключенные компоненты (модули), предназначенные для обработки запросов.

Запрос от браузера попадает на Host.

Host передает запрос OWIN Server.

OWIN Server, используя спецификацию OWIN, передает запрос первому middleware в цепочке.

Каждый middleware выполняет свою задачу и передает запрос следующему middleware.

Последний middleware формирует ответ и передает его обратно по цепочке.

OWIN Server отправляет ответ обратно Host.

Host возвращает ответ браузеру.

**Katana** — это проект, созданный Microsoft для поддержки стандарта OWIN. Этот стандарт предназначен для того, чтобы разделить веб-приложения .NET от серверов, на которых они запускаются. Это делает приложения более независимыми от конкретных серверных технологий.

IIS Host:

IIS (Internet Information Services) — Веб-сервер, который обрабатывает запросы от пользователей. При получении HTTP-запроса IIS перенаправляет его на приложение, которое работает на одном из нижележащих серверов (SystemWeb или Helios).

SystemWeb — Старая модель обработки запросов через ASP.NET pipeline. Принимает запросы от IIS, обрабатывает их, используя встроенные механизмы ASP.NET, и возвращает ответ обратно через IIS.

Helios — Современный сервер, который работает с OWIN и позволяет обойти ограничения SystemWeb. Принимает запросы от IIS напрямую, минуя старую модель обработки. Это даёт больше гибкости для разработки и интеграции с Middleware.

Пользователь отправляет HTTP-запрос

Веб-сервер **IIS** принимает запрос.

**IIS** анализирует запрос и определяет, какое приложение (SystemWeb или Helios) должно его обработать.

Если используется **SystemWeb**:

Запрос поступает в стандартный ASP.NET pipeline.

Обрабатывается с помощью встроенных механизмов, таких как HttpApplication, где выполняются события жизненного цикла (например, BeginRequest, EndRequest).

Если используется **Helios**: Запрос поступает в OWIN pipeline, где он может пройти через несколько компонентов Middleware.

Запрос передаётся по цепочке Middleware-компонентов, которые выполняют свои задачи

После завершения обработки Middleware запрос передаётся в основное приложение

Приложение выполняет свою бизнес-логику:

Создаётся ответ (например, HTML-страница, JSON-данные).

Ответ передаётся обратно через Middleware (в обратном порядке):

Ответ возвращается в **IIS**.

**IIS** отправляет окончательный ответ обратно пользователю через HTTP. Пользователь получает результат обработки своего запроса (например, страницу или данные в формате JSON).

Non-IIS Host — это сервер, который **не использует IIS** для хостинга приложения. Такие серверы могут быть полезны в тех случаях, когда нет необходимости в использовании большого и сложного IIS, либо когда требуется легковесное решение для хостинга.

OwinHost — Отдельный процесс, который запускает OWIN-приложение без IIS. Запускает приложение, обрабатывает HTTP-запросы через встроенные в .NET механизмы, предоставляя аналогичные функции, что и IIS, но в более легковесной форме.

Self-Host — Приложение само запускает себя как сервер.Использует HttpListener для обработки входящих запросов, что позволяет запускать приложение в любых контекстах (например, в консоли или десктопном приложении).

HttpListener, который отвечает за прослушивание HTTP-запросов. HttpListener — это простой встроенный в .NET сервер, который может обрабатывать HTTP-запросы. Он удобен для легковесных решений.

Middleware - Промежуточный слой, который обрабатывает запросы и ответы.

Middleware работает как прослойка между сервером (IIS или OWIN) и приложением. Он может выполнять различные функции, такие как аутентификация, логирование, обработка ошибок и т.д. Запрос проходит через него перед тем, как достигнет самого приложения, и после обработки ответа он также может пройти через этот слой.

Пользователь отправляет HTTP-запрос

**OwinHost** или **Self-Host** прослушивает входящие запросы с помощью **HttpListener**.

**HttpListener** принимает запрос и создаёт объект запроса.

Запрос передаётся в OWIN pipeline. Это промежуточный слой, который управляет обработкой запроса.

Если используется OwinHost, то запрос передаётся в OwinHost. Если используется Self-Host, то запрос передаётся в приложение, которое само выступает в роли сервера.

Запрос проходит через цепочку Middleware-компонентов.

После обработки в Middleware запрос передаётся в основное приложение (например, контроллер или обработчик).

Приложение выполняет свою бизнес-логику:

Создаётся ответ (например, HTML-страница, JSON-данные).

Ответ возвращается через Middleware (в обратном порядке)

Окончательный ответ передаётся обратно через **HttpListener**.

**HttpListener** формирует HTTP-ответ и отправляет его пользователю.

Пользователь получает ответ от OwinHost или Self-Host, содержащий результат обработки запроса (например, страницу или данные в формате JSON).

Разбор кода:

|  |
| --- |
| app.MapGet("/NAV", (HttpContext httpContext) => {  string a = httpContext.Request.Query["ParmA"];  string b = httpContext.Request.Query["ParmB"];  string result = $"GET-Http-NAV:ParmA = {a}, ParmB = {b}";  return Results.Content(result, "text/plain"); //Метод позволяет вернуть строку с указанием MIME-типа контента (text/plain),  }); |

1.app.MapGet("/NAV", ...)

Описание: Это метод маршрутизации (роутинг), который используется в приложении ASP.NET Core для обработки HTTP-запроса типа GET. Здесь указывается, что запросы по URL /NAV должны быть обработаны указанным делегатом.

2. (HttpContext httpContext) =>

Описание: Делегат или лямбда-функция, которая выполняется, когда приходит GET-запрос на URL /NAV. В качестве параметра эта функция принимает HttpContext, который содержит всю информацию о запросе и позволяет взаимодействовать с ответом.

HttpContext — это объект, представляющий информацию о текущем HTTP-запросе. С его помощью можно получить данные о запросе, заголовках, параметрах строки запроса (query string), теле запроса и другую полезную информацию.

3. string a = httpContext.Request.Query["ParmA"];

Описание: Данный код получает значение параметра ParmA из строки запроса. Строка запроса — это часть URL, которая идет после знака вопроса (?).

5. string result = $"GET-Http-NAV:ParmA = {a}, ParmB = {b}";

Описание: Здесь создается строка result с использованием строковой интерполяции. В эту строку подставляются значения параметров a и b. Например, если a = "value1", а b = "value2", то строка будет выглядеть так:

6. return Results.Content(result, "text/plain");

Описание: Этот оператор возвращает созданную строку result в ответ на запрос, при этом указывая, что MIME-тип контента — "text/plain". Это означает, что содержимое ответа — простой текст.

**5 задание**

 Когда сервер получает GET-запрос на /XML, он возвращает HTML-страницу.

Эта страница содержит форму с двумя полями для ввода чисел x и y и кнопку для выполнения расчета.

Функция JavaScript calculate() отправляет POST-запрос с введенными значениями на сервер при нажатии кнопки "Посчитать". Для этого используется объект XMLHttpRequest:

Значения полей формы кодируются и отправляются как параметры x и y.

Если сервер возвращает успешный ответ, результат (произведение x и y) выводится в элементе с ID result.

 **HTML-форма и JavaScript:**

Поля для ввода чисел (<input type='number'>).

Кнопка **Посчитать**, которая вызывает функцию **calculate()**.

Скрипт собирает данные из полей формы и отправляет их на сервер через AJAX-запрос.

 Когда сервер получает POST-запрос на /XML, он извлекает параметры формы x и y из тела запроса.

 Эти параметры приводятся к целым числам, и затем вычисляется их произведение.

 Если все корректно, сервер возвращает результат (произведение) как простой текст с MIME-типом text/plain.

 Если происходит ошибка (например, некорректные данные), сервер возвращает статус **400 Bad Request** и сообщение **"Неверные параметры"**.

Строка, в которой значения полей формы кодируются и отправляются как параметры x и y, находится внутри JavaScript-функции calculate(). Она выглядит так:

var formData = 'x=' + encodeURIComponent(x) + '&y=' + encodeURIComponent(y);

 **x и y**: Эти переменные содержат значения, которые пользователь вводит в поля формы с ID x и y.

 **encodeURIComponent()**: Эта функция кодирует значения x и y, чтобы они были корректно переданы через URL. Это важно, если в данных содержатся символы, которые имеют специальное значение в URL (например, пробелы, амперсанды и т. д.).

 **formData**: Строка, которая формируется в виде x=value1&y=value2, где value1 и value2 — это закодированные значения введенных пользователем чисел.

Эта строка затем отправляется в запросе с помощью XMLHttpRequest в следующей строке: xhr.send(formData);

 Данные отправляются на сервер через AJAX-запрос (объект XMLHttpRequest) **без перезагрузки страницы**.

 После отправки POST-запроса через JavaScript сервер возвращает результат умножения в виде текста, и этот результат динамически отображается на странице с помощью JavaScript (внутри элемента с ID result).

 Вся страница остается на месте, и только часть страницы (результат) обновляется.

**6 задание**

Когда сервер получает **GET-запрос** на /HTML, он возвращает HTML-страницу с формой, которая позволяет пользователю ввести два числа.

HTML-страница содержит:

Два поля ввода для чисел с именами x и y.

Кнопку "Посчитать" для отправки формы.

Форма отправляется методом **POST** на тот же маршрут (/HTML).

Таким образом, когда пользователь запрашивает страницу по пути /HTML, он видит форму для ввода чисел.

 Когда сервер получает **POST-запрос** на тот же маршрут /HTML, он извлекает значения полей формы x и y из тела запроса.

 Значения этих полей приводятся к целым числам с помощью int.Parse().

 Затем вычисляется произведение этих чисел и записывается в переменную result.

 Сервер возвращает новую HTML-страницу, на которой выводится результат умножения в виде абзаца (<p>).

 MIME-тип ответа также установлен на text/html; charset=utf-8, что указывает браузеру, что содержимое является HTML-документом.

 При нажатии на кнопку "Посчитать" форма отправляется с помощью стандартного HTTP-запроса POST.

 После отправки формы происходит **перезагрузка всей страницы**. Сервер возвращает новую HTML-страницу с результатом (произведение чисел).

 Таким образом, каждый раз при отправке POST-запроса браузер загружает новую страницу, а не просто обновляет часть интерфейса.

**Пояснение атрибута action:**

**method='POST'**: Указывает, что форма будет отправлена с использованием метода POST.

**action='/HTML'**: Указывает, что данные формы будут отправлены на маршрут /HTML. Это значит, что при отправке формы запрос будет отправлен на тот же маршрут, который используется для отображения самой формы.

async в данном коде является ключевым словом в C# и используется для работы с асинхронными операциями. Оно указывает, что метод является **асинхронным**, то есть может выполняться в неблокирующем режиме, позволяя другим задачам выполняться параллельно.

В данном случае async используется в контексте веб-запросов для повышения производительности и отзывчивости приложения.

**Зачем нужен async здесь?**

**Асинхронные операции ввода-вывода**:

Веб-серверу часто нужно выполнять задачи, связанные с вводом-выводом (I/O), такие как чтение запросов от клиента и запись ответов.

### 2 Лабораторная

Для построения эффективных web-приложений до появления HTML5 применялась технология **AJAX** (Asynchronous JavaScript and XML), которая основывалась на возможности доступного из JavaScript объекта XMLHTTPRequest, асинхронно отправлять http-запросы серверу и обрабатывать его ответы. В качестве формата передачи данных в АЈАХ, как правило, используются XML и JSON.

**WebSocket** - протокол для общения между клиентом и сервером, предоставляющий двухсторонне общение сверх протокола TCP. Мы подключаем WS один раз, а затем сервер может отдавать нам ответы тогда, когда посчитает нужным.

**Web Sockets API** - программный интерфейс, основанный на протоколе полнодуплексной связи WebSocket.

Для установки соединения (процедура websocket handshake) между клиентом и сервером WebSocket использует запрос и ответ в формате протокола НТТР. При этом протокол определяет две URL-схемы: ws для незашифрованного соединения и Wss — для зашифрованного.

Начинается с HTTP-запроса на установку соединения (handshake).

**Этапы рукопожатия WebSocket**

1. Инициирование запроса: Клиент отправляет HTTP-запрос на сервер с использованием метода GET. Запрос включает специальные заголовки, чтобы сервер знал, что клиент хочет установить WebSocket-соединение.

Пример запроса от клиента: клиент отправляет HTTP-запрос с методом GET и заголовками, которые указывают на намерение установить WebSocket соединение.

• GET /chat HTTP/1.1

• Host: example.com

• Upgrade: websocket

• Connection: Upgrade

• Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

• Sec-WebSocket-Version: 13

• Origin: http://example.com

Connection: Upgrade — указывает, что соединение должно быть обновлено

Upgrade: websocket — инструкция серверу для переключения на протокол WebSocket

Host: example.com — доменное имя сервера

Sec-WebSocket-Key — случайная строка, сгенерированная клиентом, которая используется сервером для создания ответа

Sec-WebSocket-Version — версия протокола WebSocket, которую поддерживает клиент

1. Ответ сервера: Если сервер поддерживает WebSocket и готов к установлению соединения, он отправляет ответ с кодом состояния 101 Switching Protocols. Этот ответ также содержит специальные заголовки, подтверждающие переход на WebSocket.

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

HTTP/1.1 101 Switching Protocols — код ответа 101 указывает на успешное обновление протокола

Upgrade: websocket — подтверждение, что протокол обновляется до WebSocket

Connection: Upgrade — подтверждение обновления соединения

Sec-WebSocket-Accept — хэш, созданный сервером на основе ключа Sec-WebSocket-Key, присланного клиентом. Этот хэш подтверждает, что сервер поддерживает WebSocket.

1. Установление соединения: После успешного рукопожатия устанавливается двустороннее соединение, и клиент и сервер могут обмениваться данными в реальном времени, используя сообщения WebSocket.

Sec-WebSocket-Key - Этот заголовок служит для проверки подлинности запроса на установление WebSocket-соединения.

• отправляет серверу случайную строку в заголовке Sec-WebSocket-Key. Это значение создается для каждого нового соединения.

• Сервер использует значение из заголовка Sec-WebSocket-Key для генерации ответа, который затем возвращает в заголовке Sec-WebSocket-Accept. Это предотвращает установление нежелательных или поддельных соединений.

• Клиент проверяет, что полученное значение соответствует ожидаемому, и только тогда соединение считается подтвержденным.

Sec-WebSocket-Extensions — заголовок для согласования дополнительных возможностей WebSocket-соединения, таких как сжатие данных или другие улучшения передачи.

permessage-deflate — это расширение для WebSocket, которое позволяет сжимать сообщения перед их отправкой, чтобы уменьшить объём передаваемых данных.

Проверяется поддержка WebSocket с помощью Modernizr.websockets.

ws:// — это схема URL, которая указывает, что соединение будет установлено по WebSocket-протоколу.

Также существует защищённая версия WebSocket — wss://, которая используется для зашифрованных соединений поверх TLS/SSL (подобно тому, как https:// работает для HTTP).

/Websockets.websocket — это путь на сервере, к которому подключается клиент..

**Описание кода**

**1. html страница**

<script src="https://cdnjs.cloudflare.com/ajax/libs/modernizr/2.8.3/modernizr.min.js"></script>

Файл modernizr\_ws.js содержит код библиотеки Modernizr , с помощью которой осуществляется проверка работоспособности Web Sockets API в текущем браузере.

**2. js**

var ws = null;

var ta;

var bstart;

var bstop;

Здесь объявляется переменная ws, которая будет использоваться для хранения объекта WebSocket. Изначально она равна null, что означает отсутствие подключения к WebSocket.

ta — будет хранить ссылку на элемент <textarea>

bstart — будет хранить ссылку на кнопку "Start"

bstop — будет хранить ссылку на кнопку "Stop"

При загрузке html-страницы выполняется анонимная функция, заданная в качестве обработчика события load встроенного объекта window

window.onload = function () {

if (Modernizr.websockets) {

console.log('Поддерживается');

ta = document.getElementById('ta')

bstart = document.getElementById('bstart');

bstop = document.getElementById('bstop');

bstart.disabled = false;

bstop.disabled = true;

}

}

**window.onload = function () { ... }** — это функция, которая срабатывает, когда веб-страница полностью загружена. То есть, весь HTML, CSS и JavaScript должны быть загружены и готовы к использованию.

if (Modernizr.websockets) — здесь проверяется, поддерживает ли браузер WebSocket. Библиотека Modernizr используется для того, чтобы проверять поддержку различных современных веб-технологий в браузерах. В данном случае проверяется наличие WebSocket API.

Если WebSocket поддерживается: console.log('Поддерживается'); — в консоль браузера

ta = document.getElementById('ta'); — находит элемент с идентификатором ta на странице и сохраняет ссылку на этот элемент в переменную ta.

bstart = document.getElementById('bstart'); — находит элемент с идентификатором bstart (кнопка "Start") и сохраняет ссылку на него в переменную bstart.

bstop = document.getElementById('bstop'); — находит элемент с идентификатором bstop (кнопка "Stop") и сохраняет ссылку на него в переменную bstop.

bstart.disabled = false; — кнопка "Start" активируется (её свойство disabled становится false, что позволяет пользователю нажимать на неё).

bstop.disabled = true; — кнопка "Stop" деактивируется (устанавливается свойство disabled в true, что делает её неактивной, пользователь не сможет её нажать).

function sendMessage() {

if (ws && ws.readyState === WebSocket.OPEN) {

ws.send("Соединение");

} else {

ta.value += "Отправка сообщений недоступна\n";

}

}

function sendMessage() — вызывается для отправки сообщения через WebSocket. Внутри этой функции происходит проверка доступности WebSocket-соединения и, если оно открыто, отправка сообщения. Если соединение недоступно, выводится сообщение об ошибке в интерфейсе.

if (ws && ws.readyState === WebSocket.OPEN) — здесь проверяется два условия:

ws — проверяется, что переменная ws (WebSocket-соединение) существует и не равна null. Это означает, что соединение было хотя бы инициировано.

ws.readyState === WebSocket.OPEN — проверяется состояние WebSocket.

readyState — это свойство, которое хранит текущий статус WebSocket-соединения:

0 — CONNECTING — соединение ещё устанавливается.

1 — OPEN — соединение установлено и готово для передачи данных.

2 — CLOSING — соединение закрывается.

3 — CLOSED — соединение закрыто или не удалось.

Условие проверяет, что WebSocket находится в состоянии OPEN, то есть соединение установлено и готово для отправки данных.

ws.send("Соединение"); — если условие истинно, и соединение WebSocket открыто, вызывается метод send. Этот метод отправляет строку "Соединение" через WebSocket на сервер. Таким образом, сообщение передаётся по открытому соединению.

else { ... } — если условие выше ложное (т.е. WebSocket не существует или не находится в состоянии OPEN), выполняется код в блоке else.

ta.value += "Отправка сообщений недоступна\n"; — если WebSocket-соединение недоступно

function exe\_start() {

if (!ws) {

ws = new WebSocket('ws://localhost:59391/Websockets.websocket');

ws.onopen = function () {

ta.value += "Соединение открыто\n";

bstart.disabled = true;

bstop.disabled = false;

sendMessage();

};

ws.onmessage = function (event) {

ta.value += event.data + "\n";

};

ws.onclose = function () {

ta.value += "Соединение закрыто.\n";

};

}

}

Для управления подключением для WebSocket определено три события:

• onopen: вызывается при успешном открытии подключения.

• onerror: вызывается при возникновении ошибки при подключении. В функцию обработчика передается информация об ошибке

• onclose: вызывается при закрытии подключения.

• onmessage: этот обработчик срабатывает при получении сообщения от сервера.

function exe\_start() — это функция, которая инициирует WebSocket-соединение с сервером. Она запускается, когда пользователь хочет начать соединение

if (!ws) — проверка, существует ли уже WebSocket-соединение.

!ws означает, что соединение ещё не было создано (переменная ws равна null или не определена). Если WebSocket уже был создан, то эта часть не будет выполняться, предотвращая создание нового соединения.

ws = new WebSocket('ws://localhost:59391/Websockets.websocket'); — здесь создаётся новое WebSocket-соединение, используя указанный URL-адрес WebSocket-сервера ('ws://localhost:59391/Websockets.websocket'). Это WebSocket соединение с локальным сервером, который работает на порту 59391. После этого объект WebSocket сохраняется в переменную ws.

ws.onopen = function () { ... } — событие onopen срабатывает, когда WebSocket-соединение успешно установлено. Функция внутри этого события выполняется после открытия соединения:

bstart.disabled = true; — кнопка "Start" (переменная bstart) деактивируется, поскольку соединение уже открыто, и повторное открытие не требуется.

bstop.disabled = false; — кнопка "Stop" активируется (свойство disabled становится false), чтобы дать возможность пользователю закрыть соединение.

sendMessage(); — функция sendMessage() вызывается сразу после открытия соединения, что позволяет отправить сообщение сразу после установки соединения

ws.onmessage = function (event) { ... } — событие onmessage срабатывает, когда WebSocket получает данные от сервера:

event.data содержит полученные данные от сервера.

ta.value += event.data + "\n"; — полученное сообщение добавляется в элемент <textarea>, выводя его пользователю. Новое сообщение выводится с переносом строки.

ws.onclose = function () { ... } — событие onclose срабатывает, когда WebSocket-соединение закрывается:

.

Конец условия — если WebSocket ещё не был создан, создаётся новый, в противном случае ничего не происходит.

function exe\_stop() {

if (ws) {

ws.close(3001, ' stopapplication');

ws = null;

bstart.disabled = false;

bstop.disabled = true;

}

}

function exe\_stop() — это функция для остановки WebSocket-соединения. Она вызывается, когда пользователь хочет разорвать соединение

if (ws) — проверка, существует ли WebSocket-соединение (переменная ws не должна быть null). Если соединение не установлено, дальнейший код внутри условия не будет выполнен.

ws.close(3001, 'stopapplication'); — здесь вызывается метод close() для закрытия WebSocket-соединения:

3001 — это код закрытия. По стандарту WebSocket, коды закрытия могут варьироваться от 1000 до 4999, где каждый код имеет своё значение. Код 3001 является произвольным пользовательским кодом, который может означать "запрос на остановку приложения" (его можно выбрать на своё усмотрение).

'stopapplication' — это текстовая причина закрытия соединения, которая может быть передана серверу. В данном случае текст 'stopapplication' означает "остановка приложения". Это помогает серверу понять, почему соединение было закрыто.

ws = null; — после того, как WebSocket-соединение было закрыто, переменная ws сбрасывается в null, чтобы указать, что соединение больше не существует. Это также предотвращает повторное закрытие уже закрытого соединения.

bstart.disabled = false; — кнопка "Start" вновь активируется (свойство disabled становится false), поскольку соединение разорвано, и его можно создать заново.

bstop.disabled = true; — кнопка "Stop" деактивируется (свойство disabled становится true), так как больше нет активного WebSocket-соединения, которое нужно остановить.

**3.Programm.cs**

public class IISHandler : IHttpHandler

{

WebSocket socket;

**IISHandler** — это класс, реализующий интерфейс IHttpHandler, который позволяет обрабатывать HTTP-запросы, в том числе запросы WebSocket. Внутри этого класса есть приватная переменная socket, которая хранит экземпляр WebSocket-соединения.

**IHttpHandler** — это интерфейс в ASP.NET, который определяет обработчик HTTP-запросов.

public bool IsReusable

{

get

{

return false;

}

}

**IsReusable** — это свойство интерфейса IHttpHandler. Если возвращается true, то один и тот же экземпляр обработчика может быть использован для обработки нескольких запросов. В данном случае возвращается false, что означает, что каждый запрос будет обрабатываться новым экземпляром класса.

public void ProcessRequest(HttpContext context)

{

if (context.IsWebSocketRequest)

{

context.AcceptWebSocketRequest(WebSocketRequest);

}

}

**ProcessRequest(HttpContext context)** — это основной метод, который вызывается для обработки HTTP-запросов. Этот метод является главным в IHttpHandler. Он принимает объект HttpContext, который предоставляет доступ к текущему запросу и ответу. Внутри этого метода реализуется логика обработки HTTP-запроса.

context.IsWebSocketRequest: Это свойство возвращает true, если текущий HTTP-запрос является WebSocket-запросом. Иными словами, это проверка, пытается ли клиент установить WebSocket-соединение.

Если это WebSocket-запрос, то он принимается с помощью метода AcceptWebSocketRequest, который вызывает асинхронный метод WebSocketRequest для обработки WebSocket-соединения.

AcceptWebSocketRequest: Этот метод сообщает серверу, что текущий запрос является запросом для установки WebSocket-соединения, и сервер начинает обработку этого соединения. В качестве аргумента передается делегат или метод, который будет вызван для обработки WebSocket-соединения. В данном случае это метод WebSocketRequest.

private async Task WebSocketRequest(AspNetWebSocketContext context)

{

socket = context.WebSocket;

string s = await Receive();

await Send(s);

int i = 0;

while (socket.State == WebSocketState.Open)

{

System.Threading.Thread.Sleep(1000);

await Send("[" + (i++).ToString() + "]");

}

}

WebSocketRequest(AspNetWebSocketContext context) — это асинхронный метод, который выполняет обработку WebSocket-соединения.

Сначала создаётся экземпляр WebSocket-соединения с помощью context.WebSocket, который хранится в переменной socket.

context.WebSocket — это свойство, которое предоставляет доступ к WebSocket-соединению, установленному клиентом.

string s = await Receive(); — вызывается метод Receive(), который получает сообщение от клиента.

await Send(s); — после получения сообщения оно отправляется обратно клиенту.

socket.State: Свойство, которое показывает текущее состояние WebSocket-соединения.

WebSocketState.Open: Значение перечисления, указывающее, что соединение активно и открыто. Цикл продолжается, пока соединение не закрыто.

В цикле while (socket.State == WebSocketState.Open) — пока WebSocket-соединение открыто, каждую секунду отправляется сообщение клиенту с индексированным значением (например, [0], [1], и так далее).

System.Threading.Thread.Sleep(1000); — задержка на 1 секунду между отправками сообщений.

private async Task<string> Receive()

{

string rc = null;

var buffer = new ArraySegment<byte>(new byte[512]);

var result = await socket.ReceiveAsync(buffer, CancellationToken.None);

rc = System.Text.Encoding.UTF8.GetString(buffer.Array, 0, result.Count);

return rc;

}

Receive() — это асинхронный метод, который получает данные от клиента:

Создаётся буфер размером 512 байт с помощью new ArraySegment<byte>(new byte[512]).

ArraySegment<byte>: Класс, который представляет часть массива байтов. Он используется для передачи фрагментов данных без необходимости копировать данные в новый массив.

Метод ReceiveAsync() асинхронно получает данные от клиента и сохраняет результат в переменную result.

Принятое сообщение конвертируется в строку с использованием кодировки UTF-8.

Возвращает полученную строку.

System.Text.Encoding.UTF8: Класс, который предоставляет методы для работы с кодировками текста. В данном случае используется кодировка UTF-8.

GetBytes(message): Метод, который преобразует строку message в массив байтов. Это необходимо, потому что WebSocket работает с бинарными данными, и строку нужно сначала преобразовать в байты.

private async Task Send(string s)

{

var sendbuffer = new ArraySegment<byte>(System.Text.Encoding.UTF8.GetBytes("Ответ: " + s));

await socket.SendAsync(sendbuffer, System.Net.WebSockets.WebSocketMessageType.Text, true, CancellationToken.None);

}

}

Send(string s) — это асинхронный метод, который отправляет сообщение клиенту:

Строка "Ответ: " добавляется к тексту сообщения.

Сообщение конвертируется в байты с использованием кодировки UTF-8.

Создаётся буфер для отправки.

Сообщение отправляется с помощью SendAsync().

Параметр WebSocketMessageType.Text указывает, что сообщение — это текст.

WebSocketMessageType.Text: Тип сообщения, указывающий, что отправляется текстовое сообщение. WebSocket поддерживает текстовые и бинарные сообщения. Вы отправляете текстовое сообщение, даже если фактически передаёте данные в байтовом виде. Это сигнализирует WebSocket, что данные должны быть интерпретированы как текст.

true: Указывает, что это финальный фрагмент сообщения. В WebSocket сообщение может состоять из нескольких фрагментов. Если true, это означает, что сообщение завершено и не будет продолжаться в следующих фрагментах.

CancellationToken.None: Токен отмены, который управляет отменой асинхронной операции. В этом случае используется CancellationToken.None, что означает отсутствие поддержки отмены

### 3 Лабораторная

**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:

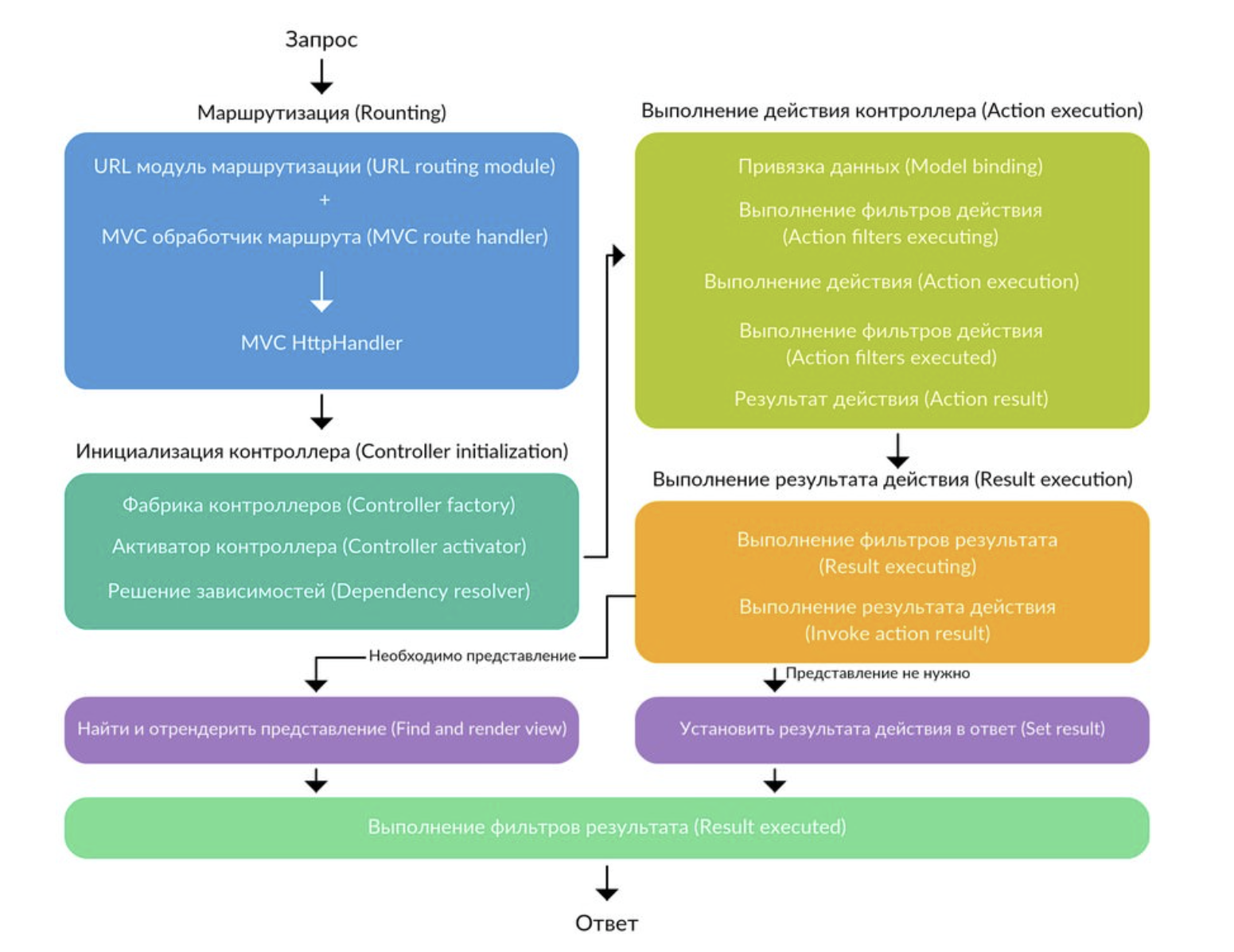
**Model (Модель)** — ответственна за управление данными. Это бизнес-логика приложения и взаимодействие с базой данных.

**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.

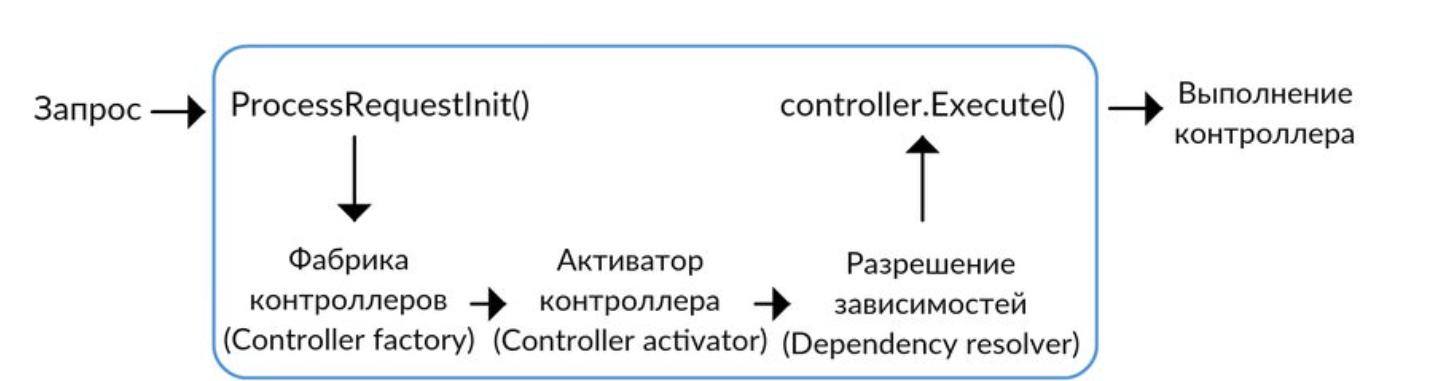
MVC использует еще один объект — активатор контроллеров (Controller Activator), который находит нужный контроллер и создает его.

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();

}

}

**Описание кода**

**1 задание**

WebApplication.CreateBuilder(args): Этот метод создаёт объект builder, который используется для настройки веб-приложения. Он принимает аргументы командной строки (через args) и настраивает приложение, включая такие компоненты, как сервер, логирование и другие службы.

builder.Services.AddRazorPages(): Этот метод добавляет поддержку Razor Pages в приложение. Razor Pages — это упрощённая модель для создания веб-страниц без использования полноценного контроллера и модели.

builder.Services.AddControllersWithViews(): Этот метод добавляет поддержку MVC-контроллеров и представлений (Views) в приложение. Это необходимо, если вы хотите использовать контроллеры для обработки запросов и возвращать представления (например, HTML).

builder.Build(): Этот метод создаёт и настраивает экземпляр веб-приложения на основе конфигураций, заданных ранее. Теперь объект app будет использоваться для добавления различных middleware-компонентов и настройки маршрутизации.

app.UseHttpsRedirection(): Этот middleware перенаправляет все HTTP-запросы на HTTPS, что обеспечивает безопасное соединение между клиентом и сервером.

app.UseStaticFiles(): Этот middleware разрешает обслуживать статические файлы (например, CSS, JavaScript, изображения) из папки wwwroot или других каталогов. Если запрос клиента направлен на статический ресурс (например, CSS-файл), этот middleware обслуживает его напрямую, без необходимости задействовать контроллер.

app.UseRouting(): Этот middleware настраивает обработку маршрутов для приложения. Он сопоставляет входящие HTTP-запросы с определёнными маршрутами в приложении, чтобы направлять их в соответствующие контроллеры и действия.

app.UseAuthorization(): Этот middleware отвечает за проверку авторизации пользователя. Он используется для проверки, имеет ли пользователь необходимые права доступа для выполнения определённых действий в приложении.

app.MapControllerRoute: Этот метод настраивает маршрутизацию для контроллеров по схеме MVC. Здесь указывается, что по умолчанию запрос будет направляться к контроллеру Home и действию Index, если другие маршруты не определены.

name: "default": Это имя маршрута, оно используется для идентификации маршрута, если необходимо.

pattern: "{controller=Home}/{action=Index}/{id?}": Это шаблон маршрута:

{controller=Home}: По умолчанию используется контроллер Home, если не указан другой контроллер в запросе.

{action=Index}: По умолчанию вызывается действие Index, если не указано другое действие.

{id?}: Это необязательный параметр id, который может быть передан в URL. Если он не передан, приложение не выдаст ошибку.

Пример запроса к такому маршруту: https://localhost:5001/Home/Index/5, Контроллер: Home

Действие: Index

Параметр id: 5

app.Run(): Этот метод запускает веб-приложение и начинает прослушивание входящих HTTP-запросов. После этого сервер начинает обрабатывать запросы по указанным маршрутам.

**2 задание**

@{

ViewData["Title"] = "Home Page";

}

Внутри фигурных скобок @{ } можно использовать код на языке C#. В данном случае этот блок используется для установки значения для ViewData.

ViewData["Title"] — Установка данных для представления

ViewData — это словарь, который используется для передачи данных из контроллера в представление. Он позволяет передать динамическую информацию, например, заголовок страницы, описание или любые другие данные.

Этот заголовок, как правило, отображается в элементе <title> в HTML-документе или используется в других местах страницы для динамического изменения контента.

public IActionResult Index()

{

return Redirect("/index.html");

}

Этот метод отвечает на запросы по маршруту Home/Index (или, в зависимости от конфигурации маршрутов, может быть доступен также по корневому маршруту /).

Основное назначение этого метода — перенаправить пользователя на статическую HTML-страницу, которая называется index.html.

При вызове этого метода происходит HTTP-перенаправление на другой URL, указанный в Redirect(). В данном случае это /index.html.

HTTP-редирект отправляет клиенту (обычно браузеру) ответ с кодом состояния 302 (или 301, если установлен постоянный редирект), что говорит клиенту о том, что запрашиваемый ресурс находится по новому адресу.

Пример: Если пользователь заходит на /Home/Index, он будет перенаправлен на /index.html, и браузер выполнит новый запрос на этот URL.

public IActionResult Privacy()

{

return View();

}

Этот метод обрабатывает запросы к странице политики конфиденциальности. Обычно это страница, на которой описывается, как приложение обрабатывает данные пользователей.

Метод возвращает результат View(), что означает, что он будет искать представление, связанное с этим действием.

По умолчанию ASP.NET Core будет искать файл Privacy.cshtml в папке Views/Home (или в другом месте, если определена другая конфигурация).

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public IActionResult Error()

{

return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });

}

Этот метод обрабатывает ошибки в приложении и возвращает страницу с сообщением об ошибке. Он может быть вызван, когда происходит исключение или ошибка в приложении.

[ResponseCache(...)]: Этот атрибут управляет кешированием ответа. В данном случае он отключает кеширование (Duration = 0, NoStore = true), что означает, что ответы не будут сохранены. Это полезно, чтобы пользователи не видели устаревшие страницы ошибок.

Метод создаёт новый экземпляр модели ErrorViewModel и передаёт его в представление.

RequestId в ErrorViewModel может быть использован для отслеживания идентификатора запроса, что полезно при отладке, чтобы узнать, какой именно запрос вызвал ошибку.

private readonly ILogger<HomeController> \_logger;

public HomeController(ILogger<HomeController> logger)

{

\_logger = logger;

}

Контроллер принимает через конструктор параметр ILogger<HomeController>, который используется для логирования. Логирование полезно для отслеживания работы приложения и отладки.

\_logger — это приватное поле, которое хранит объект для логирования.

**3 задание**

public IActionResult One()

{

ViewBag.Message = "Start/One";

return View();

}

Этот метод обрабатывает запросы к маршруту Start/One.

В методе устанавливается значение для ViewBag.Message, которое будет использоваться в представлении.

ViewBag — это динамическое свойство, которое позволяет передавать данные из контроллера в представление без необходимости создания модели. В данном случае оно получает значение "Start/One".

Метод возвращает View(), что означает, что будет возвращено представление для этого действия. По умолчанию будет искать One.cshtml в папке Views/Start.

ViewBag — это удобный способ передачи данных из контроллера в представление. Он является динамическим и позволяет вам передавать значения, которые могут быть использованы в вашем представлении для отображения информации пользователю.

<h1>@ViewBag.Message</h1>

 Эта строка отображает значение, сохраненное в ViewBag.Message в вашем контроллере.

 Когда пользователь переходит на страницу Start/One, ViewBag.Message было установлено на "Start/One" в методе One() контроллера.

 Таким образом, здесь на странице отобразится заголовок Start/One.

 Когда пользователь делает запрос по адресу /Start/One, ASP.NET Core маршрутизирует этот запрос к методу One() в StartController.

 Внутри метода One():

Устанавливается значение для ViewBag.Message.

Возвращается представление One, которое в вашем случае — это файл One.cshtml.

 Затем ASP.NET Core обрабатывает файл One.cshtml:

Он использует значение из ViewBag.Message и заголовок из ViewData["Title"], чтобы отобразить соответствующий контент.

GET /Start/Index — Вызывает метод Index контроллера StartController. Ожидается ответ с HTML-страницей, которая выводит текст "Start/Index" с ссылками на One, Two, Three.

GET /Start/One, Two, Three — Каждая из этих страниц вызывает соответствующие методы контроллера StartController (One, Two, Three). На этих страницах также должен выводиться текст с названиями этих страниц (например, "Start/One" и т. д.).

GET https://www.belstu.by/ HTTP/1.1

Это стандартный GET-запрос для загрузки главной страницы сайта БГТУ. Он возвращает HTML-документ (содержимое страницы). Статус ответа обычно 200 OK, если страница доступна, и содержит данные в формате HTML.

Пояснение заголовков:

Host: определяет хост, к которому направляется запрос. В данном случае это www.belstu.by.

User-Agent: содержит информацию о клиенте (например, браузере или инструменте), который делает запрос.

Accept: сообщает серверу, какой формат ответа ожидает клиент (например, HTML, JSON, XML и т.д.).

Connection: keep-alive: указывает на использование постоянного соединения между клиентом и сервером.

**Request Headers (заголовки запроса):**

**Host**: указывает домен и порт, к которому обращается запрос.

**User-Agent**: информация о клиенте (браузере), который делает запрос.

**Accept**: форматы данных, которые клиент готов принять (например, текст/html).

**Referer**: адрес страницы, с которой был сделан запрос (если это было переход по ссылке).

**Response Headers (заголовки ответа):**

**Content-Type**: тип содержимого ответа (например, text/html для HTML-страниц, application/json для JSON-ответов).

**Content-Length**: размер тела ответа в байтах.

**Cache-Control**: политика кэширования данных (может указывать, можно ли кэшировать данные и на какой срок).

**Server**: информация о сервере, который отвечает на запрос (например, Kestrel, IIS).

**Status**: статус ответа (например, 200 OK, 404 Not Found).

Анализ статусов HTTP:

**200 OK**: успешный ответ. Сервер обработал запрос, и данные были переданы.

**404 Not Found**: страница или ресурс не найдены. Сервер не смог найти запрашиваемый ресурс.

**500 Internal Server Error**: ошибка на стороне сервера. Это может быть ошибка в коде приложения или ошибка конфигурации.

**Задание 4**

public IActionResult S200()

{

return Ok("Status: 200 OK");

}

Этот метод возвращает статус 200 OK.

:

Метод использует метод Ok(), который является вспомогательным методом в контроллере, возвращающим статус 200 и текстовое сообщение "Status: 200 OK".

Это говорит клиенту, что запрос выполнен успешно, и данные (в данном случае текст) возвращаются.

public IActionResult S300()

{

return Redirect("https://www.example.com");

}

Этот метод возвращает статус 300, который обычно означает, что ресурс был перемещен.

Метод использует Redirect(), который отправляет клиента на новый URL (в данном случае на https://www.example.com).

Браузер автоматически перейдет по этому URL, и статус ответа будет 302 (перемещен временно) по умолчанию.

public IActionResult S500()

{

try

{

int zero = 0;

int result = 10 / zero;

return Ok(result);

}

catch (DivideByZeroException)

{

return StatusCode(500, "Status: 500 Internal Server Error");

}

}

Этот метод моделирует ситуацию, при которой происходит ошибка, возвращая статус 500 Internal Server Error.

В блоке try выполняется деление на ноль, что вызывает исключение DivideByZeroException.

Когда исключение выбрасывается, управление переходит в блок catch, где возвращается статус 500 с сообщением "Status: 500 Internal Server Error".

Это демонстрирует, как можно обрабатывать ошибки и возвращать соответствующий статус код при возникновении исключений.

**Задание 5 – передача параметров**

Статические параметры

Вы можете определить параметры в маршруте, используя специальные символы, такие как {parameterName}.

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

В этом случае id является параметром, который будет передан в действие контроллера.

public IActionResult Index(int id) { // Логика обработки с использованием id }

Каждое из представлений, которое вызывается в методах контроллера, может использовать переданные данные. Например, в представлении можно использовать ViewBag.Id, чтобы отобразить значение параметра на веб-странице

Параметры из строки запроса (query string)

Вы можете передавать параметры через строку запроса. Это делается с помощью ? после основного URL, за которым следуют параметры в формате key=value.

public IActionResult Search(string query)

{

// Логика поиска

}

Параметры в теле запроса

Для POST-запросов и других типов запросов (например, PUT, PATCH) данные часто передаются в теле запроса. В ASP.NET Core вы можете использовать модели для автоматического связывания данных.

public class UserModel

{

public string Name { get; set; }

public int Age { get; set; }

}

[HttpPost]

public IActionResult CreateUser([FromBody] UserModel user)

{

// Логика обработки user

}

Передача параметров с использованием атрибутов

Вы можете использовать атрибуты, чтобы явно указать, откуда должны извлекаться параметры:

[FromRoute]: Указывает, что параметр должен быть извлечен из маршрута.

[FromQuery]: Указывает, что параметр должен быть извлечен из строки запроса.

[FromBody]: Указывает, что параметр должен быть извлечен из тела запроса.

[FromHeader]: Указывает, что параметр должен быть извлечен из заголовка запроса.

public IActionResult GetUser([FromRoute] int id, [FromHeader] string token)

{

// Логика получения пользователя по id и токену из заголовка

}

### 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.

**Разбор кода:**

1. Контроллер TMResearch

Этот класс унаследован от Controller, что позволяет ему обрабатывать HTTP-запросы и возвращать HTTP-ответы.

Методы контроллера используются для сопоставления различных маршрутов (URI) с логикой обработки запросов.

Методы контроллера (IActionResult) предназначены для возвращения данных в ответ на HTTP-запросы. В этом примере каждый метод возвращает простое текстовое сообщение с помощью метода Content.

Метод Content возвращает простой текстовый ответ. Он используется для отправки простых сообщений клиенту в виде текстового контента без сложной логики обработки.

В ASP.NET Core маршруты для контроллеров можно настроить в файле Startup.cs или с помощью атрибутов маршрутов.

В данном случае маршруты не определены явным образом в атрибутах методов, что предполагает использование стандартной конфигурации маршрутов или другой настройки в приложении.

Контроллер принимает запросы с различными параметрами и маршрутами. Если маршрут запроса совпадает с одним из описанных маршрутов, вызовется соответствующий метод.

Если запрос не соответствует ни одному из перечисленных маршрутов, будет вызван метод MXX().

Принцип работы контроллера TMResearch в ASP.NET Core MVC основан на следующей последовательности шагов:

доступны по URI, начинающимся с /it.

1. Принципы маршрутизации по атрибутам

Определение маршрутов прямо в коде: Маршрутизация по атрибутам позволяет указать маршруты непосредственно в коде с помощью атрибутов, например, [Route], [HttpGet], [HttpPost] и других. Это делает код более читаемым и упрощает управление маршрутами.

Контроллер и методы: Маршруты могут быть заданы как для контроллера целиком (общий префикс маршрута), так и для отдельных методов действий.

2. Приоритет маршрутов

Конкретные маршруты имеют приоритет: Если несколько маршрутов могут совпадать с URI, более конкретный маршрут будет выбран. Например, если есть маршруты /it/{id} и /it/static, то запрос /it/static будет обработан вторым маршрутом.

Порядок определения: Порядок определения маршрутов в методах контроллера важен, когда маршруты пересекаются.

3. Параметры маршрута

Типы параметров: Параметры маршрута можно ограничивать по типу с помощью шаблонов, например, {id:int}, {name:alpha}, {price:float}.

Проверка значений: Атрибуты маршрутов поддерживают ограничения, такие как min, max, minlength, maxlength, что помогает фильтровать и проверять значения параметров.

Регулярные выражения: Параметры можно проверять с помощью регулярных выражений, например, {email:regex(^\\S+@\\S+\\.\\S+$)}.

4. HTTP-методы

Явное указание HTTP-методов: Маршруты могут быть ограничены конкретными методами HTTP с помощью атрибутов, таких как [HttpGet], [HttpPost], [HttpPut], [HttpDelete].

Множественные методы для одного действия: Один метод контроллера может поддерживать несколько HTTP-методов, например, [HttpGet] и [HttpPost], что позволяет использовать один метод для обработки различных типов запросов.

5. Маршрутные параметры и их особенности

Обязательные и необязательные параметры: Параметры могут быть обязательными (например, /it/{id}) или необязательными (например, /it/{id?}). Необязательные параметры должны иметь значение по умолчанию или быть проверены на null.

Преобразование типов: ASP.NET Core автоматически преобразует параметры маршрута в указанный тип (например, int, bool). Если преобразование невозможно, запрос не будет обработан.

6. Гибкость маршрутов

Сложные маршруты: Маршруты можно комбинировать с различными ограничениями и спецификаторами. Например, маршрут /it/{id:int:min(10):max(100)} принимает только целые числа от 10 до 100.

Использование нескольких маршрутов для одного метода: С помощью атрибутов [Route("path1")], [Route("path2")] можно сделать так, чтобы один метод обрабатывал запросы по нескольким различным маршрутам.

7. Маршруты с областями (Areas)

При использовании областей (Areas) маршруты могут включать в себя префиксы, которые добавляют дополнительный уровень организации URI. Например, [Area("Admin")] добавляет Admin в URI перед маршрутом контроллера.

9. Комбинирование маршрутов и методов

Атрибут [Route] на уровне контроллера: Добавляет общий префикс ко всем методам контроллера.

Комбинирование атрибутов: Использование [HttpGet("path")] или [Route("path")] позволяет комбинировать маршрут и метод в одном атрибуте.

10. Примеры распространенных ошибок

Некорректный шаблон маршрута: Ошибки в определении шаблонов (например, неправильно экранированные символы) могут привести к невозможности обработки запросов.

Отсутствие метода для обработки: Если запрос приходит на маршрут, для которого нет метода, будет возвращена ошибка 404.

Метод M04: Параметр n должен быть целым числом (int), а параметр str - строкой без дополнительных ограничений.

Метод M05: Параметр b должен быть логическим (bool), а параметр letters должен содержать только буквенные символы.

Метод M06: Параметр f должен быть числом с плавающей запятой (float), а параметр str должен быть строкой длиной от 3 до 5 символов включительно.

Метод M07: Параметр letters должен содержать только буквенные символы с длиной от 3 до 4 символов, а параметр n должен быть целым числом в диапазоне от 100 до 200 включительно.

Метод M08: Параметр mail должен соответствовать формату адреса электронной почты по заданному регулярному выражению.

### 5 Лабораторная

В ASP.NET MVC общий макет (layout) — это шаблон, который используется для создания общего интерфейса и структуры страниц приложения. Макет служит основной «оболочкой» для представлений, позволяя им наследовать общий стиль, меню, заголовок, подвал и другие элементы, которые должны быть одинаковыми на разных страницах.

Обычно макетный файл расположен в папке Views/Shared и имеет название Layout.cshtml по умолчанию.

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. В итоге мы сможем легко установить для всех представлений веб-приложения единообразный стиль оформления.

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:

1. **Будет 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.

Файл \_ViewStart.cshtml (обязательное название) в ASP.NET MVC представляет собой специальное представление, которое выполняется **до рендеринга каждого представления** и позволяет задать общие настройки для всех представлений приложения, такие как Layout.

Файл \_ViewImports.cshtml в ASP.NET MVC и ASP.NET Core MVC используется для **подключения пространств имен, вспомогательных функций и тег-хелперов**, которые будут применяться ко всем представлениям в определенной папке и ее подпапках.

* \_**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.

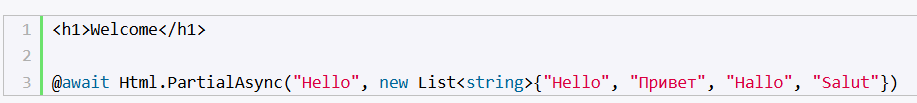
За рендеринг частичных представлений отвечает объект **PartialViewResult**, который возвращается методом **PartialView()**. Этот метод имеет четыре версии:

* PartialView(): для генерации ответа используется представление, которое по имени совпадает с вызывающим методом
* PartialView(string? viewName): в метод передается имя представления, что позволяет переопределить используемое по умолчанию представление
* PartialView(object? model): передает в представление данные в виде объекта model
* PartialView(string? viewName, object? model): переопределяет имя представления и передает в него данные в виде объекта model

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>**. В качестве параметра в метод передается имя представления.

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.

Одна из перегруженных версий методов Html.PartialAsync / Html.RenderPartialAsync позволяет передать модель в частичное представление. В итоге у нас получится стандартное строго типизированное представление. Например, в качестве второго параметра список строк:



* 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.

**Частичные представления**

Что такое частичное представление в ASP.NET MVC, и в каких случаях вы бы его использовали?

Частичное представление (Partial View) — это компонент представления, который может быть переиспользован в разных местах приложения. Оно не включает в себя полный HTML-документ, а содержит только часть разметки, которая может быть внедрена в другие представления. Частичные представления удобны для разделения сложных интерфейсов на более простые и удобные для поддержки компоненты.

Когда использовать:

Когда одна и та же часть интерфейса (например, форма ввода, карточка продукта) повторяется на нескольких страницах.

Для упрощения поддержки кода: изменение в одном частичном представлении автоматически отразится во всех местах, где оно используется.

Для управления большими представлениями, разбивая их на более мелкие логические части.

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.

**Способы вставки**

ViewBag — это динамическая структура, которая позволяет передавать данные из контроллера в представление без необходимости создания модели. Она используется для передачи небольших объемов данных, таких как сообщения об ошибках, результаты вычислений и другие временные данные.

Методы действия контроллера - Методы действия контроллера (action methods) — это методы, которые обрабатывают HTTP-запросы и возвращают результат, который может быть представлением (HTML), данными (JSON) или перенаправлением на другой URL. В ASP.NET MVC контроллеры представляют собой классы, содержащие эти методы.

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.

**Частичные представления (Partial Views)** Частичные представления позволяют вам разбить интерфейс на переиспользуемые компоненты. Это может быть удобно для создания одинаковых элементов на разных страницах. HTML Helper для частичных представлений:

* Html.Partial(): Используется для рендеринга частичного представления.
* Html.RenderPartial(): Похож на Html.Partial(), но выводит контент напрямую, что может улучшить производительность.

**Разбор кода:**

private bool IsNumber(string value) { return float.TryParse(value, out \_); } тот приватный метод проверяет, может ли строка быть преобразована в float. Он использует float.TryParse, который пытается преобразовать строку в число с плавающей запятой и возвращает true, если это удалось, и false в противном случае. Это полезно для проверки ввода пользователя.

**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.

 **Параметры x и y**: Это значения, которые пользователь вводит в форму. Они автоматически привязываются к параметрам метода.

 **Проверка корректности ввода**: Если одно из значений не является числом, устанавливается сообщение об ошибке и возвращается представление с теми же данными.

 **Вычисление результата**: Если оба значения корректны, они преобразуются в числа (с помощью float.Parse) и складываются. Результат сохраняется в ViewBag.Result.

 **Возврат представления**: После обработки запроса возвращается то же представление, чтобы пользователь мог увидеть результат.

Calc

 **Layout = null;**: Эта строка указывает, что данное представление не использует общий макет (layout). Обычно в ASP.NET MVC указывается общий шаблон для страниц, но здесь мы решили его не использовать.

 **float? x = ViewBag.x; и float? y = ViewBag.y;**: Эти строки присваивают переменным x и y значения из ViewBag. ViewBag – это динамическая структура, позволяющая передавать данные между контроллером и представлением. Знак вопроса после float обозначает, что переменные могут принимать значение null.

 **string press = (string)ViewBag.Press;**: Получает символ операции (например, +, -, \*, /), который был передан из контроллера.

 **string result = Convert.ToString(ViewBag.Result ?? "err");**: Присваивает переменной result значение из ViewBag.Result. Если ViewBag.Result равно null, то будет возвращена строка "err". Это используется для отображения результата вычисления или сообщения об ошибке.

**@await Html.PartialAsync("\_CalcLink")**: Здесь вызывается частичное представление (Partial View) с именем \_CalcLink. Этот метод позволяет разделить код на части и повторно использовать его, что делает код более организованным и удобным для поддержки.

 **@if (!string.IsNullOrWhiteSpace(press))**: Проверяет, содержит ли переменная press не пустую или не состоящую только из пробелов строку. Если это так, значит, была выбрана операция, и форма должна отображаться.

 **@using (Html.BeginRouteForm((press, "Calc", FormMethod.Post)))**: Начинает блок формы. Метод BeginRouteForm создает форму, которая отправляет POST-запрос на маршрут Calc с указанием нажатой операции (например, +, - и т. д.).

 **<p>**: Тег параграфа, в который помещаются элементы формы.

 **@await Html.PartialAsync("\_Calc")**: Вставляет частичное представление \_Calc, которое, вероятно, содержит элементы для ввода чисел x и y.

 **<input type="text" disabled value="@result" placeholder="Результат" />**: Создает текстовое поле, которое отображает результат вычисления. Поле disabled делает его недоступным для редактирования.

 **<input type="submit" name="press" value="@press" />**: Кнопка отправки формы. Значение кнопки соответствует операции, выбранной пользователем (например, +, - и т. д.).

 **<p>**: Второй параграф для отображения сообщения об ошибке, если таковое имеется.

#### 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:

html

Копировать код

<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. Это особенно полезно, когда однотипные данные (например, даты или денежные суммы) встречаются в разных местах приложения, и требуется единообразное отображение.

**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.

**Хелперы в ASP.NET MVC**

**HTML-хелперы** — это методы, которые помогают генерировать HTML-разметку на страницах Razor. Они упрощают написание кода, избавляя разработчика от необходимости вручную создавать HTML-теги, особенно когда они зависят от данных модели.

**Виды HTML-хелперов**

1. **Встроенные HTML-хелперы**:  
   ASP.NET MVC предоставляет набор встроенных хелперов, которые покрывают основные нужды для создания HTML-элементов:
   * **@Html.TextBoxFor()** и **@Html.TextBox()** — создают текстовое поле.
   * **@Html.DropDownListFor()** — создает выпадающий список.
   * **@Html.CheckBoxFor()** — генерирует чекбокс для булевых значений.
   * **@Html.LabelFor()** — создает метку для элемента формы.

Эти хелперы обеспечивают типобезопасность и возможность привязки к данным модели, что упрощает работу с формами и делает код более компактным.

**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>

### 6 Лабораторная

**Идентификация** — это процесс, с помощью которого система определяет, кто пытается получить доступ. В процессе идентификации пользователь сообщает системе свою личность, обычно с использованием имени пользователя (логина), номера или другого уникального идентификатора.

**Аутентификация** — это процесс проверки подлинности пользователя или системы, когда предоставленная информация сверяется с тем, что сохранено в системе. Аутентификация отвечает на вопрос: "Действительно ли вы тот, за кого себя выдаёте?"

**Авторизация** — это процесс, который решает, какие действия или ресурсы доступны пользователю после того, как его личность была подтверждена (аутентифицирована). Авторизация проверяет, какие права или доступ имеет данный пользователь в системе. Авторизация отвечает на вопрос: "Что вам разрешено делать?"

builder.Services.AddDefaultIdentity<User>(options => options.SignIn.RequireConfirmedAccount = false)

.AddRoles<IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>();

**AddDefaultIdentity**<User>: Этот метод настраивает систему Identity с типом User, который в вашем случае представляет модель пользователя. Это позволяет работать с механизмом аутентификации и управления пользователями.

Параметр options.SignIn.RequireConfirmedAccount = false указывает, что подтверждение аккаунта (например, через email) не требуется для входа в систему.

AddRoles<IdentityRole>: Здесь добавляется поддержка ролей. Модель роли будет использоваться как IdentityRole. Это позволяет назначать пользователям роли (например, "Admin", "User" и т.д.).

AddEntityFrameworkStores<ApplicationDbContext>: Этот метод указывает, что для хранения данных о пользователях и ролях будет использоваться Entity Framework, а именно — контекст базы данных ApplicationDbContext. Контекст базы данных должен быть настроен для работы с Identity (например, с таблицами для пользователей, ролей и т.д.).

app.UseAuthentication();

app.UseAuthorization();

**UseAuthentication**(): Этот метод настраивает middleware для обработки аутентификации, т.е. для проверки, кто вошел в систему. Это нужно для того, чтобы ASP.NET Core мог аутентифицировать пользователей в приложении.

**UseAuthorization**(): Этот метод добавляет middleware для авторизации, который проверяет, имеет ли пользователь права для выполнения определенной операции или доступа к ресурсу (например, имеет ли он нужную роль или разрешение).

|  |
| --- |
| using (var scope = app.Services.CreateScope())  {  var services = scope.ServiceProvider;  try  {  var userManager = services.GetRequiredService<UserManager<User>>();  var roleManager = services.GetRequiredService<RoleManager<IdentityRole>>();  await RoleInitializer.InitializeAsync(userManager, roleManager);  }  catch (Exception ex)  {  Console.WriteLine($"Ошибка инициализации ролей: {ex.Message}");  }  } |

CreateScope(): Создается новый scope (область действия) для внедрения зависимостей. Это нужно, чтобы использовать сервисы, такие как UserManager и RoleManager, в асинхронном контексте.

GetRequiredService<UserManager<User>>() и GetRequiredService<RoleManager<IdentityRole>>(): Эти методы получают необходимые сервисы для управления пользователями и ролями. UserManager используется для работы с пользователями, а RoleManager — для работы с ролями.

**RoleInitializer**.InitializeAsync(userManager, roleManager): Этот вызов предполагает, что в коде есть класс RoleInitializer, который инициализирует начальные роли (например, "Admin", "User") в базе данных, если они еще не были созданы.

app.UseStatusCodePages(context => { // middleware перехватывающий ошибки

if (context.HttpContext.Response.StatusCode == 404)

{

context.HttpContext.Response.Redirect("/Administrator/Error");

}

return Task.CompletedTask;

});

Если сервер возвращает статус 404 (страница не найдена), то происходит перенаправление на страницу "/Administrator/Error".

Viewes

Home – Index

var userName = User.Identity?.Name: Получаем имя текущего аутентифицированного пользователя (если он есть).

var isAuthenticated = User.Identity.IsAuthenticated: Проверка, аутентифицирован ли текущий пользователь.

var roles = new List<string>(): Создаем список для хранения ролей пользователя.

Если пользователь аутентифицирован, то с помощью UserManager.GetUserAsync(User) мы получаем объект пользователя и извлекаем его роли через UserManager.GetRolesAsync(user)

**@if (!User.IsInRole("Administrator"))**: Если пользователь не является администратором,

@Url.Action(...): Это синтаксис Razor для генерации URL для действия контроллера. Он создает URL для действия, переданного в аргументы.

"Register" — имя действия (метода контроллера).

"Administrator" — имя контроллера.

new { returnController = "Home", returnAction = "Index" } — это анонимный объект, передающий параметры в URL, которые обычно используются для возврата на предыдущую страницу после регистрации или авторизации (например, на страницу Home/Index).

@if (condition): Эта конструкция Razor используется для условного рендеринга элементов в шаблоне. Внутри условных блоков можно вставлять HTML и C# код.

@Url.Action(...): Это метод Razor, который генерирует URL для действия в контроллере. Это удобный способ создать ссылку на действие, не беспокоясь о ручном указании пути.

User.IsInRole("RoleName"): Метод IsInRole используется для проверки, имеет ли текущий пользователь указанную роль.

**Action** — это метод в контроллере, который обрабатывает HTTP-запросы и возвращает результат, такой как HTML-страница, JSON-ответ или редирект. Давайте разберем подробнее, что такое Action в ASP.NET Core.

Action (или метод действия) — это метод в контроллере, который выполняет определенные операции в ответ на HTTP-запрос. Каждый метод контроллера, который помечен атрибутом (например, [HttpGet] или [HttpPost]), или любой публичный метод контроллера, который соответствует маршруту, будет считаться Action.

**Url.Action** — это метод в Razor, который генерирует URL для указанного действия (Action) в указанном контроллере. Когда вы вызываете этот метод, вы фактически создаете URL, который будет указывать на конкретное действие в контроллере.

@model ChangeRoleViewModel

Это указание на модель, с которой будет работать данный Razor-шаблон. В данном случае это класс ChangeRoleViewModel, который должен содержать необходимые данные для отображения в представлении, такие как информация о пользователе и доступные роли.

asp-action="Assign": Указывает действие, которое будет обрабатывать запрос (в данном случае Assign).

method="post": Указывает, что форма будет отправляться с использованием POST-запроса.

<div asp-validation-summary="ModelOnly" class="alert alert-danger"></div>

Этот элемент выводит ошибки валидации, если они имеются. Атрибут asp-validation-summary="ModelOnly" означает, что будут отображаться ошибки, связанные с моделью. Они будут помещены в блок с классом alert alert-danger, что создает стиль предупреждения.

@foreach (IdentityRole role in Model.AllRoles): Это цикл, который перебирает все роли, переданные в модель (Model.AllRoles).

asp-for="returnAction": Генерирует скрытое поле для действия, куда будет выполнен редирект после изменения ролей.

<a asp-controller="@Model.returnController" asp-action="@Model.returnAction" class="btn btn-primary">Назад</a>: Это ссылка, которая возвращает пользователя на предыдущую страницу, указанную в модели. С помощью asp-controller и asp-action задаются контроллер и действие для возвращения.

<span asp-validation-for="CurrentPassword" class="text-danger"></span>

мы видим использование ASP.NET Core Razor для вывода сообщения об ошибке валидации для поля модели, с именем CurrentPassword. Давайте разберем, что происходит:

1. asp-validation-for="CurrentPassword":

asp-validation-for — это атрибут в Razor, который используется для привязки сообщения об ошибке валидации к конкретному полю формы. В данном случае он связан с полем CurrentPassword модели.

Когда форма отправляется на сервер и происходит валидация, если для поля CurrentPassword возникнет ошибка (например, если введенный пароль не соответствует текущему паролю), то это сообщение будет отображено в этом элементе.

CurrentPassword — это имя свойства модели, для которого нужно отобразить сообщение об ошибке. Например, если у вас есть модель с таким свойством:

Required(ErrorMessage = "Текущий пароль обязателен.")]

[DataType(DataType.Password)]

Если пользователь не введет текущий пароль и форма будет отправлена, то ASP.NET Core сгенерирует ошибку для поля CurrentPassword, и она будет отображена внутри тега <span>, где будет указан текст ошибки, например: "Текущий пароль обязателен.".

мы видим объявление класса UserManager<TUser>, который является частью ASP.NET Core Identity и представляет собой класс для управления пользователями. Давайте разберем все компоненты этого объявления:

1. public class **UserManager**<TUser>:

UserManager<TUser> — это класс, предназначенный для управления пользователями в ASP.NET Core Identity. Он предоставляет методы для выполнения операций с пользователями, таких как создание, удаление, обновление и аутентификация пользователей.

TUser — это обобщенный тип, который представляет модель пользователя. Обычно это класс, который наследует от IdentityUser. Например, если вы используете стандартную модель пользователя, то тип TUser может быть IdentityUser. Однако вы можете создать собственную модель пользователя, расширяя IdentityUser, и указать эту модель как тип TUser.

**IdentityUser** — это стандартный класс, предоставляемый в ASP.NET Core Identity, который представляет собой базовую модель пользователя. Он содержит стандартные свойства, которые необходимы для аутентификации и управления пользователями в системе.

Давайте рассмотрим, что такое IdentityUser, какие у него свойства и как его можно расширить.

1. Что такое IdentityUser?

IdentityUser — это базовый класс, который предоставляется в ASP.NET Core Identity для работы с пользователями. Этот класс предоставляет стандартные свойства, которые могут быть использованы для хранения информации о пользователе, такие как имя пользователя, электронная почта, пароль и другие связанные данные.

Он является частью пространства имен Microsoft.AspNetCore.Identity и представляет собой тип, который используется при создании пользователей для системы аутентификации.

GetRolesAsync — это встроенный метод в ASP.NET Core Identity. Он является частью интерфейса IUserRoleStore<TUser>, который реализуется в стандартных хранилищах пользователей Identity, например, в UserManager<TUser>.

Когда вы используете UserManager<TUser> для работы с пользователями, метод GetRolesAsync доступен для получения ролей, назначенных пользователю. Обычно этот метод используется через UserManager, который инкапсулирует операции с ролями и пользователями, и упрощает взаимодействие с системой аутентификации и авторизации.

asp-route-id передает параметр маршрута в URL.

asp-route-returnController добавляет дополнительный параметр маршрута, который указывает, на какой контроллер нужно вернуться после выполнения действия.

Models

[Required]: Атрибут указывает, что поле обязательно для заполнения. Если пользователь не введет текущий пароль, появится ошибка с сообщением "Пароль обязателен для ввода".

[DataType(DataType.Password)]: Атрибут указывает, что это поле должно отображаться как поле для ввода пароля (с маскировкой символов).

[Display(Name = "Текущий пароль")]: Атрибут определяет отображаемое имя для этого поля. На странице будет выведено название "Текущий пароль".

[Compare("NewPassword", ErrorMessage = "Пароли не совпадают")]: Этот атрибут проверяет, чтобы значение в ConfirmNewPassword совпадало с значением в NewPassword. Если пароли не совпадают, будет показано сообщение об ошибке "Пароли не совпадают".

public string returnAction { get; set; }

public string returnController { get; set; }

Эти свойства могут быть использованы для хранения информации о том, к какому действию и контроллеру нужно вернуться после выполнения операции. Это обычно используется для перенаправления пользователя на страницу, с которой он пришел после изменения пароля.

Это список всех доступных ролей в системе. Он используется для отображения всех возможных ролей, которые можно назначить пользователю. Тип IdentityRole представляет роль в ASP.NET Identity.

Свойство ShowRequestId проверяет, существует ли RequestId (он не пустой и не null).

Это полезно, например, для вывода идентификатора запроса в сообщениях об ошибках или логах только в тех случаях, когда он действительно присутствует.

ILogger — это интерфейс, предоставляющий методы для записи логов (например, LogInformation, LogError, и т.д.).

Тип: SignInManager<User>

Назначение: Этот сервис управляет процессом входа пользователя в систему, а также предоставляет функции для аутентификации, такие как вход по паролю или вход через внешние сервисы (например, Google или Facebook).

SignInManager — это встроенный сервис для работы с процессом аутентификации пользователей в ASP.NET Core Identity.

UserManager — это основной сервис для работы с учетными записями пользователей в ASP.NET Core Identity.

RoleManager — это сервис для работы с ролями в ASP.NET Core Identity.

IdentityRole — это стандартная роль в ASP.NET Core Identity, представляющая роль пользователя (например, "Администратор", "Пользователь").

Метод AddToRoleAsync является частью класса UserManager<TUser> в ASP.NET Core Identity. Он используется для добавления пользователя в определённую роль или несколько ролей.

|  |
| --- |
| using Microsoft.AspNetCore.Identity;  using Microsoft.AspNetCore.Mvc;  using System.Threading.Tasks;  public class RoleInitializer : Controller  {  // Асинхронный метод для инициализации ролей и пользователя  public static async Task InitializeAsync(UserManager<User> userManager, RoleManager<IdentityRole> roleManager)  {  string adminEmail = "admin"; // Email администратора  string password = "Password12345!"; // Пароль для администратора  // Проверяем, существует ли роль "Administrator"  if (await roleManager.FindByNameAsync("Administrator") == null)  {  // Если роль не существует, создаем ее  await roleManager.CreateAsync(new IdentityRole("Administrator"));  }  // Проверяем, существует ли роль "User"  if (await roleManager.FindByNameAsync("User") == null)  {  // Если роль не существует, создаем ее  await roleManager.CreateAsync(new IdentityRole("User"));  }  // Проверяем, существует ли пользователь с email "admin"  if (await userManager.FindByNameAsync(adminEmail) == null)  {  // Если пользователь не существует, создаем нового администратора  User admin = new User { Email = adminEmail, UserName = adminEmail };  IdentityResult result = await userManager.CreateAsync(admin, password); // Создаем пользователя с паролем  // Если создание пользователя прошло успешно, добавляем его в роль "Administrator"  if (result.Succeeded)  {  await userManager.AddToRoleAsync(admin, "Administrator");  }  }  }  } |

Метод GetExternalAuthenticationSchemesAsync возвращает список всех доступных способов внешней аутентификации, например, через Google или Facebook. Если таких схем несколько, то переменная hasExternalLogins будет true.

Каждая ссылка (<a>) получает дополнительный класс через метод @ManageNavPages.[PageName]NavClass(ViewContext). Это позволяет динамически применить активный класс CSS в зависимости от текущей страницы. Таким образом, пользователь всегда будет видеть, на какой странице он находится.

@ManageNavPages.IndexNavClass(ViewContext): Это динамическое добавление CSS-класса для ссылки в зависимости от текущей страницы. Каждая страница может определить, является ли она активной, и отображать соответствующий стиль.

asp-page="./[Page]": Это директива, которая указывает на страницу Razor, на которую будет направлен пользователь при клике на ссылку. Например, при клике на "Profile" будет вызвана страница Index в контексте текущей Razor-страницы.

await SignInManager.GetExternalAuthenticationSchemesAsync(): Метод, который асинхронно получает список доступных внешних схем аутентификации.

Он получает пользователя с помощью метода \_userManager.GetUserAsync(User) и проверяет, есть ли у него пароль с помощью метода \_userManager.HasPasswordAsync(user). Если пароль не установлен, пользователя перенаправляют на страницу для установки пароля (./SetPassword).

\_userManager.GetUserAsync(User): Получает текущего пользователя по контексту запроса.

\_userManager.GetTwoFactorEnabledAsync(user): Проверяет, включена ли двухфакторная аутентификация для пользователя.

\_userManager.SetTwoFactorEnabledAsync(user, false): Отключает двухфакторную аутентификацию для пользователя, устанавливая флаг в false.

\_logger.LogInformation(...): Логирует информацию о том, что пользователь отключил 2FA.

StatusMessage: Содержит сообщение, которое будет отображено на странице после успешного выполнения операции.

Про библиотеку using Microsoft.AspNetCore.Identity

Регистрация как делается

|  |
| --- |
| [HttpGet]  public IActionResult Register(string returnController = "Home", string returnAction = "Index")  {  // Возвращает страницу регистрации с начальной моделью RegisterViewModel.  // Параметры returnController и returnAction используются для указания,  // куда перенаправить пользователя после успешной регистрации.  return View(new RegisterViewModel { returnAction = returnAction, returnController = returnController });  }  [HttpPost]  public async Task<IActionResult> Register(RegisterViewModel model)  {  // Проверяем, что модель прошла валидацию  if (ModelState.IsValid)  {  // Создаём нового пользователя с введёнными данными  User user = new User { Email = model.Email, UserName = model.Email, Year = model.Year };  // Пытаемся создать пользователя в базе данных  var result = await \_userManager.CreateAsync(user, model.Password);  // Если регистрация успешна  if (result.Succeeded)  {  // Добавляем пользователя в роль "User"  await \_userManager.AddToRoleAsync(user, "User");  // Выполняем вход пользователя в систему  await \_signInManager.SignInAsync(user, false);  // Перенаправляем пользователя на указанный returnAction и returnController  return RedirectToAction(model.returnAction, model.returnController);  }  else  {  // Если регистрация не удалась, добавляем ошибки в ModelState  foreach (var error in result.Errors)  {  ModelState.AddModelError(string.Empty, error.Description);  }  }  }  // Если модель не валидна или регистрация не удалась, возвращаем ту же форму  return View(model);  } |

Авторизация

|  |
| --- |
| [HttpPost]  public async Task<IActionResult> SignIn(SignInViewModel model)  {  // Проверяем, что модель прошла валидацию (например, все поля заполнены корректно)  if (ModelState.IsValid)  {  // Пытаемся выполнить вход пользователя  var result = await \_signInManager.PasswordSignInAsync(  model.Email, // Адрес электронной почты пользователя  model.Password, // Пароль, введённый пользователем  model.RememberMe, // Флаг, указывает, нужно ли запомнить пользователя  false); // Указывает, блокировать ли пользователя после неудачных попыток входа  // Если вход выполнен успешно  if (result.Succeeded)  {  // Перенаправляем пользователя на указанный returnAction и returnController  return RedirectToAction(model.returnAction, model.returnController);  }  else  {  // Если вход не удался, добавляем сообщение об ошибке в ModelState  ModelState.AddModelError("", "Неправильный логин и (или) пароль");  }  }  // Если модель не валидна или вход не удался, возвращаем ту же форму с ошибками  return View(model);  } |

### 7 Лабораторная

**Сессия** — это механизм сохранения данных, которые специфичны для определённого пользователя, на протяжении нескольких запросов. Она позволяет ассоциировать данные с конкретным пользователем на основе уникального идентификатора сессии (**Session ID**), который передаётся в браузер в виде cookie.

Жизненный цикл сессии

Когда пользователь впервые обращается к серверу, ASP.NET Core создаёт уникальный идентификатор сессии (Session ID).

Этот идентификатор отправляется клиенту через cookie (обычно с именем ASP.NET\_SessionId или другим настроенным именем).

Сессия используется для хранения данных между запросами.

HttpContext.Session.SetString("Mode", "Owner");

var mode = HttpContext.Session.GetString("Mode");

Данные сохраняются в серверном хранилище (например, в памяти, базе данных или распределённой системе).

На каждом последующем запросе клиент отправляет cookie с идентификатором сессии.

Сервер находит данные сессии по этому идентификатору и предоставляет их приложению.

Сессия завершается, если пользователь неактивен в течение определённого времени (тайм-аут).

После завершения данные сессии удаляются из хранилища.

Для использования сессий в ASP.NET Core нужно настроить middleware:

services.AddDistributedMemoryCache(); // Хранилище данных сессии в памяти

services.AddSession(options =>

{

options.IdleTimeout = TimeSpan.FromMinutes(30); // Тайм-аут сессии

options.Cookie.HttpOnly = true; // Защита от JavaScript-доступа

options.Cookie.IsEssential = true; // Для работы даже с GDPR-настройками

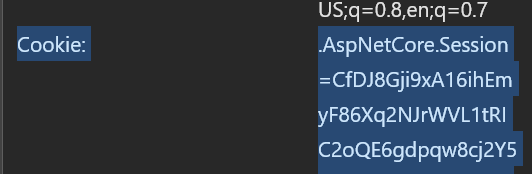
});

Как определяется сессия?

Основной способ определения сессии — это cookie, которое отправляется клиенту.

Если cookie не найдено (например, при первом запросе), создаётся новая сессия.

Идентификатор сессии (Session ID), как правило, передаётся в заголовке HTTP-запроса через cookie.



Как Session ID передаётся между клиентом и сервером?

Первый запрос клиента: Клиент не отправляет cookie, так как сессия ещё не создана.

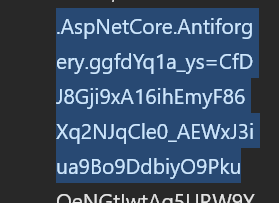
Сервер создаёт сессию и возвращает Session ID в ответе, задав cookie:

Set-Cookie: ASP.NET\_SessionId=abc123xyz456; path=/; HttpOnly

Последующие запросы:

Клиент автоматически включает cookie с Session ID в заголовке:

Cookie: ASP.NET\_SessionId=abc123xyz456



Антифальсификационные токены (Antiforgery Tokens) в ASP.NET Core используются для защиты веб-приложений от атак CSRF (Cross-Site Request Forgery), то есть межсайтовой подделки запросов.

Когда клиент отправляет запрос, браузер автоматически передаёт cookie, а также токен из формы. Сервер проверяет, что:

Значение токена в теле запроса совпадает с токеном из cookie.

Оба токена были созданы сервером для одной и той же сессии.

CSRF-токен — это уникальный токен, который сервер генерирует и проверяет для каждого защищённого запроса. Он предназначен для предотвращения CSRF-атак, поскольку злоумышленник не сможет предугадать и передать корректный токен.

Жизненный цикл CSRF-токена

При загрузке страницы сервер генерирует уникальный токен (обычно случайную строку) для текущей сессии или запроса.

Этот токен отправляется клиенту вместе с HTML-кодом страницы или через JavaScript.

В скрытом поле формы: Токен добавляется в каждую форму на странице.

Через cookie: В редких случаях токен может быть передан через cookie (но это менее безопасно).

Когда клиент отправляет POST, PUT или DELETE запрос (или любой другой потенциально опасный запрос), токен передаётся серверу вместе с данными формы.

Токен может быть передан:

В теле запроса (например, в скрытом поле формы).

В заголовке (например, с помощью JavaScript).

Проверка токена сервером:

Сервер проверяет:

Соответствует ли токен ожидаемому значению (например, токену, сохранённому в сессии или базе данных).

Связан ли токен с текущим пользователем/сессией.

Если токен отсутствует, недействителен или не совпадает с ожидаемым, сервер отклоняет запрос.

Завершение:

Если проверка токена успешна, запрос обрабатывается.

После успешной обработки токен может быть сброшен (например, если он одноразовый).

использоваться для сохранения каких-то временных данных, которые должны быть доступны, пока пользователь работает с приложением, и не требуют постоянного хранения.

Для хранения состояния сессии на сервере создается словарь или хеш-таблица, которая хранится в кэше и которая существует для всех запросов из одного браузера в течение некоторого времени. На клиенте хранится идентификатор сессии в куках. Этот идентификатор посылается на сервер с каждым запросом. Сервер использует этот идентификатор для извлечения нужных данных из сессии. Эти куки удаляются только при завершении сессии. Но если сервер получает куки, которые установлены уже для истекшей сессии, то для этих кук создается новая сессия.

Сервер хранит данные сессии в течение ограниченного промежутка времени после последнего запроса. По умолчанию этот промежуток равен 20 минутам, хотя его также можно изменить.

Следует учитывать, что данные сессии специфичны для одного браузера и не разделяются между браузерами. То есть для каждого браузера на одном компьютере будет создаваться свой набор данных.

