

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания
по выполнению лабораторных работ
по дисциплине «Сервис-ориентированные информационные
системы»

для студентов направления подготовки
09.04.02 «Информационные системы и технологии»
Профиль «Управление данными»

Ставрополь, 2025

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	3
ЛАБОРАТОРНАЯ РАБОТА 1. ПРОЕКТ НА ОСНОВЕ ASP.NET CORE	4
ЛАБОРАТОРНАЯ РАБОТА 2. ПРИЛОЖЕНИЕ ASP.NET CORE14	
ЛАБОРАТОРНАЯ РАБОТА 3. ФОРМИРОВАНИЕ ВЕБ-ОТВЕТОВ	31
ЛАБОРАТОРНАЯ РАБОТА 4. ПОЛУЧЕНИЕ ДАННЫХ ЗАПРОСА.....	37
ЛАБОРАТОРНАЯ РАБОТА 5. ОТПРАВКА ФОРМ	46
ЛАБОРАТОРНАЯ РАБОТА 6. ОТПРАВКА ФАЙЛОВ. ПЕРЕДАЧА ФАЙЛОВ НА СЕРВЕР	53
ЛАБОРАТОРНАЯ РАБОТА 7. СОЗДАНИЕ ПРОСТЕЙШЕГО API	65
ЛАБОРАТОРНАЯ РАБОТА 8. МЕХАНИЗМ DEPENDENCY INJECTION.....	83
ЛАБОРАТОРНАЯ РАБОТА 9. МАРШРУТИЗАЦИЯ	94
ЛАБОРАТОРНАЯ РАБОТА 10. ЛОГГИРОВАНИЕ.....	102
ПРИЛОЖЕНИЕ 1. ПРИМЕРНЫЕ ПРЕДМЕТНЫЕ ОБЛАСТИ ДЛЯ РЕАЛИЗАЦИИ СТУДЕНТАМИ В ЛАБОРАТОРНЫХ РАБОТАХ	112
СПИСОК ЛИТЕРАТУРЫ	113

ВВЕДЕНИЕ

1. Цели и задачи освоения дисциплины. Методические указания по дисциплине «Сервис-ориентированные информационные системы» для студентов направления 09.04.02 «Информационные системы и технологии». Пособие охватывает теоретические аспекты проектирования и разработки приложений для среды Интернет. Основное внимание уделяется доступной технологии разработки веб-приложений – ASP.NET Core. Методические указания предназначены для студентов, обладающих теоретическими знаниями в области проектирования приложений и практическими навыками программирования (предпочтительно языки C#, HTML, JavaScript). Цель методических указаний: сформировать у студентов целостный взгляд на современные тенденции в области web-программирования; обеспечить студентов обширным теоретическим материалом, достаточным для освоения методик разработки приложений, основанных на сервисной архитектуре; сформировать систему компетенций.

ЛАБОРАТОРНАЯ РАБОТА 1. ПРОЕКТ НА ОСНОВЕ ASP.NET CORE

1. Цель и задачи

Цель лабораторной работы: изучение основ построения проектов на основе ASP.NET Core.

Задачи лабораторной работы:

1. Изучить архитектуру и особенности ASP.NET Core.
2. Научиться создавать и управлять проектами на основе технологии ASP.NET Core.
3. Изучить основные компоненты и технологии ASP.NET Core.

2. Теоретическое обоснование

2.1. Основы ASP.NET Core

ASP.NET Core представляет технологию 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#.

История ASP.NET фактически началась с выходом первой версии .NET в начале 2002 года и с тех пор ASP.NET и .NET развивались параллельно: выход новой версии .NET знаменовал выход новой версии ASP.NET, поскольку ASP.NET работает поверх .NET. В то же время изначально ASP.NET была нацелена на работу исключительно в Windows на веб-сервере IIS (хотя благодаря проекту Mono приложения на ASP.NET можно было запускать и на Linux).

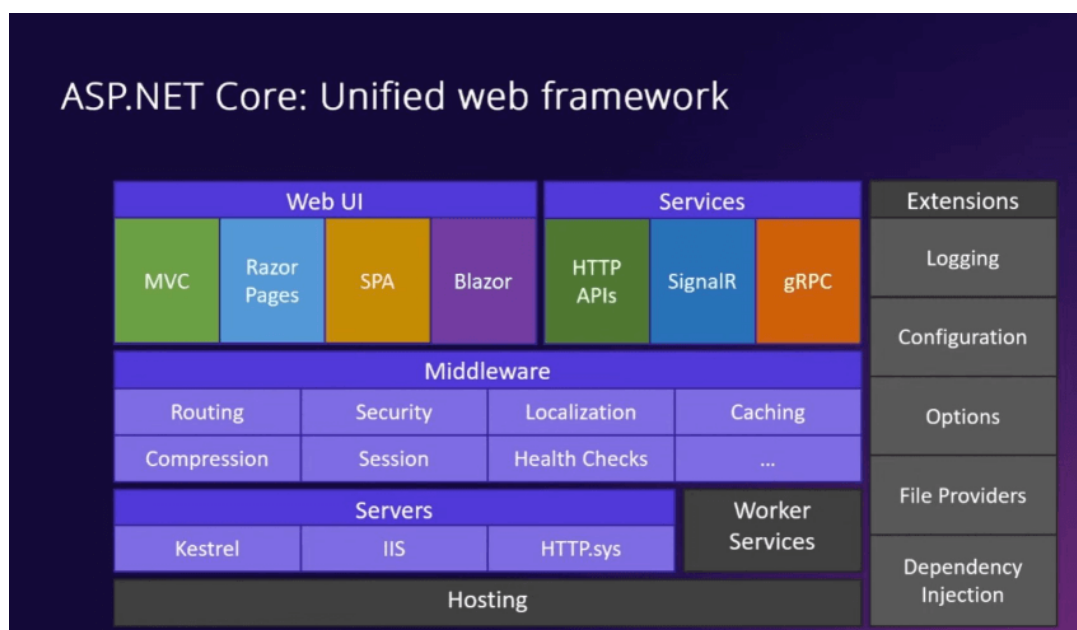
Но 2014 год ознаменовал большие перемены, фактически революцию в развитии платформы: компания Microsoft взяла курс на развитие ASP.NET как кроссплатформенной технологии, которая развивается как opensource-проект. Данное развитие платформы в дальнейшем получило название ASP.NET Core,

собственно как ее официально именуют Microsoft до сих пор. Первый релиз обновленной платформы увидел свет в июне 2016 года. Теперь она стала работать не только на Windows, но и на MacOS и Linux. Она стала более легковесной, модульной, ее стало проще конфигурировать, в общем, она стала больше отвечать требованиям текущего времени.

ASP.NET Core теперь полностью является opensource-фреймворком. Все исходные файлы фреймворка доступны на github в репозитории <https://github.com/dotnet/aspnetcore/>.

2.2. Архитектура и модели разработки

Текущую архитектуру платформы ASP.NET Core можно выразить следующим образом:



Архитектура ASP.NET Core в C# и .NET

На самом верхнем уровне располагаются различные модели взаимодействия с пользователем. Это технологии построения пользовательского интерфейса и обработки ввода пользователя, как MVC, Razor Pages, SPA (Single Page Application - одностраничные приложения с использованием Angular, React, Vue) и Balzor. Кроме того, это сервисы в виде встроенных HTTP API, библиотеки SignalR или сервисов GRPC.

Все эти технологии базируются и/или взаимодействуют с чистым ASP.NET Core, который представлен прежде всего различными встроенными компонентами *middleware* - компонентами, которые применяются для обработки запроса. Кроме того, технологии высшего уровня также взаимодействуют с различными расширениями, которые не являются непосредственной частью ASP.NET Core, как расширения для логгирования, конфигурации и т.д.

Это вкратце архитектура платформы, теперь рассмотрим модели разработки приложения ASP.NET Core:

- Прежде всего это базовый ASP.NET Core, который поддерживает все основные моменты, необходимые для работы современного веб-приложения: маршрутизация, конфигурация, логгирования, возможность работы с различными системами баз данных и т.д.. В ASP.NET Core 6 в фреймворк был добавлен так называемый *Minimal API* - минимизированная упрощенная модель, который еще упростила процесс разработки и написания кода приложения. Все остальные модели разработки работают поверх базового функционала ASP.NET Core
- ASP.NET Core MVC представляет в общем виде построения приложения вокруг трех основных компонентов - *Model* (модели), *View* (представления) и *Controller* (контроллеры), где модели отвечают за работу с данными, контроллеры представляют логику обработки запросов, а представления определяют визуальную составляющую.



- Razor Pages представляет модель, при котором за обработку запроса отвечают специальные сущности - страницы Razor Pages. Каждую отдельную такую сущность можно ассоциировать с отдельной веб-страницей.
- ASP.NET Core Web API представляет реализацию паттерна REST, при котором для каждого типа http-запроса (GET, POST, PUT, DELETE) предназначен отдельный ресурс. Подобные ресурсы определяются в виде методов контроллера Web API. Данная модель особенно подходит для одностраничных приложений, но не только.
- Blazor представляет фреймворк, который позволяет создавать интерактивные приложения как на стороне сервера, так и на стороне клиента и позволяет задействовать на уровне браузера низкоуровневый код WebAssembly.

2.3. Особенности платформы

- ASP.NET Core работает поверх платформы .NET и, таким образом, позволяет задействовать весь ее функционал.
- В качестве языков разработки применяются языки программирования, поддерживаемые платформой .NET. Официально встроенная поддержка для проектов ASP.NET Core есть у языков C# и F#
- ASP.NET Core представляет кросс-платформенный фреймворк, приложения на котором могут быть развернуты на всех основных популярных операционных системах: Windows, Mac OS, Linux. И таким образом, с помощью ASP.NET Core мы можем как создавать кросс-платформенные приложения на Windows, на Linux и Mac OS, так и запускать на этих ОС.
- Благодаря модульности фреймворка все необходимые компоненты веб-приложения могут загружаться как отдельные модули через пакетный менеджер Nuget.

- Поддержка работы с большинством распространенных систем баз данных: MS SQL Server, MySQL, Postgres, MongoDB
- ASP.NET Core характеризуется расширяемостью. Фреймворк построен из набора относительно независимых компонентов. И мы можем либо использовать встроенную реализацию этих компонентов, либо расширить их с помощью механизма наследования, либо вовсе создать и применять свои компоненты со своим функционалом.
- Богатый инструментарий для разработки приложений. В качестве инструментария разработки мы можем использовать такую среду разработки с богатым функционалом, как **Visual Studio** от компании Microsoft.

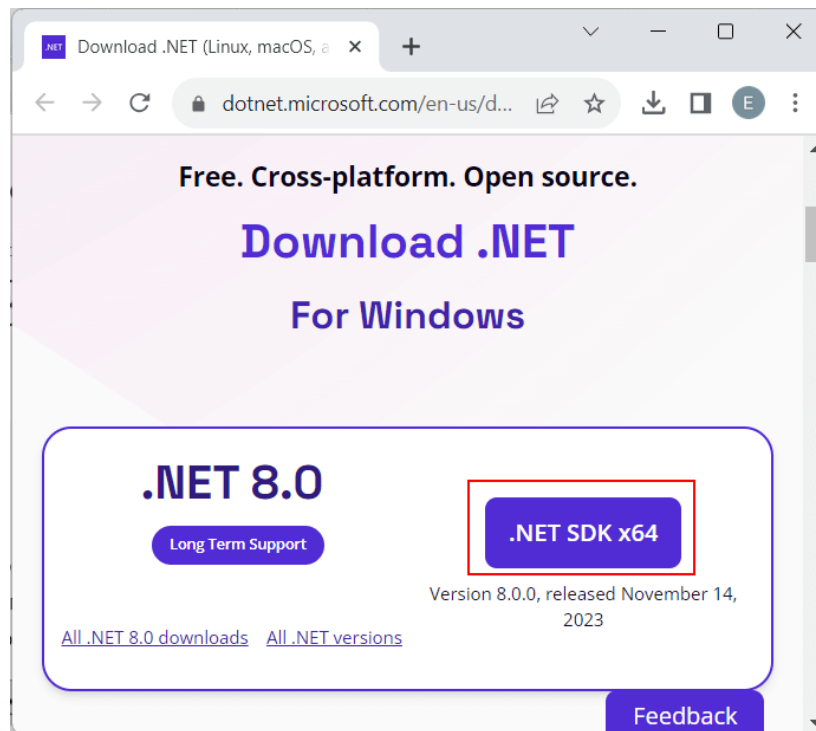
Также можно использовать для разработки среду **Rider** от компании JetBrains. Кроме того, имеющаяся оснастка .NET CLI позволяет создавать и запускать проекты ASP.NET в консоли. И таким образом, для написания кода можно использовать обычный текстовый редактор, например, **Visual Studio Code**.

3. Методика и порядок выполнения работы

3.1. Учебная задача

Создадим первую программу на ASP.NET Core. Что нам для этого потребуется? Прежде всего необходим текстовый редактор для написания кода программы. В данном случае я буду использовать в качестве текстового редактора Visual Studio Code

Также для компиляции и запуска программы нам потребуется .NET SDK. Для его установки перейдем на официальный сайт по ссылке [.NET SDK](#)



Откроем терминал/командную строку и перейдем к созданной папке проекта с помощью команды **cd**

```
cd C:\dotnet\aspnet\helloapp
```

В данном случае мы для создания и запуска проекта мы будем использовать встроенную инфраструктуру .NET CLI, которая устанавливается вместе с .NET SDK.

Для создания проекта в .NET CLI применяется команда **dotnet new**, после которой указывается тип проекта. Для ASP.NET Core есть ряд встроенных типов проектов. В данном случае мы будем использовать самый простейший тип - **web**. Поэтому введем в терминале команду

```
dotnet new web
```

```

Командная строка
Microsoft Windows [Version 10.0.22631.2715]
(с) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\eugen>cd C:\dotnet\aspnet\helloapp

C:\dotnet\aspnet\helloapp>dotnet new web
Шаблон "Пустой шаблон ASP.NET Core" успешно создан.

Идет обработка действий после создания ...
Восстановление C:\dotnet\aspnet\helloapp\helloapp.csproj:
  Определение проектов для восстановления ...
  Восстановлен C:\dotnet\aspnet\helloapp\helloapp.csproj (за 3,57 sec).
Восстановление выполнено.

C:\dotnet\aspnet\helloapp>

```

Структура проекта ASP.NET Core

Рассмотрим базовую структуру простейшего стандартного 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
- **helloapp.csproj**: стандартный файл проекта C#, который соответствует назанию проекта (по умолчанию названию каталога) и описывает все его настройки.
- **Program.cs**: главный файл приложения, с которого и начинается его выполнение. Код этого файла настраивает и запускает веб-приложение

Например, посмотрим на содержимое файла **helloapp.csproj**

```

<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

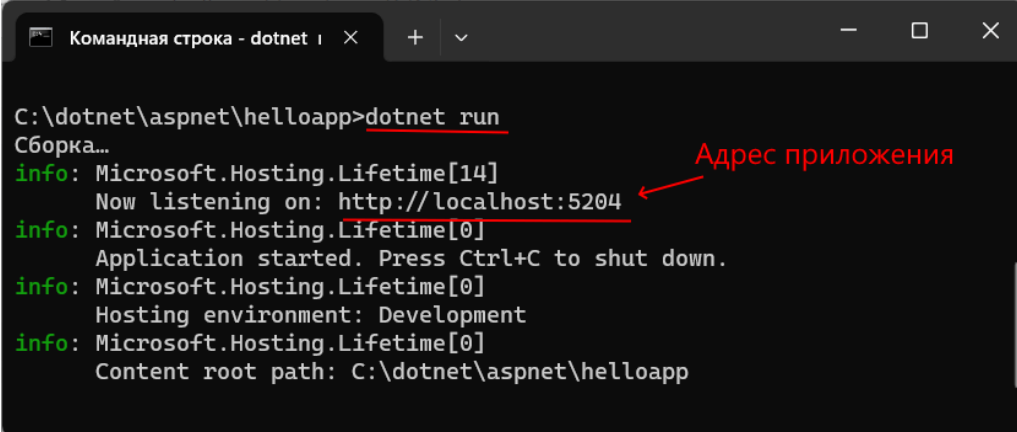
```

</Project>

Ключевой компонент здесь - атрибут `Sdk="Microsoft.NET.Sdk.Web"`, который собственно и определяет, что приложение будет использовать SDK "Microsoft.NET.Sdk.Web", который предназначен именно для веб-проектов.

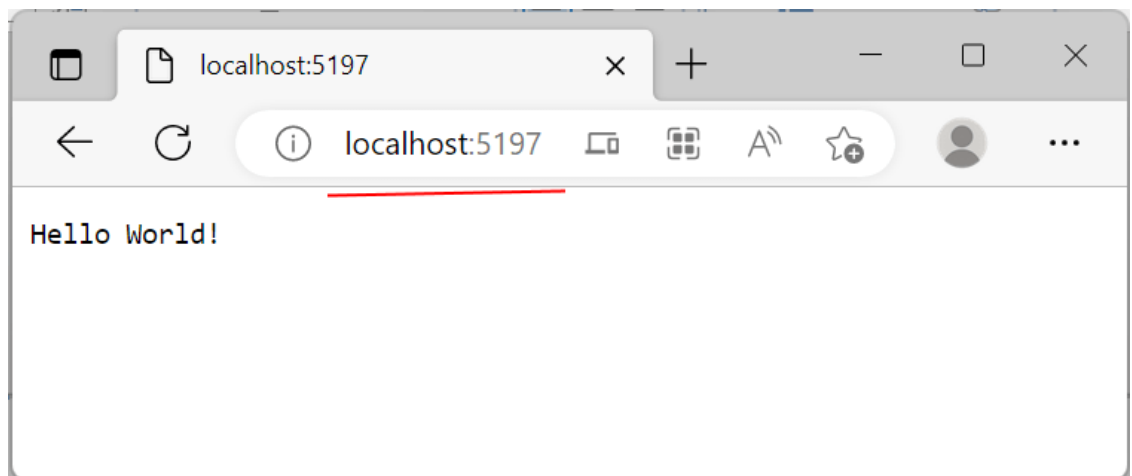
Запуск проекта

Проект по умолчанию не представляет какой-то грандиозной функциональности, тем не менее этот проект мы уже можем запустить. Итак, запустим проект. Для этого выполним команду



```
Командная строка - dotnet | x + v
C:\dotnet\aspnet\helloapp>dotnet run
Сборка...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5204
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\dotnet\aspnet\helloapp
```

При запуске в консоли мы можем увидеть адрес, по которому мы можем обращаться к приложению. В моем случае это адрес "http://localhost:5204". И я могу обратиться по этому адресу к приложению в браузере и увидеть в нем строку "Hello World!" - результат работы кода по умолчанию из файла **Program.cs**:



Запуск приложения и файл Program.cs

Рассмотрим код файла **Program.cs**, который создает подобное приложение:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

Это так называемое **Minimal API** – упрощенная минимизированная модель для запуска веб-приложения в ASP.NET. Приложение в ASP.NET Core представляет объект **Microsoft.AspNetCore.Builder.WebApplication**. Этот объект настраивает всю конфигурацию приложения, его маршруты, используемые зависимости и т.д.. Исходный код класса на github: <https://github.com/dotnet/aspnetcore/blob/main/src/DefaultBuilder/src/WebApplication.cs>

Для создания объекта **WebApplication** необходим специальный класс-строитель – **WebApplicationBuilder**. И в файле Program.cs вначале создается данный объект с помощью статического метода **WebApplication.CreateBuilder**:

```
var builder = WebApplication.CreateBuilder(args);
```

В качестве параметра в метод передаются аргументы, которые передаются приложению при запуске.

Получив объект **WebApplicationBuilder**, у него вызывается метод **Build()**, который собственно и создает объект **WebApplication**:

```
var app = builder.Build();
```

С помощью объекта **WebApplication** можно настроить всю инфраструктуру приложения - его конфигурацию, маршруты и так далее. В файле Program.cs по умолчанию для приложения определяется один маршрут:

```
app.MapGet("/", () => "Hello World!");
```

Метод **MapGet()** в качестве первого параметра принимает путь, по которому можно обратиться к приложению. В данном случае это путь "/", то есть по сути корень веб-приложения - имя домена и порта, после которых может идти слеш, например, <https://localhost:7256/>

В качестве второго параметра в метод **MapGet()** передаются обработчик запроса по этому маршруту в виде функции. Здесь это лямбда-выражение, которое возвращает строку "Hello World!". Именно поэтому при обращении к приложению мы увидим данную строку в браузере.

И в конце необходимо запустить приложение. Для этого у класса **WebApplication** вызывается метод **Run()**:

```
app.Run();
```

В итоге запустится приложение в виде консоли, и мы сможем обращаться к приложению из различных браузеров.

3.2. Индивидуальное задание

1. Проанализируйте технологии, рассмотренные в теоретической части лабораторной работы.

2. Создайте приложение в соответствии с учебной задачей.

3. Выберите тематику разрабатываемого приложения. Для этого можно использовать Приложение 1 или согласовать выбор предметной области с ведущим преподавателем.

4. В соответствии с выбранной тематикой разработайте каркас приложения по аналогии с алгоритмом, описанном в учебной задаче.

4. Контрольные вопросы

1. Опишите структуру и назначение ASP.NET Core.

2. Поясните назначение и особенности технологий ASP.NET Core, ASP.NET MVC, Razor Pages, ASP.NET Web API, Blazor.

3. Перечислите основные особенности платформы ASP.NET Core.

ЛАБОРАТОРНАЯ РАБОТА 2. ПРИЛОЖЕНИЕ ASP.NET CORE

1. Цель и задачи

Цель лабораторной работы: изучение архитектуры приложения ASP.NET Core.

Задачи лабораторной работы:

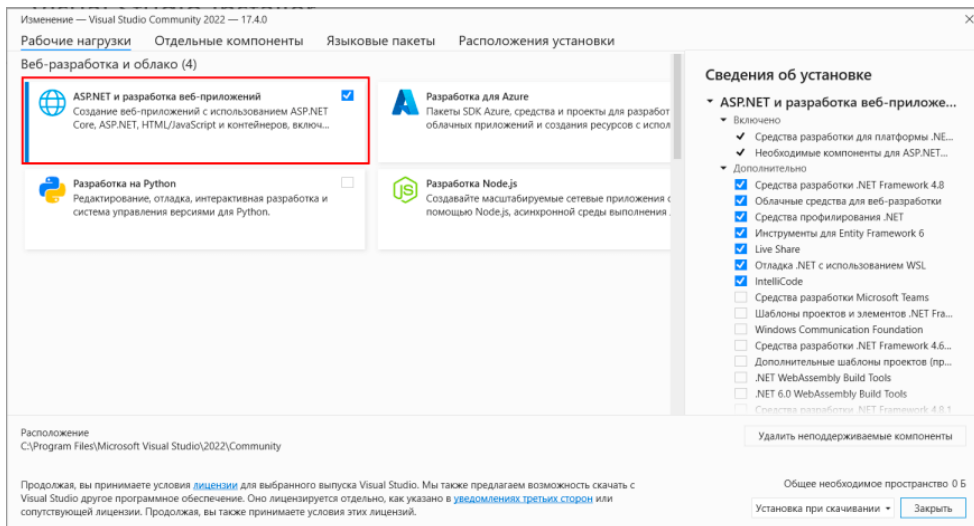
1. Изучить основы построения приложений с использованием ASP.NET Core
2. Научиться применять Visual Studio для выполнения основных этапов разработки приложений ASP.NET Core
3. Исследовать механизм обработки веб-запросов.

2. Теоретическое обоснование

2.1. Приложение Visual Studio

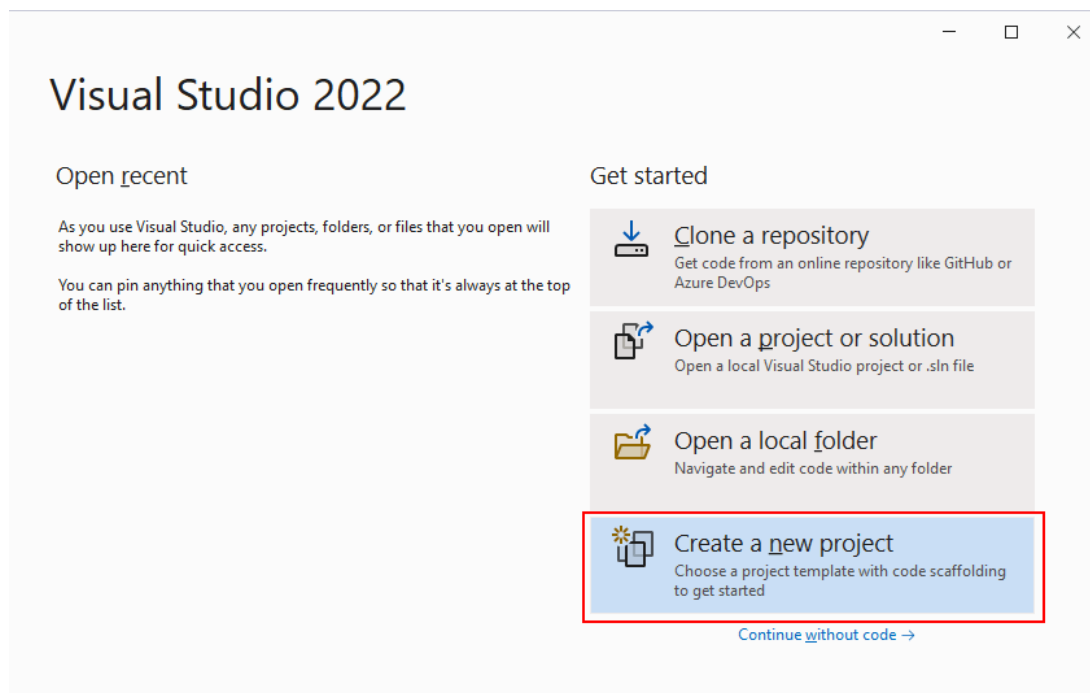
В прошлой работе было рассмотрено создание первого проекта ASP.NET Core с помощью .NET CLI с компиляцией и запуском проекта в терминале. Это самый простой способ для создания приложений ASP.NET. Однако также мы можем использовать среду разработки Visual Studio, которая упрощает многие аспекты по работе с проектом ASP.NET. Рассмотрим, как создавать проект в Visual Studio.

Чтобы добавить в Visual Studio поддержку проектов для ASP.NET Core, в программе установки среди рабочих нагрузок можно выбрать только пункт **ASP.NET и разработка веб-приложений**.

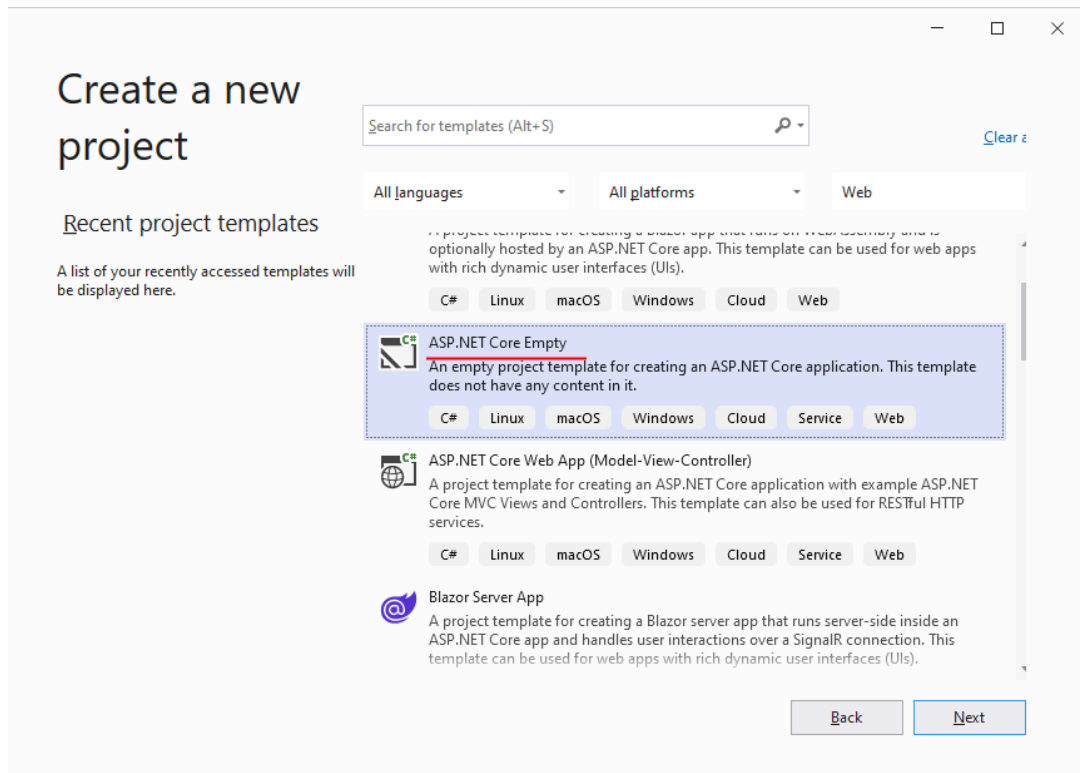


И при инсталляции Visual Studio на ваш компьютер будут установлены все необходимые инструменты для разработки программ, в том числе фреймворк .NET.

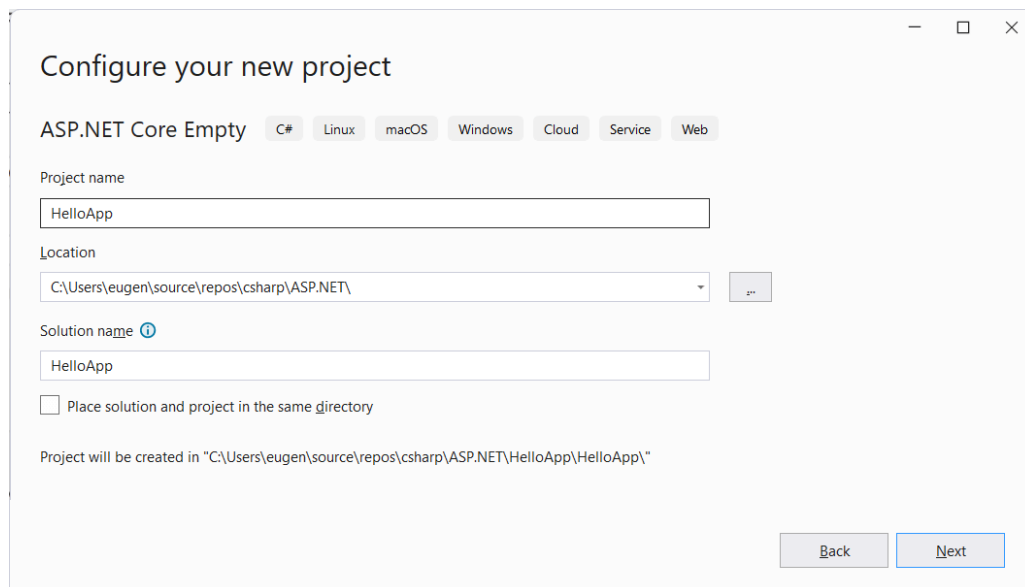
После завершения установки создадим первый проект на ASP.NET Core. Вначале откроем Visual Studio. На стартовом экране выберем **Create a new project** (Создать новый проект)



На следующем окне в качестве типа проекта выберем **ASP.NET Core Empty**



Далее на следующем этапе нам будет предложено указать имя проекта и каталог, где будет располагаться проект.



В поле **Project Name** дадим проекту какое-либо название. В моем случае это **HelloApp**.

На следующем окне Visual Studio предложит нам выбрать версию .NET, которая будет использоваться для проекта. Выберем здесь последнюю доступную версию.

Additional information

ASP.NET Core Empty C# Linux macOS Windows Cloud Service Web

Framework ⓘ

.NET 8.0 (Long Term Support)

Configure for HTTPS ⓘ

Enable Docker ⓘ

Docker OS ⓘ

Linux

Do not use top-level statements ⓘ

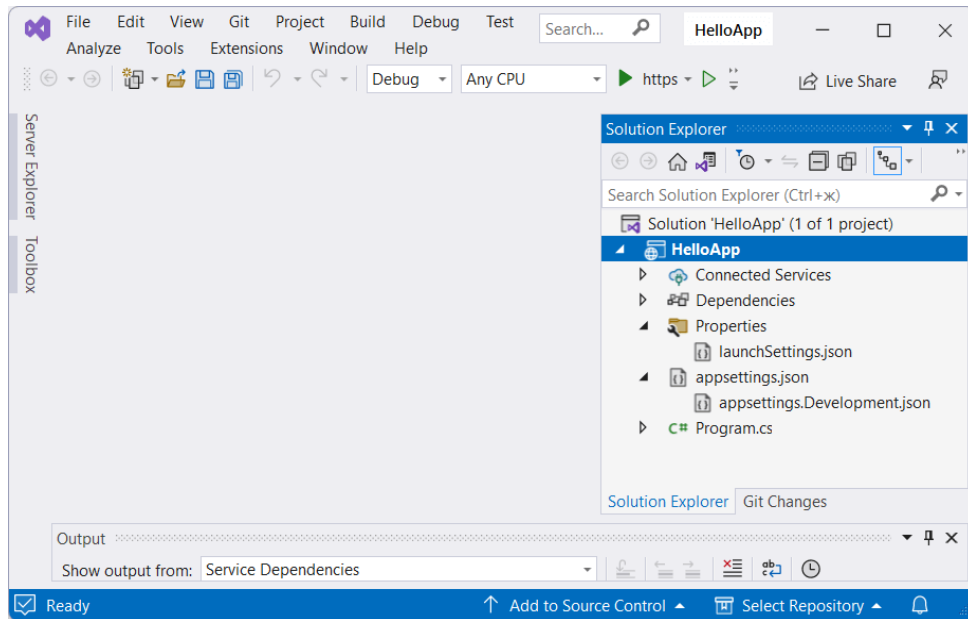
Back Create

Кроме того, здесь с помощью флажка **Configure for HTTPS** можно установить использование протокола https. По умолчанию этот флажок отмечен, это значит, что проект по умолчанию будет использовать протокол https.

Другой флажок – **Enable Docker** позволяет задействовать Docker. Если этот флажок установлен, то в поле ниже можно будет выбрать ОС, которая будет использоваться под Docker. Но в данном случае оставим этот флажок неотмеченным.

Третий флажок – **Do not use level statements** позволяет, как и в консольных проектах, отключить поддержку выражений верхнего уровня. Этот флажок также оставим неотмеченным.

Оставим все остальные настройки по умолчанию и нажмем на кнопку Create (Создать) для создания проекта. После этого Visual Studio создаст и откроет нам проект:



Структура проекта ASP.NET Core

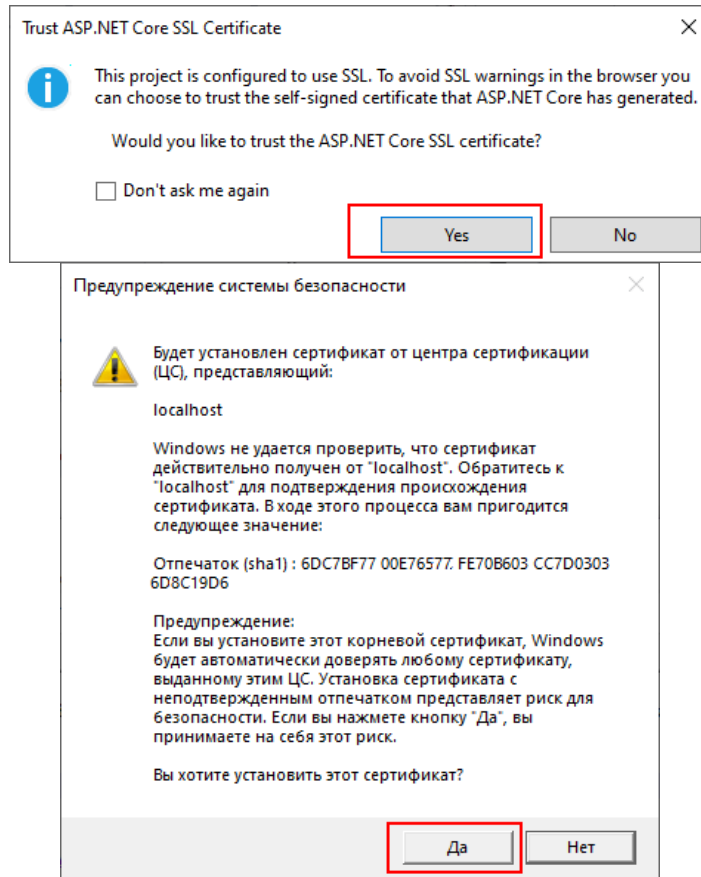
Рассмотрим базовую структуру стандартного проекта ASP.NET Core в Visual Studio. Проект **ASP.NET Core Empty** содержит очень простую структуру - необходимый минимум для запуска приложения:

- **Connected Services**: подключенные сервисы из Azure
- **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 проекта 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

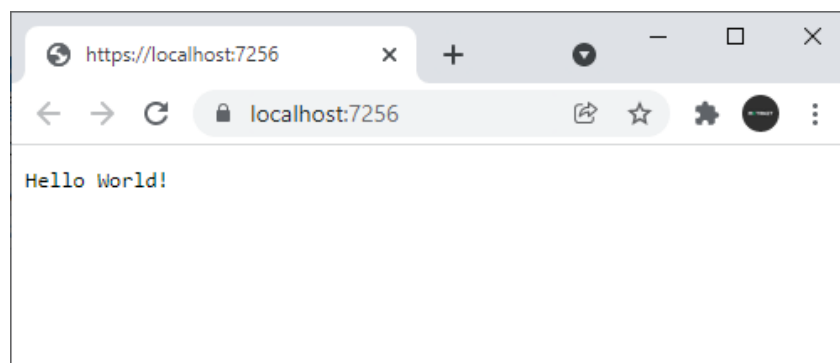
Запуск проекта

Проект по умолчанию не представляет какой-то грандиозной функциональности, тем не менее этот проект мы уже можем запустить. Итак,

запустим проект. При запуске нам может отобразиться окно, где надо подтвердить доверие для сертификата SSL, а также его установку



И после подтверждения и установки сертификата отобразится консоль, где выводится некоторая базовая информация о приложении. И, кроме того, будет запущен браузер, где мы сможем лицезреть строку "Hello World!" - результат работы кода по умолчанию из файла Program.cs:

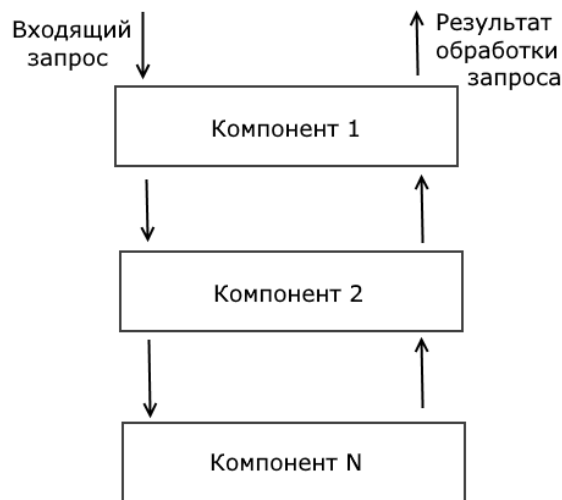


2.2. Конвейер обработки запроса и middleware

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** (в русском языке до сих пор нет адекватного термина для подобным компонента, поэтому далее они именуются преимущественно как "компоненты middleware" или просто 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** (terminal middleware). 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.

Каждый компонент middleware может быть определен как метод (встроенный inline компонент), либо может быть вынесен в отдельный класс.

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.

Рассмотрим, какую информацию мы можем получить из `HttpContext`. Для этого пройдемся по его свойствам:

- **Connection**: представляет информацию о подключении, которое установлено для данного запроса
- **Features**: получает коллекцию HTTP-функциональностей, которые доступны для этого запроса
- **Items**: получает или устанавливает коллекцию пар ключ-значение для хранения некоторых данных для текущего запроса
- **Request**: возвращает объект **HttpRequest**, который хранит информацию о текущем запросе
- **RequestAborted**: уведомляет приложение, когда подключение прерывается, и соответственно обработка запроса должна быть отменена
- **RequestServices**: получает или устанавливает объект **IServiceProvider**, который предоставляет доступ к контейнеру сервисов запроса
- **Response**: возвращает объект **HttpResponse**, который позволяет управлять ответом клиенту
- **Session**: хранит данные сессии для текущего запроса
- **TraceIdentifier**: представляет уникальный идентификатор запроса для логов трассировки
- **User**: представляет пользователя, ассоциированного с этим запросом
- **WebSockets**: возвращает объект для управления подключениями `WebSocket` для данного запроса

Используя эти свойства мы можем в компоненте `middleware` получить если не все, то большую часть необходимых данных о запросе и отправить обратно клиенту некоторый ответ.

Встроенные компоненты middleware

Стоит отметить, что ASP.NET Core уже по умолчанию предоставляет ряд встроенных компонентов middleware для часто встречающихся задач:

- Authentication: предоставляет поддержку аутентификации
- Authorization: предоставляет поддержку авторизации
- Cookie Policy: отслеживает согласие пользователя на хранение связанной с ним информации в куках
- CORS: обеспечивает поддержку кроссдоменных запросов
- DeveloperExceptionPage: генерирует веб-страницу с информацией об ошибке при работе в режиме разработки
- Diagnostics: набор middleware, который предоставляет страницы статусных кодов, функционал обработки исключений, страницу исключений разработчика
- Forwarded Headers: перенаправляет заголовки запроса
- Health Check: проверяет работоспособность приложения asp.net core
- Header Propagation: обеспечивает передачу заголовков из HTTP-запроса
- HTTP Logging: логирует информацию о входящих запросах и генерируемых ответах
- HTTP Method Override: позволяет входящему POST-запросу переопределить метод
- HTTPS Redirection: перенаправляет все запросы HTTP на HTTPS
- HTTP Strict Transport Security (HSTS): для улучшения безопасности приложения добавляет специальный заголовок ответа
- MVC: обеспечивает функционал фреймворка MVC
- OWIN: обеспечивает взаимодействие с приложениями, серверами и компонентами, построенными на основе спецификации OWIN
- Request Localization: обеспечивает поддержку локализации
- Response Caching: позволяет кэшировать результаты запросов
- Response Compression: обеспечивает сжатие ответа клиенту
- URL Rewrite: предоставляет функциональность URL Rewriting

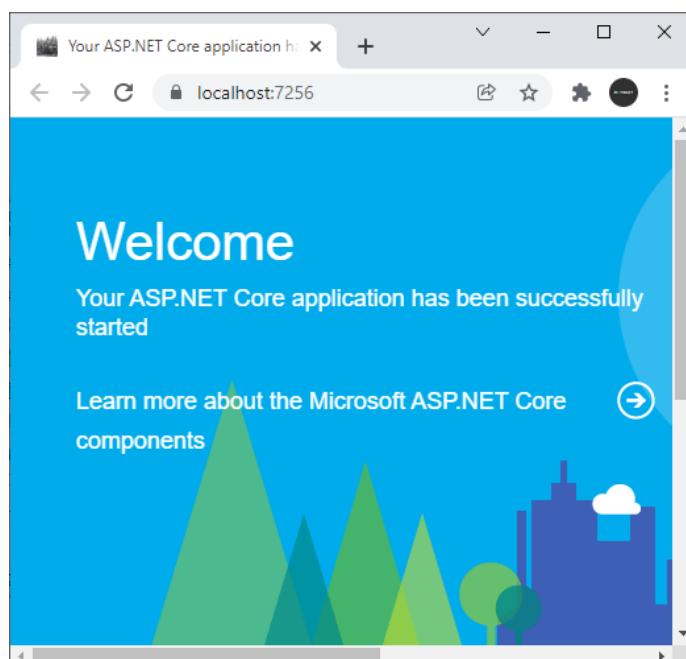
- Endpoint Routing: предоставляет механизм маршрутизации
- Session: предоставляет поддержку сессий
- SPA: обрабатывает все запросы, возвращая страницу по умолчанию для SPA-приложения (одностраничного приложения)
- Static Files: предоставляет поддержку обработки статических файлов
- WebSockets: добавляет поддержку протокола WebSockets
- W3CLogging: генерирует логи доступа в соответствии с форматом W3C Extended Log File Format

Для встраивания этих компонентов в конвейер обработки запроса для интерфейса `IApplicationBuilder` определены методы расширения типа `UseXXX`.

Например, фреймворк ASP.NET Core по умолчанию предоставляет такой middleware как **WelcomePageMiddleware**, который отправляет клиенту некоторую стандартную веб-страницу. Для подключения этого компонента в конвейер запроса применяется метод расширения **UseWelcomePage()**:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.UseWelcomePage(); // подключение WelcomePageMiddleware
app.Run();
```

И при выполнении этого приложения браузер представит нашему взору следующую красочную страницу:



3. Методика и порядок выполнения работы

3.1. Учебная задача

Создание и запуск приложения. `WebApplication` и `WebApplicationBuilder`.

В центре приложения ASP.NET находится класс **WebApplication**. Например, если мы возьмем проект ASP.NET по типу **ASP.NET Core Empty**, то в файле **Program.cs** мы встретим следующий код:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

Переменная `app` в данном коде как раз представляет объект **WebApplication**. Однако для создания этого объекта необходим другой объект - **WebApplicationBuilder**, который в данном коде представлен переменной `builder`.

Класс `WebApplicationBuilder`

Создание приложения по умолчанию фактически начинается с класса **WebApplicationBuilder**. Исходный код доступен по адресу [WebApplicationBuilder.cs](#)

Для его создания объекта этого класса вызывается статический метод **WebApplication.CreateBuilder()**:

```
WebApplicationBuilder builder
= WebApplication.CreateBuilder();
```

Для инициализации объекта `WebApplicationBuilder` в этот метод могут передаваться аргументы командной строки, указанные при запуске приложения (доступны через неявно определенный параметр `args`):

```
WebApplicationBuilder builder
= WebApplication.CreateBuilder(args);
```

Либо можно передавать объект **WebApplicationOption**:

```
WebApplicationOptions options = new() { Args = args };
WebApplicationBuilder builder
= WebApplication.CreateBuilder(options);
```

Кроме создания объекта `WebApplication` класс **WebApplicationBuilder** выполняет еще ряд задач, среди которых можно выделить следующие:

- Установка конфигурации приложения
- Добавление сервисов
- Настройка логгирования в приложении
- Установка окружения приложения
- Конфигурация объектов `IHostBuilder` и `IWebHostBuilder`, которые применяются для создания хоста приложения

Для реализации этих задач в классе `WebApplicationBuilder` определены следующие свойства:

- **Configuration**: представляет объект `ConfigurationManager`, который применяется для добавления конфигурации к приложению.
- **Environment**: предоставляет информацию об окружении, в котором запущено приложение.
- **Host**: объект `IHostBuilder`, который применяется для настройки хоста.
- **Logging**: позволяет определить настройки логгирования в приложении.
- **Services**: представляет коллекцию сервисов и позволяет добавлять сервисы в приложение.
- **WebHost**: объект `IWebHostBuilder`, который позволяет настроить отдельные настройки сервера.

Класс `WebApplication`

Метод `build()` класса `WebApplicationBuilder` создает объект

WebApplication:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();
WebApplication app = builder.Build();
```

Класс `WebApplication` применяется для управления обработкой запроса, установки маршрутов, получения сервисов и т.д. Исходный код класса можно найти на Github по адресу [WebApplication.cs](https://github.com/aspnet/WebApplication.cs).

Класс `WebApplication` применяет три интерфейса:

- **IHost**: применяется для запуска и остановки хоста, который прослушивает входящие запросы
- **IApplicationBuilder**: применяется для установки компонентов, которые участвуют в обработке запроса

- **EndpointRouteBuilder**: применяется для установки маршрутов, которые сопоставляются с запросами

Для получения доступа к функциональности приложения можно использовать свойства класса `WebApplication`:

- **Configuration**: представляет конфигурацию приложения в виде объекта **IConfiguration**
- **Environment**: представляет окружение приложения в виде **IWebHostEnvironment**
- **Lifetime**: позволяет получать уведомления о событиях жизненного цикла приложения
- **Logger**: представляет логгер приложения по умолчанию
- **Services**: представляет сервисы приложения
- **Urls**: представляет набор адресов, которые использует сервер

Для управления хостом класс `WebApplication` определяет следующие методы:

- **Run()**: запускает приложение
- **RunAsync()**: асинхронно запускает приложение
- **Start()**: запускает приложение
- **StartAsync()**: запускает приложение
- **StopAsync()**: останавливает приложение

Таким образом, после вызова метод `Run/Start/RunAsync/StartAsync` приложение будет запущено, и мы сможем к нему обращаться:

```
WebApplicationBuilder builder =
    WebApplication.CreateBuilder();
WebApplication app = builder.Build();
app.Run();
```

При необходимости с помощью метода **StopAsync()** можно программным способом завершить выполнение приложения:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();
WebApplication app = builder.Build();
app.MapGet("/", () => "Hello World!");
await app.StartAsync();
await Task.Delay(10000);
await app.StopAsync();
```

Метод Run и определение терминального middleware

Самый простой способ добавления middleware в конвейер обработки запроса в ASP.NET Core представляет метод **Run()**, который определен как метод расширения для интерфейса `IApplicationBuilder` (соответственно его поддерживает и класс `WebApplication`):

```
IApplicationBuilder.Run(RequestDelegate handler)
```

Метод `Run` добавляет терминальный компонент - такой компонент, который завершает обработку запроса. Поэтому соответственно 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`. Этот делегат имеет следующее определение:

```
public delegate Task RequestDelegate(HttpContext context);
```

Он принимает в качестве параметра контекст запроса `HttpContext` и возвращает объект `Task`.

Используем этот метод для определения простейшего компонента:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(async (context)
=> await context.Response.WriteAsync("Hello Nikolaev E.I."));
app.Run();
```

Здесь для делегата `RequestDelegate` передается лямбда-выражение, параметр которого – `HttpContext` можно использовать для отправки ответа. В частности, метод `context.Response.WriteAsync()` позволяет отправить клиенту некоторый ответ - в данном случае отправляется простая строка.

Здесь следует сделать пару замечаний. Прежде всего, не стоит путать метод **Run()**, который определен в классе `WebApplication` и который запускает приложение, и метод расширения **Run()**, который встраивает компонент middleware. Это два разных метода, которые выполняют разные задачи. И, как видно из кода выше, вызываются оба этих метода.

Второй момент - метод **Run()**, который запускает приложение, вызывается **после** добавления компонента `middleware`. И мы НЕ можем написать так:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(); // приложение запущено
// в этой строке уже нет смысла
app.Run(async (context) => await
context.Response.WriteAsync("Hello METANIT.COM"));
```

При необходимости естественно мы можем вынести код `middleware` в отдельный метод:

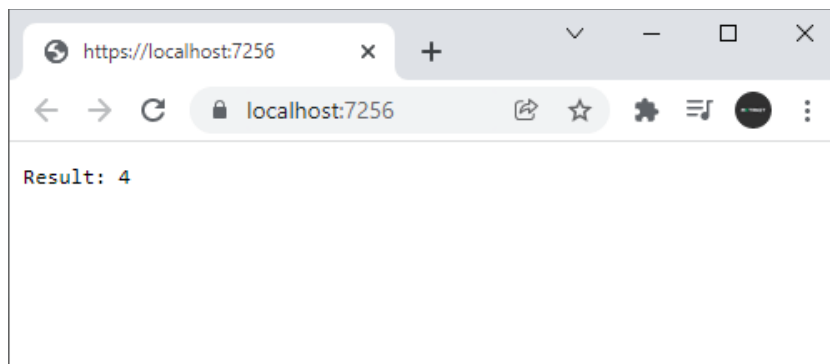
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(HandleRequest);
app.Run();
async Task HandleRequest(HttpContext context)
{
    await context.Response.WriteAsync("Hello METANIT.COM 2");
}
```

Жизненный цикл `middleware`

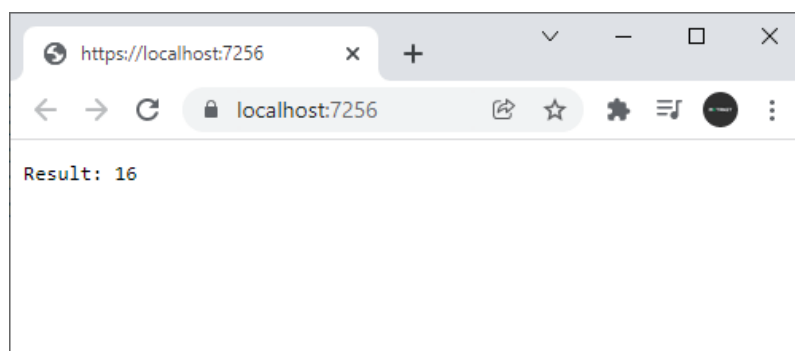
Компоненты `middleware` создаются один раз и существуют в течение всего жизненного цикла приложения. То есть для последующей обработки запросов используются одни и те же компоненты. Например, определим в файле **Program.cs** следующий код:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
int x = 2;
app.Run(async (context) =>
{
    x = x * 2; // 2 * 2 = 4
    await context.Response.WriteAsync($"Result: {x}");
});
app.Run();
```

При запуске приложения мы естественно ожидаем, что браузер выведет число 4 в качестве результата:



Однако при последующих запросах мы увидим, что результат переменной `x` не равен 4.



Также стоит отметить, что браузер Google Chrome может посылать два запроса – один собственно к приложению, а другой - к файлу иконки `favicon.ico`, поэтому в Google Chrome результат может отличаться не 2 раза, а гораздо больше.

3.2. Индивидуальное задание

1. Проанализируйте технологии, рассмотренные в теоретической части лабораторной работы.
2. Создайте приложение в соответствии с учебной задачей.
3. В соответствии с выбранной предметной областью, реализуйте приложение ASP.NET Core и продемонстрируйте применение следующих элементов, классов: `WebApplication`, `WebApplicationBuilder`.

4. Контрольные вопросы

1. Что такое контроллер в веб-приложении?
2. Опишите назначение и методы класса `WebApplication`.

3. Поясните назначение middleware.
4. Опишите назначение и методы класса `WebApplicationBuilder`.

ЛАБОРАТОРНАЯ РАБОТА 3. ФОРМИРОВАНИЕ ВЕБ-ОТВЕТОВ

1. Цель и задачи

Цель лабораторной работы: изучить механизм формирования ответов в веб-приложениях.

Задачи лабораторной работы:

1. Изучить принципы работы с классами `HttpContext`, `HttpResponse`.
2. Научиться использовать свойства `HttpContext` при программировании веб-приложений.
3. Научиться использовать свойства `HttpResponse` при программировании веб-приложений.
4. Научиться использовать статусы ответов.

2. Теоретическое обоснование

2.1. Основы отправки ответов

Все данные запроса передаются в `middleware` через объект **`Microsoft.AspNetCore.Http.HttpContext`**. Этот объект инкапсулирует информацию о запросе, позволяет управлять ответом и, кроме того, имеет еще много другой функциональности. Например, возьмем простейшее приложение:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Run(async (context) =>
    await context.Response.WriteAsync("Hello Nikolaev E."));
app.Run();
```

Здесь параметр `context`, который передается в `middleware` в методе `app.Run()` как раз представляет объект **`HttpContext`**. И через этот объект, точнее через его свойство **`Response`** мы можем отправить клиенту некоторый ответ: `context.Response.WriteAsync($"Hello Nikolaev E.")`.

Свойство **`Response`** объекта `HttpContext` представляет объект **`HttpResponse`** и устанавливает, что будет отправляться в виде ответа. Для

установки различных аспектов ответа класс `HttpResponse` определяет следующие свойства:

- **Body**: получает или устанавливает тело ответа в виде объекта **Stream**
- **BodyWriter**: возвращает объект типа **PipeWriter** для записи ответа
- **ContentLength**: получает или устанавливает заголовок Content-Length
- **ContentType**: получает или устанавливает заголовок Content-Type
- **Cookies**: возвращает куки, отправляемые в ответе
- **HasStarted**: возвращает true, если отправка ответа уже началась
- **Headers**: возвращает заголовки ответа
- **Host**: получает или устанавливает заголовок Host
- **HttpContext**: возвращает объект **HttpContext**, связанный с данным объектом `Response`
- **StatusCode**: возвращает или устанавливает статусный код ответа

Чтобы отправить ответ, мы можем использовать ряд методов класса `HttpResponse`:

- **Redirect()**: выполняет переадресацию (временную или постоянную) на другой ресурс
- **WriteAsJson()/WriteAsJsonAsync()**: отправляет ответ в виде объектов в формате JSON
- **WriteAsync()**: отправляет некоторое содержимое. Одна из версий метода позволяет указать кодировку. Если кодировка не указана, то по умолчанию применяется кодировка UTF-8
- **SendFileAsync()**: отправляет файл

Самый простой способ отправки ответа представляет метод **WriteAsync()**, в который передается отправляемые данные. В качестве дополнительного параметра мы можем указать кодировку:

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello Nikolaev E.",
System.Text.Encoding.Default);
});
```

2.2. Установка заголовков

Для установки заголовков применяется свойство **Headers**, которое представляет тип **IHeaderDictionary**. Для большинства стандартных заголовков HTTP в этом интерфейсе определены одноименные свойства, например, для заголовка "content-type" определено свойство `ContentType`. Другие, в том числе свои кастомные заголовки можно добавить через метод `Append()`. Например:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(async (context) =>
{
    var response = context.Response;
    response.Headers.ContentLanguage = "ru-RU";
    response.Headers.ContentType = "text/plain; charset=utf-8";
    response.Headers.Append("secret-id", "256");
    await response.WriteAsync("Привет Nikolaev!");
});
app.Run();
```

Стоит отметить, что для вывода кириллицы желательно устанавливать заголовок `ContentType`, в том числе кодировку, которая применяется в отправляемом содержимом (в примере выше это "text/plain; charset=utf-8").

Также стоит отметить, что вместо

```
response.Headers.ContentType = "text/plain; charset=utf-8";
    можно было бы написать
response.ContentType = "text/plain; charset=utf-8";
```

2.3. Установка кодов статуса

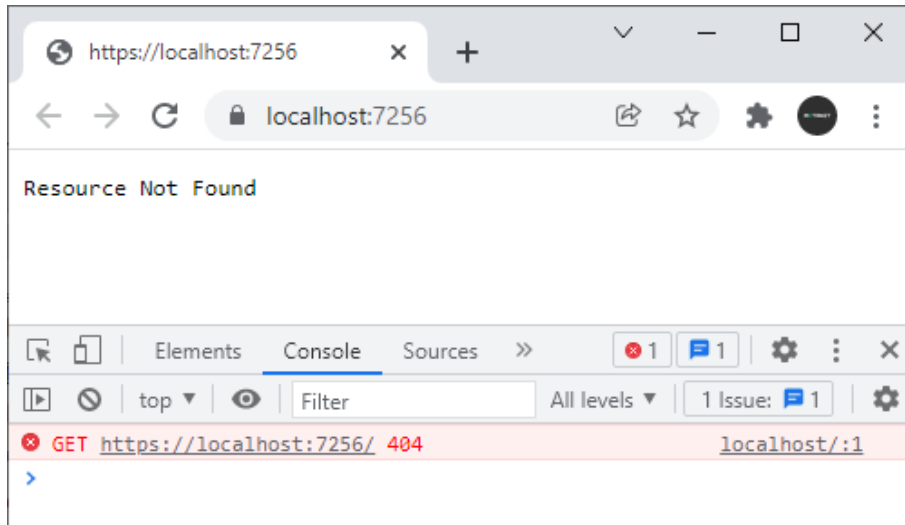
Для установки статусных кодов применяется свойство **StatusCode**, которому передается числовой код статуса:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Run(async (context) =>
{
    context.Response.StatusCode = 404;
    await context.Response.WriteAsync("Resource Not Found");
});

app.Run();
```

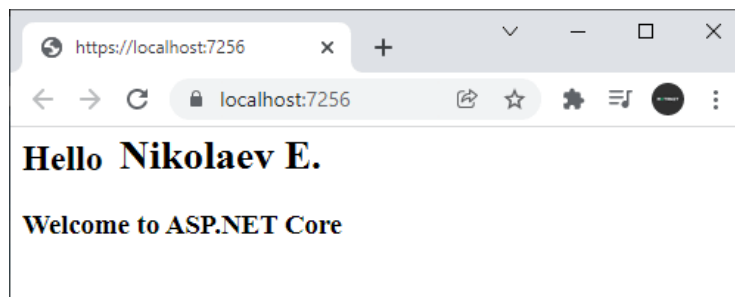
В данном случае устанавливается код 404, который указывает, что ресурс не найден.



2.3. Отправка html-кода

Если необходимо отправить html-код, то для этого необходимо установить для заголовка Content-Type значение text/html:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(async (context) =>
{
    var response = context.Response;
    response.ContentType = "text/html; charset=utf-8";
    await response.WriteAsync(
"<h2>Nikolaev E.</h2><h3>Welcome to ASP.NET Core</h3>"
);
});
app.Run();
```



3. Методика и порядок выполнения работы

3.1. Учебная задача

Изучите код, представленный в теоретической части лабораторной работы.

3.2. Индивидуальное задание

1. Проанализируйте технологии, рассмотренные в теоретической части лабораторной работы.

2. Создайте приложение в соответствии с учебной задачей.

3. В соответствии с выбранной предметной областью, реализуйте приложение ASP.NET Core и продемонстрируйте применение следующих элементов, классов: HttpContext, Response.

4. Выполните установку статуса в своем приложении, установку заголовка и отправку html-содержимого в качестве ответа.

4. Контрольные вопросы

1. Опишите назначение класса HttpContext, укажите назначение и тип его свойств.

2. Опишите назначение класса HttpResponse, укажите назначение и тип его свойств.

3. Поясните механизм установки заголовков веб-ответов.

4. Опишите механизм отправки html-страницы в качестве ответа.

5. Как получить доступ к объекту HttpResponse в контроллере?

6. Как добавить или изменить заголовок ответа?

7. Какие стандартные заголовки HTTP можно настроить через HttpResponse?

8. Как управлять кэшированием ответа через заголовки?

9. Как отправить простой текст в ответ?

10. Как работать с потоками данных в HttpResponse?

11. Какие форматы данных можно отправлять через `HttpResponse`?
12. Как настроить кодировку ответа?
13. Как управлять `Content-Type` в ответе?
14. Как сериализовать объекты в JSON через `HttpResponse`?
15. Как правильно использовать асинхронные методы при работе с `HttpResponse`?

ЛАБОРАТОРНАЯ РАБОТА 4. ПОЛУЧЕНИЕ ДАННЫХ ЗАПРОСА

1. Цель и задачи

Цель лабораторной работы: изучение принципов обработки веб-запросов.

Задачи лабораторной работы:

1. Изучить возможности класса `HttpRequest`.
2. Научиться использовать веб-пути.
3. Научиться разбирать строки запросов.

2. Теоретическое обоснование

2.1. Класс `HttpRequest`

Свойство **Request** объекта `HttpContext` представляет объект `HttpRequest` и хранит информацию о запросе в виде следующих свойств:

- **Body**: предоставляет тело запроса в виде объекта **Stream**
- **BodyReader**: возвращает объект типа **PipeReader** для чтения тела запроса
- **ContentLength**: получает или устанавливает заголовок `Content-Length`
- **ContentType**: получает или устанавливает заголовок `Content-Type`
- **Cookies**: возвращает коллекцию куки (объект `Cookies`), ассоциированных с данным запросом
- **Form**: получает или устанавливает тело запроса в виде форм
- **HasFormContentType**: проверяет наличие заголовка `Content-Type`
- **Headers**: возвращает заголовки запроса
- **Host**: получает или устанавливает заголовок `Host`
- **HttpContext**: возвращает связанный с данным запросом объект `HttpContext`
- **IsHttps**: возвращает `true`, если применяется протокол `https`
- **Method**: получает или устанавливает метод HTTP

- **Path**: получает или устанавливает путь запроса в виде объекта `RequestPath`
- **PathBase**: получает или устанавливает базовый путь запроса. Такой путь не должен содержать завершающий слеш
- **Protocol**: получает или устанавливает протокол, например, HTTP
- **Query**: возвращает коллекцию параметров из строки запроса
- **QueryString**: получает или устанавливает строку запроса
- **RouteValues**: получает данные маршрута для текущего запроса
- **Scheme**: получает или устанавливает схему запроса HTTP

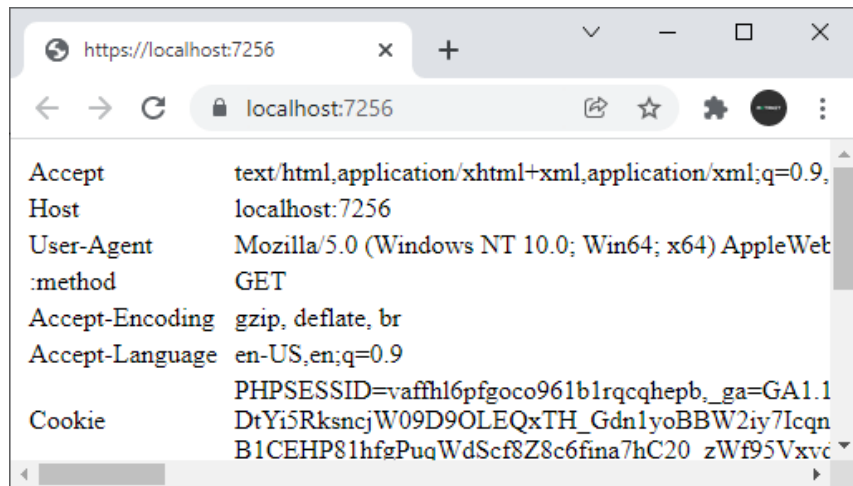
Рассмотрим применение некоторых из этих свойств.

Получение заголовков запроса

Для получения заголовков применяется свойство **Headers**, которое представляет тип **IHeaderDictionary**. Например, получим все заголовки запроса и выведем их на веб-страницу:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(async (context) =>
{
    context.Response.ContentType =
        "text/html; charset=utf-8";
    var stringBuilder
= new System.Text.StringBuilder("<table>");

    foreach(var header in context.Request.Headers)
    {
        stringBuilder.Append(
$"<tr><td>{header.Key}</td><td>{header.Value}</td></tr>");
    }
    stringBuilder.Append("</table>");
    await
context.Response.WriteAsync(stringBuilder.ToString());
});
app.Run();
```



Для большинства стандартных заголовков HTTP в этом интерфейсе определены одноименные свойства, например, для заголовка "content-type" определено свойство `ContentType`, а для заголовка "accept" - свойство `Accept`:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(async (context) =>
{
    var acceptHeaderValue = context.Request.Headers.Accept;
    await context.Response.WriteAsync($"Accept:
                                     {acceptHeaderValue}");
});

app.Run();
```

Также подобные заголовки, а также какие-то кастомные заголовки, для которых не определены подобные свойства, можно получить как и любой дугой элемент словаря:

```
var acceptHeaderValue = context.Request.Headers["accept"];
```

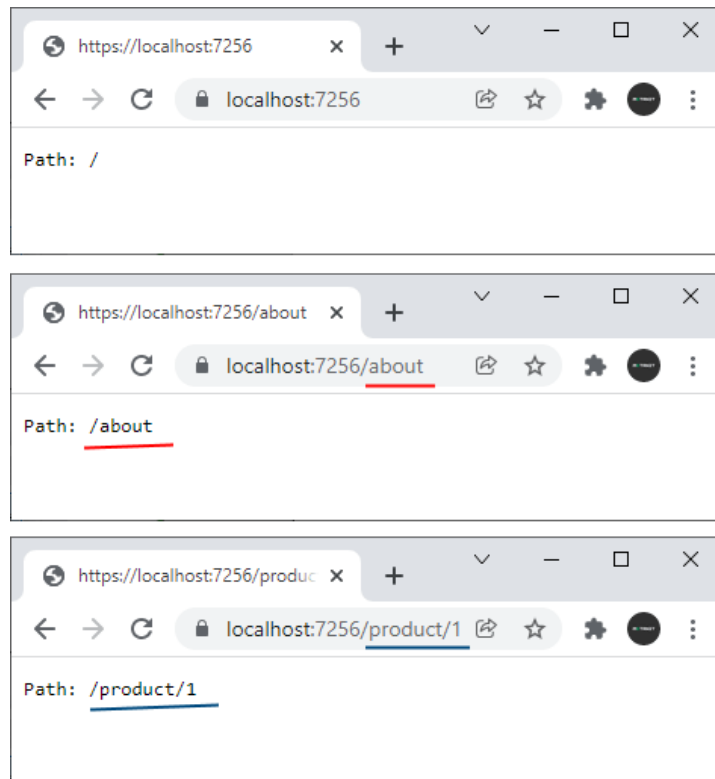
Для ряда заголовков в классе `HttpRequest` определены отдельные свойства: `Host`, `Method`, `ContentType`, `ContentLength`.

2.2. Получение пути запроса

Свойство **path** позволяет получить запрошенный путь, то есть адрес, к которому обращается клиент:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(async (context) =>
await context.Response.WriteAsync(
$"Path: {context.Request.Path}")
```

```
));
app.Run();
```



Это свойство позволяет нам узнать, по какому адресу обращается пользователь. Например, мы можем определить условную обработку запроса в зависимости от запрошенного адреса:

Свойство **path** позволяет получить запрошенный путь, то есть адрес, к которому обращается клиент:

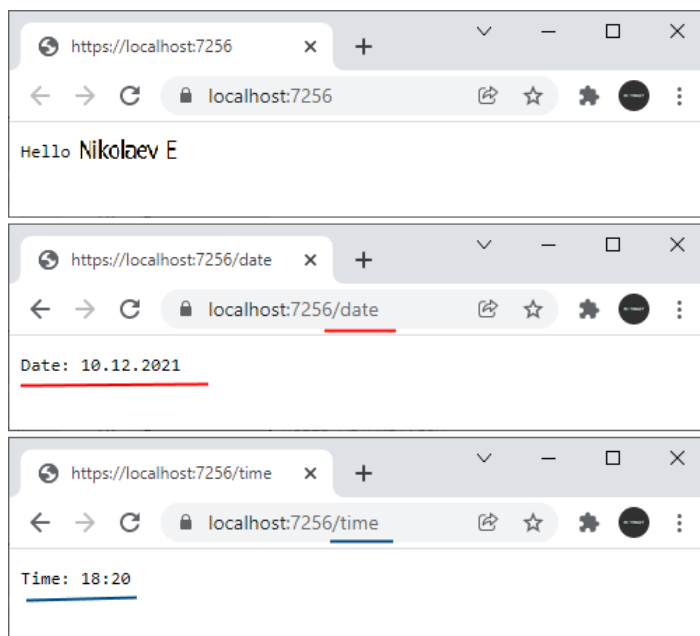
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Run(async (context) =>
{
    var path = context.Request.Path;
    var now = DateTime.Now;
    var response = context.Response;

    if (path=="/date")
        await response.WriteAsync(
            $"Date: {now.ToShortDateString()}");
    else if (path == "/time")
        await response.WriteAsync(
            $"Time: {now.ToShortTimeString()}");
    else
        await response.WriteAsync("Hello Nikolaev E.");
});
```

```
});
app.Run();
```

В данном случае, если пользователь обращается по адресу `"/date"`, то ему отображается текущая дата, а если обращается по адресу `"/time"` - текущее время. В остальных случаях отображается некоторое универсальное сообщение:



Подобным образом можно определить свою систему маршрутизации, однако в ASP.NET Core по умолчанию есть инструменты, которые проще использовать для создания системы маршрутизации в приложении.

2.3. Строка запроса

Свойство **QueryString** позволяет получить строку запроса. Строка запроса представляет ту часть запрошенного адреса, которая идет после символа `?` и представляет набор параметров, разделенных символом амперсанда `&`:

```
?параметр1=значение1&параметр2=значение2&параметр3=значение3
```

Каждому параметру с помощью знака равно передается некоторое значение. Стоит отметить, что строка запроса (query string) НЕ входит в путь запроса (path):

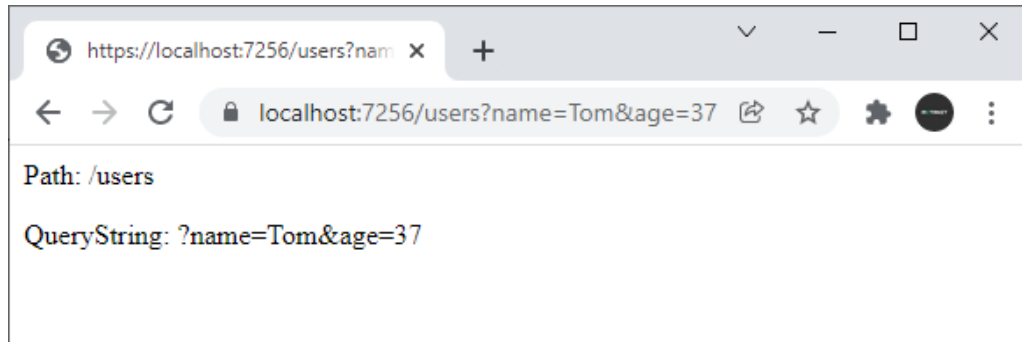
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
```

```

app.Run(async (context) =>
{
    context.Response.ContentType = "text/html; charset=utf-8";
    await context.Response.WriteAsync($"<p>Path:
{context.Request.Path}</p>" +
    $"<p>QueryString: {context.Request.QueryString}</p>");
});

app.Run();

```



Так, в данном случае идет обращение по адресу

`https://localhost:7256/users?name=Tom&age=37`

Путь запроса или `path` представляет ту часть адреса, которая идет после домена/порта и до символа `?`.

`/users`

Строка запроса или `query string` представляет ту часть адреса, которая идет начиная с символа `?`.

`?name=Tom&age=37`

То есть в данном случае через строку запроса передаются два параметра. Первый параметр - `name` имеет значение "Tom". Второй параметр - `age` имеет значение 37.

С помощью свойства **Query** можно получить все параметры строки запроса в виде словаря:

```

var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(async (context) =>
{
    context.Response.ContentType = "text/html; charset=utf-8";
    var stringBuilder
= new System.Text.StringBuilder(
"<h3>Параметры строки запроса</h3><table>");
    stringBuilder.Append(

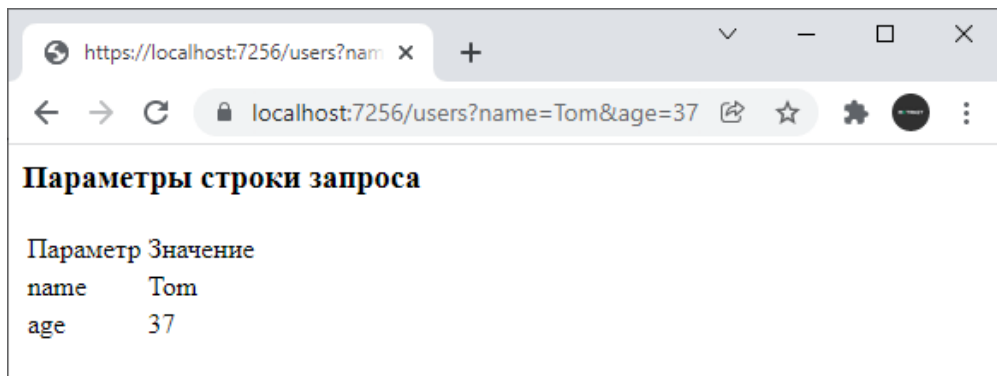
```

```

"<tr><td>Параметр</td><td>Значение</td></tr>");
    foreach (var param in context.Request.Query)
    {
        stringBuilder.Append(
$"<tr><td>{param.Key}</td><td>{param.Value}</td></tr>");
    }
    stringBuilder.Append("</table>");
    await
context.Response.WriteAsync(stringBuilder.ToString());
});

app.Run();

```



Соответственно можно вытащить из словаря Query значения отдельных параметров:

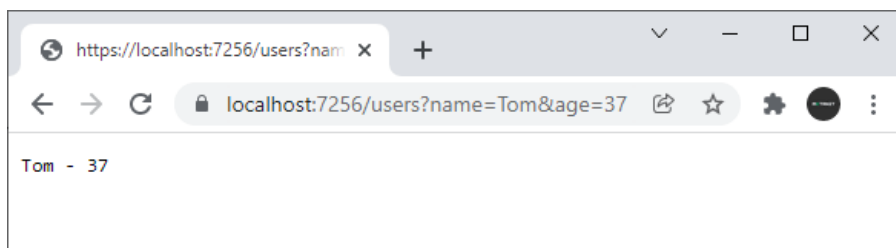
```

var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Run(async (context) =>
{
    string name = context.Request.Query["name"];
    string age = context.Request.Query["age"];
    await context.Response.WriteAsync($"{name} - {age}");
});

app.Run();

```



3. Методика и порядок выполнения работы

3.1. Учебная задача

Изучите код, представленный в теоретической части лабораторной работы.

3.2. Индивидуальное задание

1. Проанализируйте технологии, рассмотренные в теоретической части лабораторной работы.

2. Создайте приложение в соответствии с учебной задачей.

3. В соответствии с выбранной предметной областью, реализуйте приложение ASP.NET Core и продемонстрируйте применение следующих элементов, классов: HttpContext, HttpRequest (необходимо использовать как можно больше свойств HttpRequest в своем проекте).

4. Выполните установку статуса в своем приложении, установку заголовка и отправку html-содержимого в качестве ответа.

4. Контрольные вопросы

1. Что такое HttpRequest в ASP .NET Core?
2. Какие основные свойства объекта HttpRequest вы знаете?
3. Как получить доступ к объекту HttpRequest в контроллере?
4. Какие существуют способы обработки входящих запросов?
5. Как получить доступ к параметрам запроса?
6. Какие типы параметров запроса существуют?
7. Как работает привязка модели в Core?
8. Как обрабатывать сложные типы данных в параметрах запроса?
9. Как получить доступ к телу запроса?
10. Какие форматы данных можно получать через тело запроса?
11. Как работать с потоками данных в теле запроса?
12. Как обрабатывать multipart/form-data запросы?

13. Как работать с заголовками запроса?
14. Какие стандартные заголовки HTTP можно получить?
15. Как обрабатывать куки в запросе?
16. Как проверить подлинность запроса через заголовки?

ЛАБОРАТОРНАЯ РАБОТА 5. ОТПРАВКА ФОРМ

1. Цель и задачи

Цель лабораторной работы: создание эффективной системы веб-форм в ASP.NET Core, обеспечивающей корректную обработку пользовательских данных, их валидацию и безопасное взаимодействие с сервером.

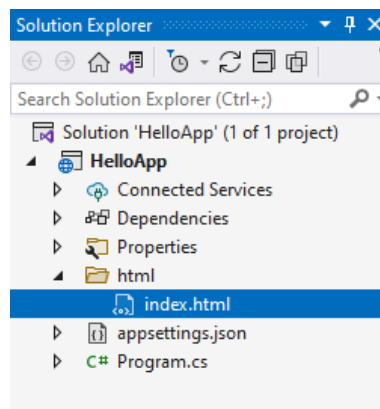
Задачи лабораторной работы:

1. Настройка маршрутов для обработки форм
2. Реализация клиентской валидации
3. Организация правильной структуры контроллера
4. Обработка POST-запросов.

2. Теоретическое обоснование

2.1. Основы форм

Часто данные отправляются на сервер с помощью форм html, обычно в запросе типа POST. Для получения подобных данных в классе **HttpRequest** определено свойство **Form**. Рассмотрим, как мы можем получить подобные данные. Прежде всего определим в проекте в папке **html** файл **index.html**



Определим в нем следующее содержимое:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Мой учебный проект</title>
```

```

</head>
<body>
  <h2>User form</h2>
  <form method="post" action="postuser">
    <p>Name: <input name="name" /></p>
    <p>Age: <input name="age" type="number" /></p>
    <input type="submit" value="Send" />
  </form>
</body>
</html>

```

Здесь определена форма условно для ввода данных пользователя, которая в запросе типа POST (атрибут `method="post"`) отправляет данные по адресу `"postuser"` (атрибут `action="postuser"`)

На форме определены два поля ввода. Первое поле предназначено для ввода имени пользователя. Второе поле - для ввода возраста пользователя.

Для получения этих данных определим в файле **Program.cs** следующий код:

```

var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(async (context) =>
{
    context.Response.ContentType = "text/html; charset=utf-8";
    if (context.Request.Path == "/postuser")
    {
        var form = context.Request.Form;
        string name = form["name"];
        string age = form["age"];
        await context.Response.WriteAsync(
            $"<div><p>Name: {name}</p><p>Age: {age}</p></div>");
    }
    else
    {
        await context.Response.SendFileAsync(
            "html/index.html");
    }
});
app.Run();

```

Здесь, если запрошен адрес `"/postuser"`, то предполагается, что отправлена некоторая форма. Сначала получаем отправленную форму в переменную `form`:

```
var form = context.Request.Form;
```

Свойство `Request.Form` возвращает объект **IFormCollection** – своего рода словарь, где по ключу можно получить значение элемента. При этом в

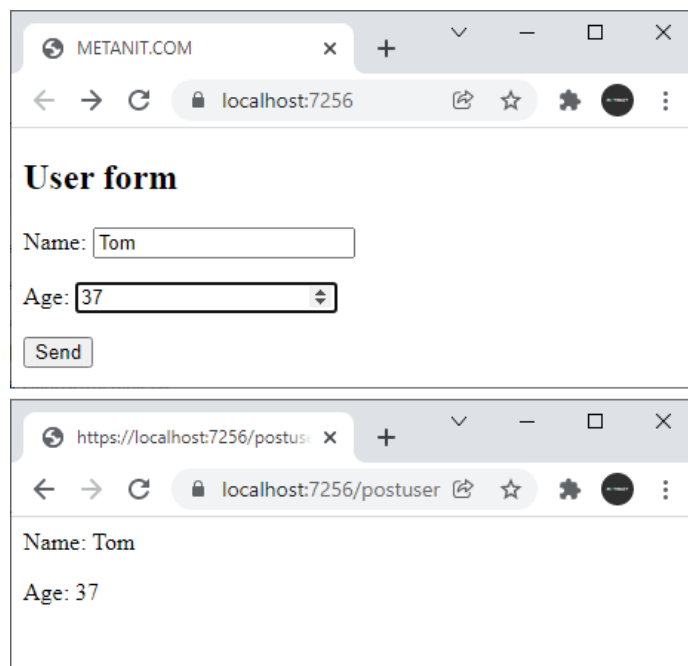
качестве ключей выступает названия полей форм (значения атрибутов name элементов формы):

```
<input name="age" type="number" />
```

Так, в данном случае название поля (значение атрибута name) равно "age". Соответственно в Request.Form по этому имени мы можем получить его значение:

```
string age = form["age"];
```

После получения данных формы они отправляются обратно клиенту:



2.2. Получение массивов

Усложним задачу и добавим в форму на странице **index.html** несколько полей, которые будут представлять массив:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title> Мой учебный проект </title>
</head>
<body>
  <h2>User form</h2>
  <form method="post" action="postuser">
    <p>Name: <br />
      <input name="name" />
    </p>
    <p>Age: <br />
      <input name="age" type="number" />
    </p>
```

```

    <p>
      Languages:<br />
      <input name="languages" /><br />
      <input name="languages" /><br />
      <input name="languages" /><br />
    </p>
    <input type="submit" value="Send" />
  </form>
</body>
</html>

```

Здесь добавлено три поля ввода, которые имеют одно и то же имя. Поэтому при их отправке будет формироваться массив из трех значений.

Теперь получим эти значения в коде C#:

```

var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Run(async (context) =>
{
    context.Response.ContentType = "text/html; charset=utf-8";

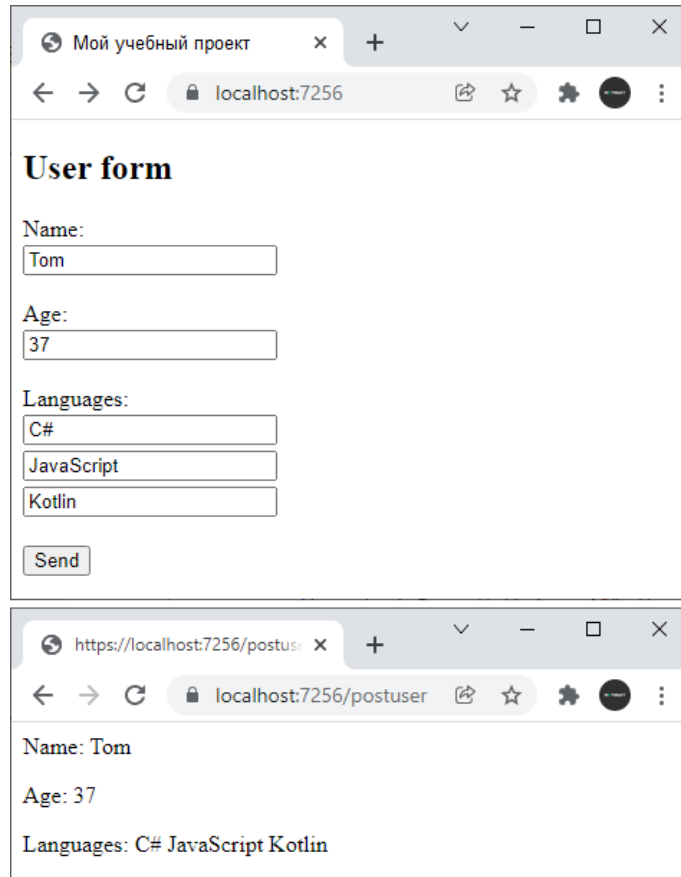
    if (context.Request.Path == "/postuser")
    {
        var form = context.Request.Form;
        string name = form["name"];
        string age = form["age"];
        string[] languages = form["languages"];
        // создаем из массива languages одну строку
        string langList = "";
        foreach (var lang in languages)
        {
            langList += $" {lang}";
        }
        await context.Response.WriteAsync(
            $"<div><p>Name: {name}</p>" +
            $"<p>Age: {age}</p>" +
            $"<div>Languages:{langList}</div></div>");
    }
    else
    {
        await context.Response.sendFileAsync(
            "html/index.html");
    }
});
app.Run();

```

Поскольку параметр "languages" представляет массив, то и сопоставляться он будет с массивом строк:

```
string[] languages = form["languages"];
```

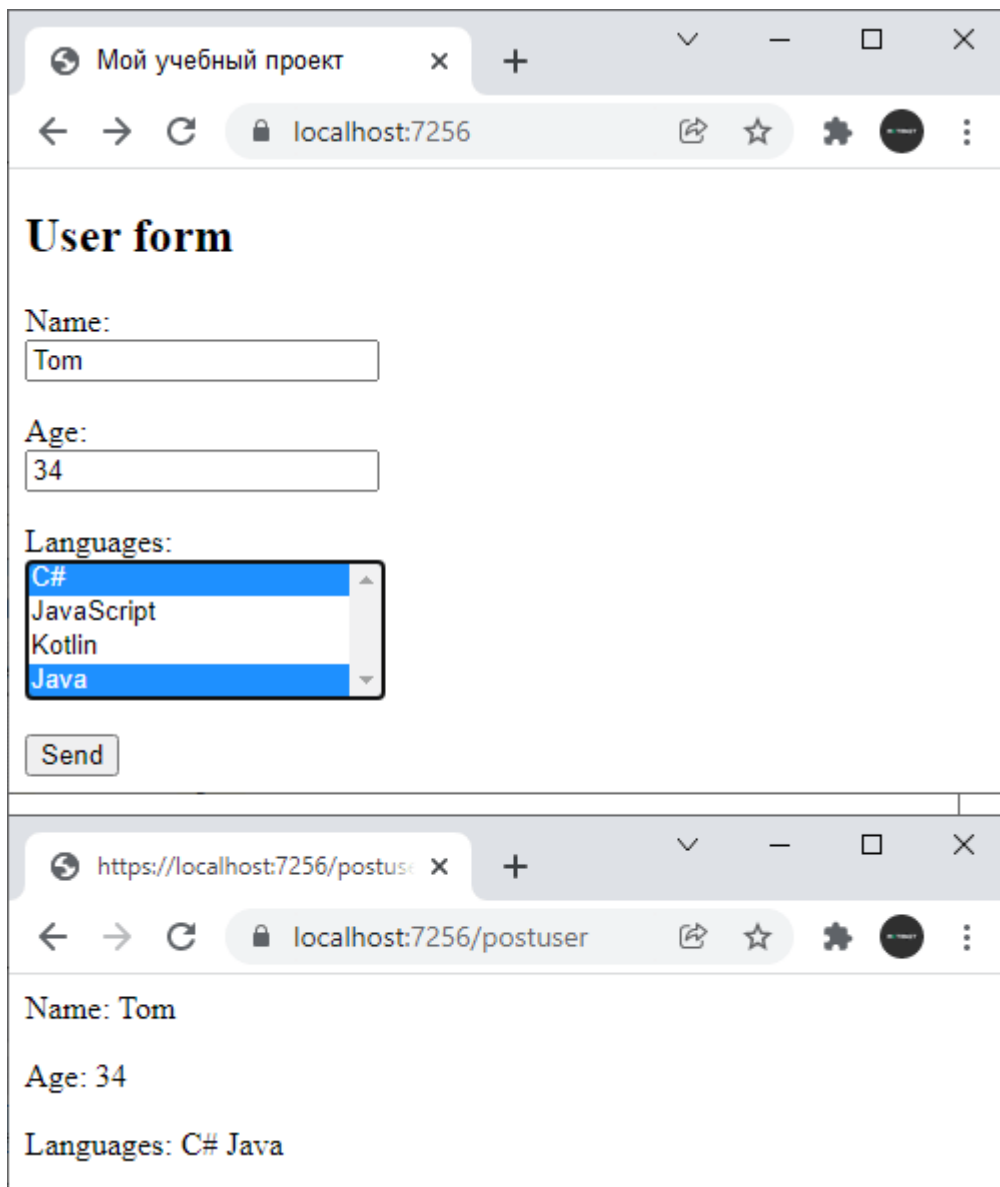
Для вывода на веб-страницу из этого массива формируется код html в виде строки:



Подобным образом можно передавать значения массива полей других типов, либо полей, которые представляют набор элементов, например, элемента **select**, который поддерживает множественный выбор:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>METANIT.COM</title>
</head>
<body>
  <h2>User form</h2>
  <form method="post" action="postuser">
    <p>Name: <br />
      <input name="name" />
    </p>
    <p>Age: <br />
      <input name="age" type="number" />
    </p>
    <p>
      Languages:<br />
      <select multiple name="languages">
```

```
        <option>C#</option>
        <option>JavaScript</option>
        <option>Kotlin</option>
        <option>Java</option>
    </select>
</p>
<input type="submit" value="Send" />
</form>
</body>
</html>
```



3. Методика и порядок выполнения работы

3.1. Учебная задача

Изучите код, представленный в теоретической части лабораторной работы.

3.2. Индивидуальное задание

1. Проанализируйте технологии, рассмотренные в теоретической части лабораторной работы.

2. Модифицируйте приложение, полученное в ходе выполнения предыдущих лабораторных работ. Для этого необходимо внедрить следующие возможности: отправка форм, отправка массивов объектов.

4. Контрольные вопросы

1. Что такое формы в ASP.NET Core?
2. Какие типы форм существуют в ASP.NET Core?
3. Как создать базовую форму в ASP.NET Core?
4. Какие основные компоненты входят в состав формы?
5. Как правильно структурировать HTML-разметку формы?
6. Какие атрибуты HTML5 используются для валидации форм?
7. Как работать с тегом `<form>` в ASP.NET Core?
8. Какие вспомогательные методы Tag Helpers используются для форм?
9. Как реализовать клиентскую валидацию форм?
10. Как настроить серверную валидацию?
11. Какие встроенные атрибуты валидации существуют?
12. Как создать собственные атрибуты валидации?

ЛАБОРАТОРНАЯ РАБОТА 6. ОТПРАВКА ФАЙЛОВ. ПЕРЕДАЧА ФАЙЛОВ НА СЕРВЕР

1. Цель и задачи

Цель лабораторной работы: Обеспечение безопасной и эффективной системы для работы с загрузкой файлов в ASP.NET Core, включая корректную обработку, хранение и защиту загружаемых данных..

Задачи лабораторной работы:

1. Создание форм для загрузки файлов
2. Настройка физического хранилища (файловая система)
3. Настройка маршрутов для обработки загрузки

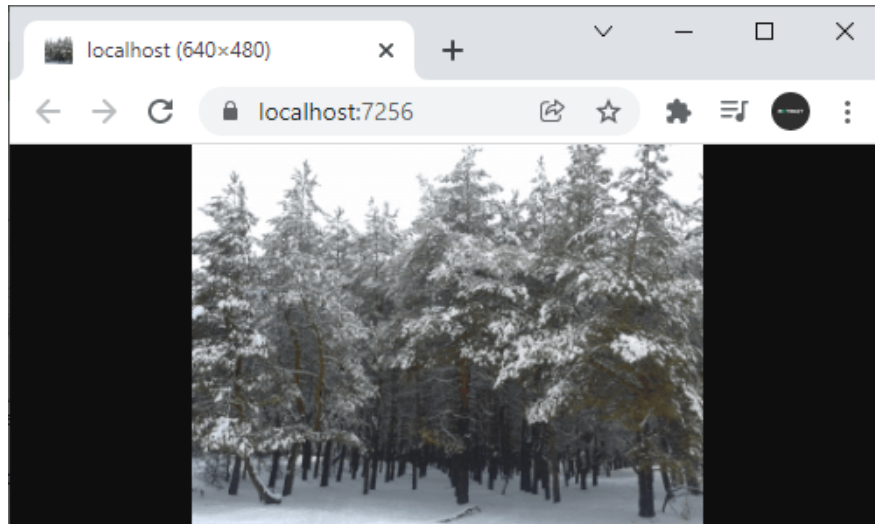
2. Теоретическое обоснование

2.1. Отправка файлов

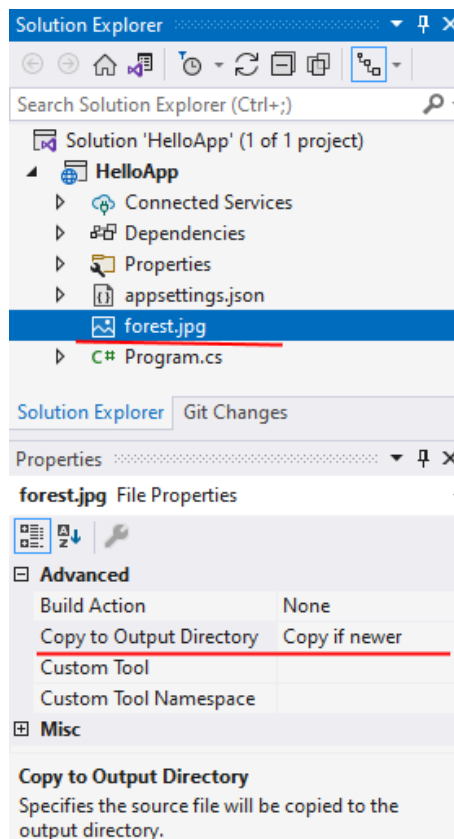
Для отправки файлов применяется метод **SendFileAsync()**, который получает либо путь к файлу в виде строки, либо информацию о файле в виде объекта **IFileInfo**. Например, допустим нам надо отправить файл по адресу "D:\\forest.jpg":

```
var builder = WebApplication.CreateBuilder();  
var app = builder.Build();  
app.Run(async (context)  
=> await context.Response.SendFileAsync("D:\\forest.jpg"));  
app.Run();
```

По умолчанию браузер попытается открыть файл. Так, в случае с изображениями они отображаются в браузере:



Также мы можем использовать относительные пути. Например, добавим в проект какой-нибудь файл (в моем случае это файл forest.jpg):



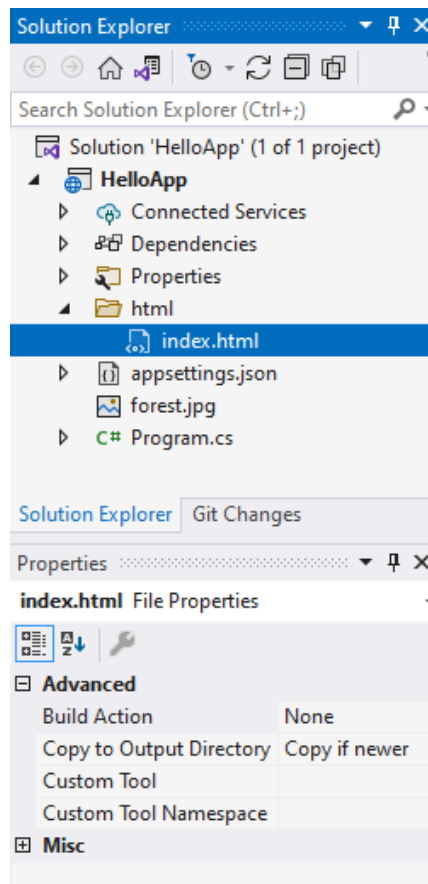
Для этого файла в окне свойств установим для опции **Copy to Output Directory** значение Copy if newer или Copy always, чтобы файл автоматически копировался в выходной каталог при построении приложения. И установим относительный путь относительно корня приложения:

```
var builder = WebApplication.CreateBuilder();
```

```
var app = builder.Build();
app.Run(async (context)
=> await context.Response.SendFileAsync("forest.jpg"));
app.Run();
```

2.2. Отправка html-страницы

Подобным образом мы можно отправлять и другие типы файлов, например, html-страницу. Так, определим в проекте новую папку, которую назовем **html**. В эту папку добавим новый файл **index.html**:



Определим в файле **index.html** следующий код:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Учебный проект</title>
</head>
<body>
  <h2>Hello ASP.NET Core!</h2>
</body>
</html>
```

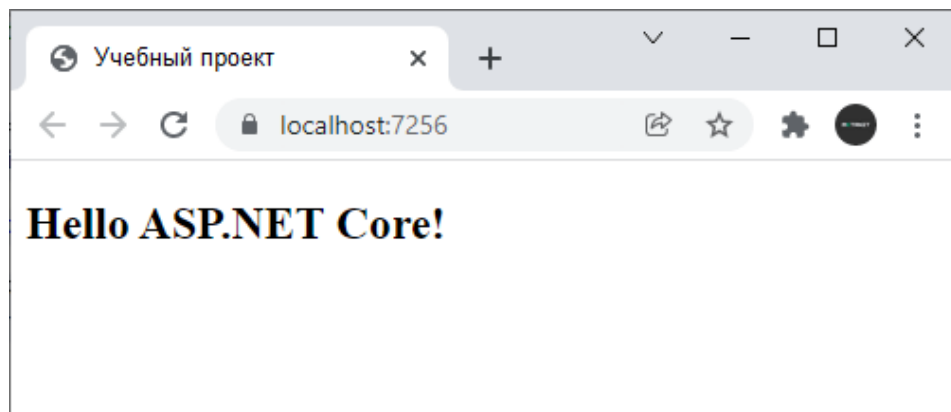
Определим для отправки веб-страницы следующий код:

```

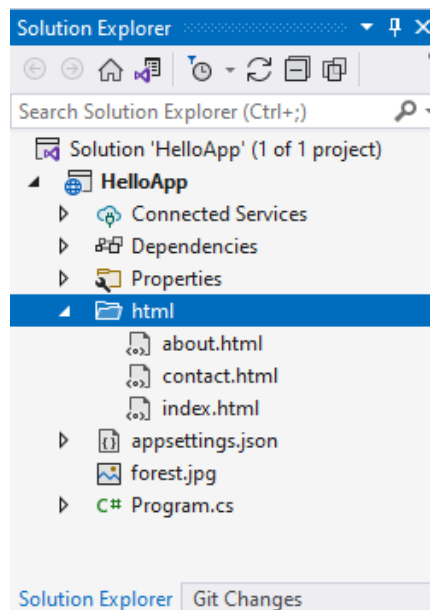
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(async (context) =>
{
    context.Response.ContentType
        = "text/html; charset=utf-8";
    await context.Response.SendFileAsync(
        "html/index.html");
});
app.Run();

```

В итоге при обращении к приложению сервер возвратит страницу `index.html`:



Теперь немного усложним задачу. Добавим в проект в папку `html` еще пару файлов. Назовем их, к примеру, `about.html` и `contact.html`.



Для отправки этих файлов определим следующий код:

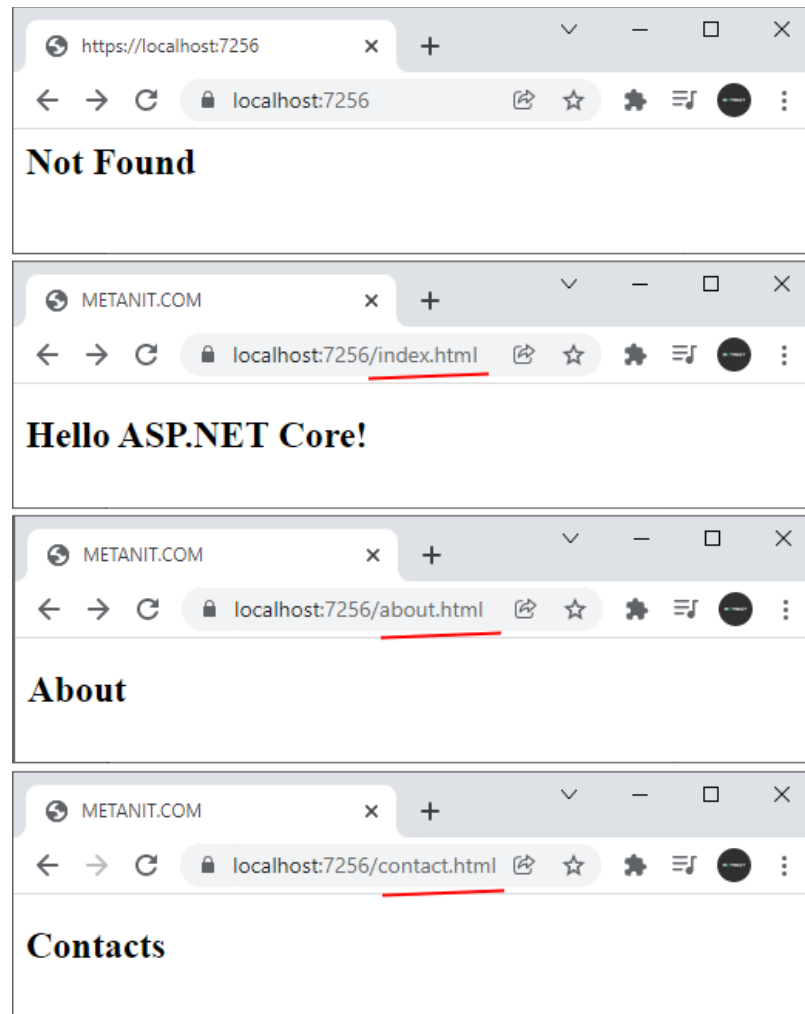
```

var builder = WebApplication.CreateBuilder();
var app = builder.Build();

```

```
app.Run(async(context) =>
{
    var path = context.Request.Path;
    var fullPath = $"html/{path}";
    var response = context.Response;
    response.ContentType = "text/html; charset=utf-8";
    if (File.Exists(fullPath))
    {
        await response.SendFileAsync(fullPath);
    }
    else
    {
        response.StatusCode = 404;
        await response.WriteAsync("<h2>Not Found</h2>");
    }
});
app.Run();
```

Когда приходит запрос, мы сопоставляем путь запроса (path) с файлами в папке html. То есть если path = about.html, то нам надо опрaвить в ответ файл about.html. При этом проверяем наличие файла. Если он есть в папке, то отправляем данный файл. Если нет, то отправляем статусный код 404 и сообщение, что ресурс не найден:



Стоит отметить, что в ASP.NET Core уже имеется встроенный middleware, который позволяет упростить работу со статическими файлами.

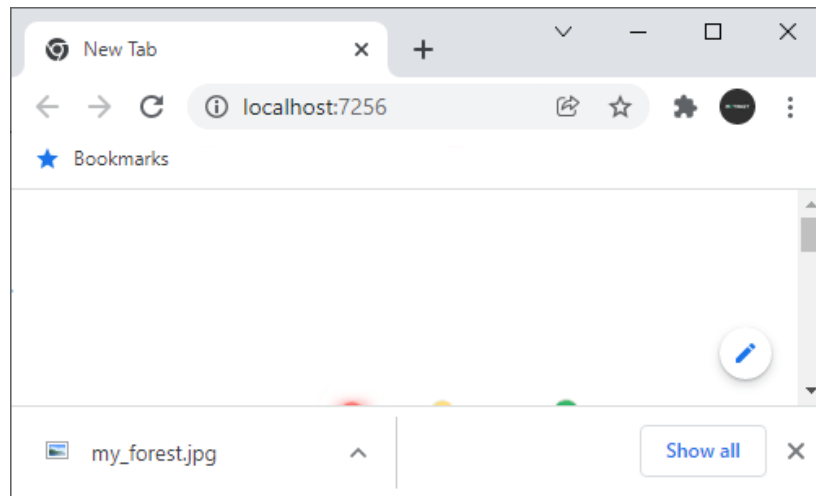
2.3. Загрузка файла

По умолчанию браузер пытается открыть отправляемый файл, что может быть полезно в случае файлов html - мы можем определить файл html и таким образом отправить клиенту веб-страницу. Но также может быть необходимо, чтобы браузер загружал файл без его открытия. В этом случае мы можем установить для заголовка "Content-Disposition" значение "attachment":

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(async (context) =>
{
    context.Response.Headers.ContentDisposition
        = "attachment; filename=my_forest.jpg";
    await context.Response.SendFileAsync("forest.jpg");
});
```

```
});
app.Run();
```

В этом случае загруженный файл получит имя "my_forest.jpg"



IFileInfo

В примерах выше применялась версия метода **SendFileAsync()**, которая получает путь к файлу в виде строки. Также можно использовать другую версию, которая получает информацию о файле в виде объекта **IFileInfo**:

```
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Run(async (context) =>
{
    var fileProvider = new PhysicalFileProvider(
        Directory.GetCurrentDirectory());
    var fileinfo = fileProvider.GetFileInfo("forest.jpg");

    context.Response.Headers.ContentDisposition =
        "attachment; filename=my_forest2.jpg";
    await context.Response.SendFileAsync(fileinfo);
});

app.Run();
```

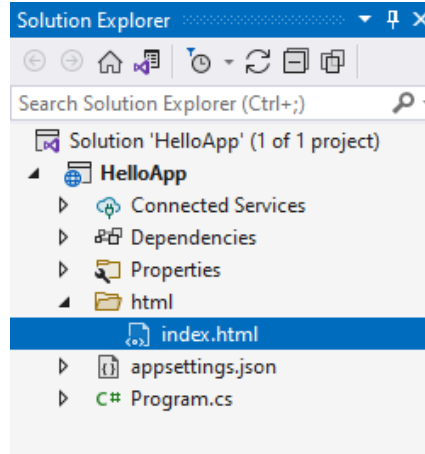
В этом случае сначала необходимо определить объект **PhysicalFileProvider**, конструктор которого получает каталог для поиска файлов. В его метод `fileProvider.GetFileInfo()` передается путь к файлу в рамках этого каталога. А результатом метода является объект **IFileInfo**, который передается в `SendFileAsync()`

2.4. Передача файлов на сервер

Рассмотрим, как загружать файлы на сервер в ASP.NET Core. Все загружаемые файлы в ASP.NET Core представлены типом **IFormFile** из пространства имен `Microsoft.AspNetCore.Http`. Соответственно для получения отправленного файла в контроллере необходимо использовать `IFormFile`. Затем с помощью методов `IFormFile` мы можем произвести различные манипуляции файлом - получить его свойства, сохранить, получить его поток и т.д. Некоторые его свойства и методы:

- `ContentType`: тип файла
- `FileName`: название файла
- `Length`: размер файла
- `CopyTo/CopyToAsync`: копирует файл в поток
- `OpenReadStream`: открывает поток файла для чтения

Для тестирования данной возможности определим в проекте папку **html**, в которой создадим файл **index.html**



Определим в файле **index.html** следующий код:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>Учебный проект</title>
</head>
<body>
  <h2>Выберите файл для загрузки</h2>
  <form action="upload" method="post">
```

```

        enctype="multipart/form-data">
        <input type="file" name="uploads" /><br>
        <input type="file" name="uploads" /><br>
        <input type="file" name="uploads" /><br>
        <input type="submit" value="Загрузить" />
    </form>
</body>
</html>

```

В данном случае форма содержит набор элементов с типом **file**, через которые можно выбрать файлы для загрузки. В данном случае на форме три таких элемента, но их может быть и меньше и больше. А благодаря установке атрибута формы **enctype="multipart/form-data"** браузер будет знать, что вместе с формой надо передать файлы.

Отправляться файлы будут в запросе типа POST на адрес `"/upload"`.

Теперь в файле **Program.cs** определим код, который будет получать загружаемые файлы:

```

var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Run(async (context) =>
{
    var response = context.Response;
    var request = context.Request;

    response.ContentType = "text/html; charset=utf-8";

    if (request.Path == "/upload" && request.Method=="POST")
    {
        IFormFileCollection files = request.Form.Files;

        // путь к папке, где будут храниться файлы
        var uploadPath =
            $"{Directory.GetCurrentDirectory()}/uploads";

        // создаем папку для хранения файлов
        Directory.CreateDirectory(uploadPath);

        foreach (var file in files)
        {
            // путь к папке uploads
            string fullPath = $"{uploadPath}/{file.FileName}";
            // сохраняем файл в папку uploads
            using (var fileStream = new FileStream(
                fullPath, FileMode.Create))
            {
                await file.CopyToAsync(fileStream);
            }
        }
    }
}

```

```

    }
    await response.WriteAsync("Файлы успешно загружены");
}
else
{
    await response.SendFileAsync("html/index.html");
}
});
app.Run();

```

Здесь если запрос приходит по адресу `"/upload"`, а сам запрос представляет запрос типа `POST`, то приложение получает коллекцию загруженных файлов с помощью свойства **`Request.Form.Files`**, которое представляет тип **`IFormFileCollection`**:

```
IFormFileCollection files = request.Form.Files;
```

Далее определяем каталог для загружаемых файлов (предполагается, что файлы будут храниться в каталоге `"uploads"`, которая располагается в папке приложения)

```
var uploadPath = $"{Directory.GetCurrentDirectory()}/uploads";
```

Если такой папки нет, то создаем ее. Затем перебираем всю коллекцию файлов.

```
foreach (var file in files)
```

Каждый отдельный файл в этой коллекции представляет тип `IFormFile`.

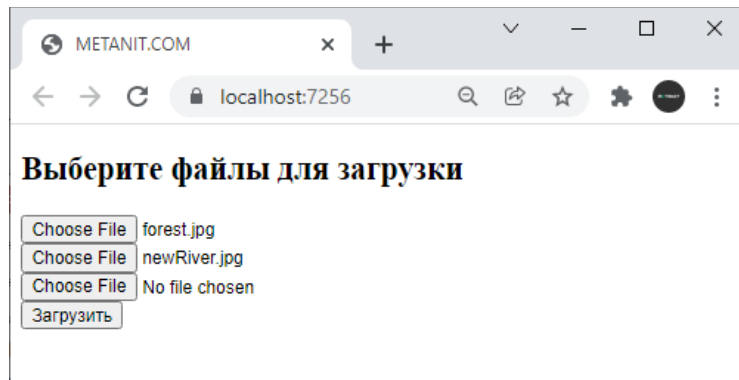
Для копирования файла в нужный каталог создается поток `FileStream`, в который записывается файл с помощью метода `CopyToAsync`.

```

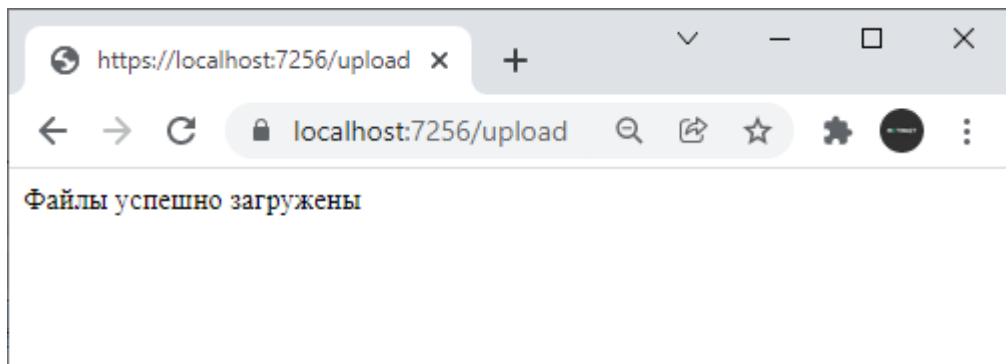
using (var fileStream = new FileStream(
    fullPath, FileMode.Create))
{
    await file.CopyToAsync(fileStream);
}

```

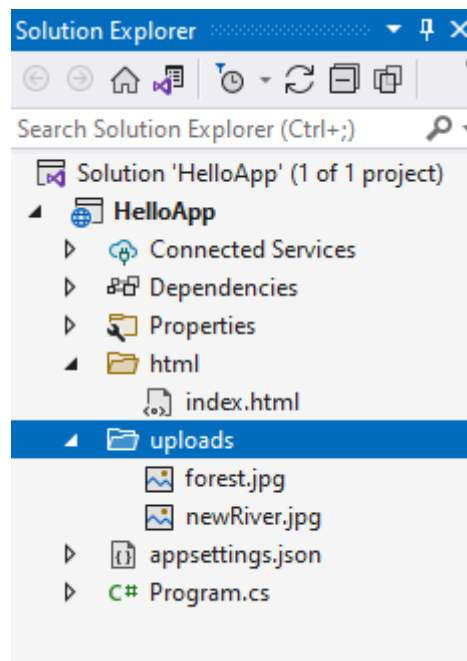
Если запрос идет по другому адресу и/или не представляет тип `POST`, то отправляем клиенту `html`-страницу **`index.html`**. Обратимся к приложению и выберем файлы для загрузки:



И после успешной загрузки нам отобразится соответствующее сообщение:



А в каталоге проекта будет создана папка uploads, в которой появятся загруженные файлы:



3. Методика и порядок выполнения работы

3.1. Учебная задача

Изучите код, представленный в теоретической части лабораторной работы.

3.2. Индивидуальное задание

1. Проанализируйте технологии, рассмотренные в теоретической части лабораторной работы.

2. Модифицируйте приложение, полученное в ходе выполнения предыдущих лабораторных работ. Для этого необходимо внедрить следующие возможности: отправка пользователю файла изображения, файла pdf, страницы html; отправка пользователем файла на сервер.

4. Контрольные вопросы

1. Поясните механизм отправки файла в качестве ответа. Опишите классы, которые будут задействованы в данном процессе.

2. Поясните механизм загрузки файла на сервер. Опишите классы, которые будут задействованы в данном процессе.

3. Как создать форму для загрузки файла в ASP.NET Core?

ЛАБОРАТОРНАЯ РАБОТА 7. СОЗДАНИЕ ПРОСТЕЙШЕГО API

1. Цель и задачи

Цель лабораторной работы: разработка веб-сервиса, который предоставляет доступ к ресурсам через унифицированные и стандартизированные интерфейсы.

Задачи лабораторной работы:

1. Проектирование архитектуры API.
2. Реализация RESTful-контроллеров, настройка маршрутизации.
3. Сериализация данных.

2. Теоретическое обоснование

Рассмотренного в прошлых темах материала достаточно для создания примитивного приложения. В этой теме попробуем реализовать простейшее приложение Web API в стиле REST. Архитектура REST предполагает применение следующих методов или типов запросов HTTP для взаимодействия с сервером, где каждый тип запроса отвечает за определенное действие:

- **GET** (получение данных)
- **POST** (добавление данных)
- **PUT** (изменение данных)
- **DELETE** (удаление данных)

Поскольку в приложении ASP.NET Core мы можем легко получить и адрес запроса и тип запроса, то реализовать подобную архитектуру не составит труда.

2.1. Создание сервера на ASP.NET Core

Вначале определим веб-приложение на ASP.NET Core, которое и будет собственно представлять Web API:

```
using System.Text.RegularExpressions;
```

```

// начальные данные
List<Person> users = new List<Person>
{
    new() { Id = Guid.NewGuid().ToString(),
           Name = "Tom",
           Age = 37 },
    new() { Id = Guid.NewGuid().ToString(),
           Name = "Bob",
           Age = 41 },
    new() { Id = Guid.NewGuid().ToString(),
           Name = "Sam",
           Age = 24 }
};

var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Run(async (context) =>
{
    var response = context.Response;
    var request = context.Request;
    var path = request.Path;

    // 2e752824-1657-4c7f-844b-6ec2e168e99c
    string expressionForGuid =
        @"^/api/users/\w{8}-\w{4}-\w{4}-\w{4}-\w{12}$";
    if (path == "/api/users" && request.Method=="GET")
    {
        await GetAllPeople(response);
    }
    else if (Regex.IsMatch(path, expressionForGuid)
             && request.Method == "GET")
    {
        // получаем id из адреса url
        string? id = path.Value?.Split("/")[3];
        await GetPerson(id, response);
    }
    else if (path == "/api/users"
             && request.Method == "POST")
    {
        await CreatePerson(response, request);
    }
    else if (path == "/api/users" && request.Method == "PUT")
    {
        await UpdatePerson(response, request);
    }
    else if (Regex.IsMatch(path, expressionForGuid)
             && request.Method == "DELETE")
    {
        string? id = path.Value?.Split("/")[3];
        await DeletePerson(id, response);
    }
    else

```

```

        {
            response.ContentType = "text/html; charset=utf-8";
            await response.SendFileAsync("html/index.html");
        }
    });

app.Run();

// получение всех пользователей
async Task GetAllPeople(HttpResponse response)
{
    await response.WriteAsJsonAsync(users);
}
// получение одного пользователя по id
async Task GetPerson(string? id, HttpResponse response)
{
    // получаем пользователя по id
    Person? user = users.FirstOrDefault((u) => u.Id == id);
    // если пользователь найден, отправляем его
    if (user != null)
        await response.WriteAsJsonAsync(user);
    // если не найден, отправляем статусный код
    // и сообщение об ошибке
    else
    {
        response.StatusCode = 404;
        await response.WriteAsJsonAsync(
            new {
                message = "Пользователь не найден" });
    }
}

async Task DeletePerson(string? id, HttpResponse response)
{
    // получаем пользователя по id
    Person? user = users.FirstOrDefault((u) => u.Id == id);
    // если пользователь найден, удаляем его
    if (user != null)
    {
        users.Remove(user);
        await response.WriteAsJsonAsync(user);
    }
    // если не найден, отправляем статусный код
    // и сообщение об ошибке
    else
    {
        response.StatusCode = 404;
        await response.WriteAsJsonAsync(
            new { message = "Пользователь не найден" });
    }
}

async Task CreatePerson(

```

```

        HttpResponseMessage response, HttpRequest request)
    {
        try
        {
            // получаем данные пользователя
            var user = await request.ReadFromJsonAsync<Person>();
            if (user != null)
            {
                // устанавливаем id для нового пользователя
                user.Id = Guid.NewGuid().ToString();
                // добавляем пользователя в список
                users.Add(user);
                await response.WriteAsJsonAsync(user);
            }
            else
            {
                throw new Exception("Некорректные данные");
            }
        }
        catch (Exception)
        {
            response.StatusCode = 400;
            await response.WriteAsJsonAsync(
                new { message = "Некорректные данные" });
        }
    }
}

```

```

async Task UpdatePerson(
    HttpResponseMessage response, HttpRequest request)
{
    try
    {
        // получаем данные пользователя
        Person? userData =
            await request.ReadFromJsonAsync<Person>();
        if (userData != null)
        {
            // получаем пользователя по id
            var user = users.FirstOrDefault(
                u => u.Id == userData.Id);
            // если пользователь найден,
            // изменяем его данные
            // и отправляем обратно клиенту
            if (user != null)
            {
                user.Age = userData.Age;
                user.Name = userData.Name;
                await response.WriteAsJsonAsync(user);
            }
            else
            {
                response.StatusCode = 404;
                await response.WriteAsJsonAsync(

```

```

        new { message = "Пользователь не найден" });
    }
    else
    {
        throw new Exception("Некорректные данные");
    }
}
catch (Exception)
{
    response.StatusCode = 400;
    await response.WriteAsJsonAsync(
        new { message = "Некорректные данные" });
}
}
public class Person
{
    public string Id { get; set; } = "";
    public string Name { get; set; } = "";
    public int Age { get; set; }
}

```

Разберем в общих чертах этот код. Вначале идет определение данных - список объектов `Person`, с которыми будут работать клиенты:

```

List<Person> users = new List<Person>
{
    new() { Id = Guid.NewGuid().ToString(),
           Name = "Tom",
           Age = 37 },
    new() { Id = Guid.NewGuid().ToString(),
           Name = "Bob",
           Age = 41 },
    new() { Id = Guid.NewGuid().ToString(),
           Name = "Sam",
           Age = 24 }
};

```

Стоит обратить внимание, что каждый объект `Person` имеет свойство `Id`, которое в качестве значения получает `Guid` - уникальный идентификатор, например "2e752824-1657-4c7f-844b-6ec2e168e99c".

Для упрощения данные определены в виде обычного списка объектов, но в реальной ситуации обычно подобные данные извлекаются из какой-нибудь базы данных.

В методе `app.Run()` определяем компонент `middleware`, который в зависимости от типа запросов (`GET/POST/PUT/DELETE`) выполняет те или иные действия.

Так, когда приложение получает запрос типа GET по адресу "api/users", то срабатывает следующий код:

```
if (path == "/api/users" && request.Method=="GET")
{
    await GetAllPeople(response);
}
// получение всех пользователей
async Task GetAllPeople(HttpResponse response)
{
    await response.WriteAsJsonAsync(users);
}
```

Запрос GET предполагает получение объектов, и в данном случае отправляем выше определенный список объектов Person.

Когда клиент обращается к приложению для получения одного объекта по id в запрос типа GET по адресу "api/users/[id]", то срабатывает следующий код:

```
else if (Regex.IsMatch(path,
    expressionForGuid) && request.Method == "GET")
{
    // получаем id из адреса url
    string? id = path.Value?.Split("/") [3];
    await GetPerson(id, response);
}

// получение одного пользователя по id
async Task GetPerson(string? id, HttpResponse response)
{
    // получаем пользователя по id
    Person? user = users.FirstOrDefault(
        (u) => u.Id == id);

    // если пользователь найден, отправляем его
    if (user != null)
        await response.WriteAsJsonAsync(user);
    // если не найден, отправляем статусный код
    // и сообщение об ошибке
    else
    {
        response.StatusCode = 404;
        await response.WriteAsJsonAsync(
            new { message = "Пользователь не найден" });
    }
}
```

Чтобы убедиться, что в запрошенном адресе после "/api/users/" указан id, проверяем соответствие адреса регулярному выражению: "`^/api/users/\w{8}-\w{4}-\w{4}-\w{12}$`". Данное выражение проверяет соответствие

последнего сегмента адреса значению Guid, который имеет формат xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

В этом случае нам надо найти нужного пользователя по Id в списке и отправить клиенту. Если же пользователь по Id не был найден, то возвращаем статусный код 404 с некоторым сообщением в формате JSON.

При получении запроса DELETE действует аналогичная логика:

```
else if (Regex.IsMatch(path, expressionForGuid)
        && request.Method == "DELETE")
{
    // получаем id из адреса url
    string? id = path.Value?.Split("/") [3];
    await DeletePerson(id, response);
}

async Task DeletePerson(string? id, HttpResponseMessage response)
{
    // получаем пользователя по id
    Person? user = users.FirstOrDefault((u) => u.Id == id);
    // если пользователь найден, удаляем его
    if (user != null)
    {
        users.Remove(user);
        await response.WriteAsJsonAsync(user);
    }
    // если не найден, отправляем статусный код
    // и сообщение об ошибке
    else
    {
        response.StatusCode = 404;
        await response.WriteAsJsonAsync(
            new { message = "Пользователь не найден" });
    }
}
```

Только в данном случае, если пользователь найден в списке, удаляем его из списка и посылаем клиенту.

При получении запроса с методом POST по адресу "/api/users" используем метод `request.ReadFromJsonAsync()` для извлечения данных из запроса:

```
else if (path == "/api/users" && request.Method == "POST")
{
    await CreatePerson(response, request);
}

async Task CreatePerson(HttpResponse response,
                        HttpRequest request)
```

```

{
    try
    {
        // получаем данные пользователя
        var user = await request.ReadFromJsonAsync<Person>();
        if (user != null)
        {
            // устанавливаем id для нового пользователя
            user.Id = Guid.NewGuid().ToString();
            // добавляем пользователя в список
            users.Add(user);
            await response.WriteAsJsonAsync(user);
        }
        else
        {
            throw new Exception("Некорректные данные");
        }
    }
    catch (Exception)
    {
        response.StatusCode = 400;
        await response.WriteAsJsonAsync(
            new { message = "Некорректные данные" });
    }
}

```

Поскольку при извлечении данных из запроса может произойти исключение (например, в результате парсинга в JSON), оборачиваем весь код в try..catch. И в случае успешного получения данных устанавливаем у нового объекта свойство Id, добавляем его в список users и отправляем обратно клиенту.

Если приложению приходит PUT-запрос, то также с помощью метода request.ReadFromJsonAsync() получаем отправленные клиентом данные. Если объект найден в списке, то изменяем его данные и отправляем обратно клиенту, иначе отправляем статусный код 404:

```

else if (path == "/api/users" && request.Method == "PUT")
{
    await UpdatePerson(response, request);
}
async Task UpdatePerson(HttpResponse response,
                        HttpRequest request)
{
    try
    {
        // получаем данные пользователя
        Person? userData
            = await request.ReadFromJsonAsync<Person>();
    }
}

```

```

if (userData != null)
{
    // получаем пользователя по id
    var user = users.FirstOrDefault(
        u => u.Id == userData.Id);
    // если пользователь найден, изменяем его данные
    // и отправляем обратно клиенту
    if (user != null)
    {
        user.Age = userData.Age;
        user.Name = userData.Name;
        await response.WriteAsJsonAsync(user);
    }
    else
    {
        response.StatusCode = 404;
        await response.WriteAsJsonAsync(
            new { message = "Пользователь не найден" });
    }
}
else
{
    throw new Exception("Некорректные данные");
}
}
catch (Exception)
{
    response.StatusCode = 400;
    await response.WriteAsJsonAsync(
        new { message = "Некорректные данные" });
}
}

```

В случае, если запрос идет по другому адресу, то отправляем клиенту веб-страницу **index.html**, которую мы далее определим:

```

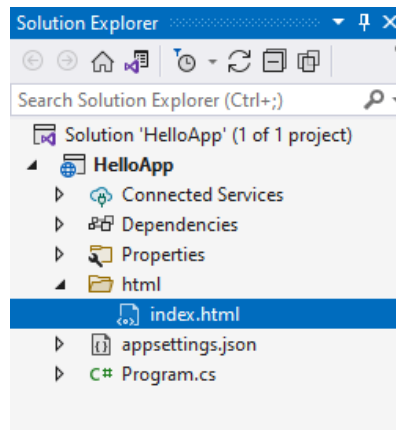
else
{
    response.ContentType = "text/html; charset=utf-8";
    await response.SendFileAsync("html/index.html");
}

```

Таким образом, мы определили простейший API. Теперь добавим код клиента.

2.2. Определение клиента

Теперь добавим в проект папку **html**, в которую добавим новый файл **index.html**



Определим в файле **index.html** следующим код для взаимодействия с сервером ASP.NET Core:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Мой проект</title>
</head>
<body>
  <h2>Список пользователей</h2>
  <div>
    <input type="hidden" id="userId" />
    <p>
      Имя:<br/>
      <input id="userName" />
    </p>
    <p>
      Возраст:<br />
      <input id="userAge" type="number" />
    </p>
    <p>
      <button id="saveBtn">Сохранить</button>
      <button id="resetBtn">Сбросить</button>
    </p>
  </div>
  <table>
    <thead>
      <tr>
        <th>Имя</th>
        <th>Возраст</th>
        <th></th>
      </tr>
    </thead>
    <tbody>
```

```

    </tbody>
</table>

<script>
// Получение всех пользователей
    async function getUsers() {
        // отправляет запрос и получаем ответ
        const response = await fetch("/api/users", {
            method: "GET",
            headers: { "Accept": "application/json" }
        });
        // если запрос прошел нормально
        if (response.ok === true) {
            // получаем данные
            const users = await response.json();
            const rows = document.querySelector("tbody");
            // добавляем полученные элементы в таблицу
            users.forEach(user => rows.append(row(user)));
        }
    }
// Получение одного пользователя
    async function getUser(id) {
        const response = await fetch(`/api/users/${id}`, {
            method: "GET",
            headers: { "Accept": "application/json" }
        });
        if (response.ok === true) {
            const user = await response.json();
            document.getElementById("userId").value =
user.id;
            document.getElementById("userName").value =
user.name;
            document.getElementById("userAge").value =
user.age;
        }
        else {
            // если произошла ошибка,
            // получаем сообщение об ошибке
            const error = await response.json();
            console.log(error.message); // и выводим
        }
    }
// Добавление пользователя
    async function createUser(userName, userAge) {

        const response = await fetch("api/users", {
            method: "POST",
            headers: {
                "Accept": "application/json",
                "Content-Type": "application/json" },
            body: JSON.stringify({
                name: userName,
                age: parseInt(userAge, 10)
            })
        });
    }

```

```

    })
  });
  if (response.ok === true) {
    const user = await response.json();
    document.querySelector("tbody").append(
      row(user));
  }
  else {
    const error = await response.json();
    console.log(error.message);
  }
}
// Изменение пользователя
async function editUser(userId, userName, userAge) {
  const response = await fetch("api/users", {
    method: "PUT",
    headers: {
      "Accept": "application/json",
      "Content-Type": "application/json" },
    body: JSON.stringify({
      id: userId,
      name: userName,
      age: parseInt(userAge, 10)
    })
  });
  if (response.ok === true) {
    const user = await response.json();
    document.querySelector(`tr[data-
rowid='${user.id}']`).replaceWith(row(user));
  }
  else {
    const error = await response.json();
    console.log(error.message);
  }
}
// Удаление пользователя
async function deleteUser(id) {
  const response = await fetch(`/api/users/${id}`, {
    method: "DELETE",
    headers: { "Accept": "application/json" }
  });
  if (response.ok === true) {
    const user = await response.json();
    document.querySelector(`tr[data-
rowid='${user.id}']`).remove();
  }
  else {
    const error = await response.json();
    console.log(error.message);
  }
}

// сброс данных формы после отправки

```

```

function reset() {
    document.getElementById("userId").value =
    document.getElementById("userName").value =
    document.getElementById("userAge").value = "";
}
// создание строки для таблицы
function row(user) {

    const tr = document.createElement("tr");
    tr.setAttribute("data-rowid", user.id);

    const nameTd = document.createElement("td");
    nameTd.append(user.name);
    tr.append(nameTd);

    const ageTd = document.createElement("td");
    ageTd.append(user.age);
    tr.append(ageTd);

    const linksTd = document.createElement("td");

    const editLink = document.createElement("button");
    editLink.append("Изменить");
    editLink.addEventListener("click",
        async () => await getUser(user.id));
    linksTd.append(editLink);

    const removeLink = document.createElement("button");
    removeLink.append("Удалить");
    removeLink.addEventListener("click",
        async () => await deleteUser(user.id));

    linksTd.append(removeLink);
    tr.appendChild(linksTd);

    return tr;
}
// сброс значений формы
document.getElementById("resetBtn").addEventListener("click", () => reset());

// отправка формы
document.getElementById("saveBtn").addEventListener(
    "click", async () => {
    const id = document.getElementById("userId").value;
    const name =
        document.getElementById("userName").value;
    const age =
        document.getElementById("userAge").value;
    if (id === "")
        await createUser(name, age);
    else
        await editUser(id, name, age);
}

```

```

        reset();
    });

    // загрузка пользователей
    getUsers();
</script>
</body>
</html>

```

Основная логика здесь заключена в коде javascript. При загрузке страницы в браузере получаем все объекты из БД с помощью функции `getUsers()`:

```

async function getUsers() {
    // отправляет запрос и получаем ответ
    const response = await fetch("/api/users", {
        method: "GET",
        headers: { "Accept": "application/json" }
    });
    // если запрос прошел нормально
    if (response.ok === true) {
        // получаем данные
        const users = await response.json();
        const rows = document.querySelector("tbody");
        // добавляем полученные элементы в таблицу
        users.forEach(user => rows.append(row(user)));
    }
}

```

Для добавления строк в таблицу используется функция `row()`, которая возвращает строку. В этой строке будут определены ссылки для изменения и удаления пользователя.

Ссылка для изменения пользователя с помощью функции `getUser()` получает с сервера выделенного пользователя:

```

async function getUser(id) {
    const response = await fetch(`/api/users/${id}`, {
        method: "GET",
        headers: { "Accept": "application/json" }
    });
    if (response.ok === true) {
        const user = await response.json();
        document.getElementById("userId").value = user.id;
        document.getElementById("userName").value = user.name;
        document.getElementById("userAge").value = user.age;
    }
    else {
        // если произошла ошибка, получаем сообщение об ошибке
        const error = await response.json();
        console.log(error.message); // и выводим его на консоль
    }
}

```

}

И выделенный пользователь добавляется в форму над таблицей. Эта же форма применяется и для добавления объекта. С помощью скрытого поля, которое хранит `id` пользователя, мы можем узнать, какое действие выполняется - добавление или редактирование. Если `id` не установлен (равен пустой строке), то выполняется функция `createUser`, которая отправляет данные в POST-запросе:

```

async function createUser(userName, userAge) {

  const response = await fetch("api/users", {
    method: "POST",
    headers: {
      "Accept": "application/json",
      "Content-Type": "application/json" },
    body: JSON.stringify({
      name: userName,
      age: parseInt(userAge, 10)
    })
  });
  if (response.ok === true) {
    const user = await response.json();
    document.querySelector("tbody").append(row(user));
  }
  else {
    const error = await response.json();
    console.log(error.message);
  }
}

```

Если же ранее пользователь был загружен на форму, и в скрытом поле сохранился его `id`, то выполняется функция `editUser`, которая отправляет PUT-запрос:

```

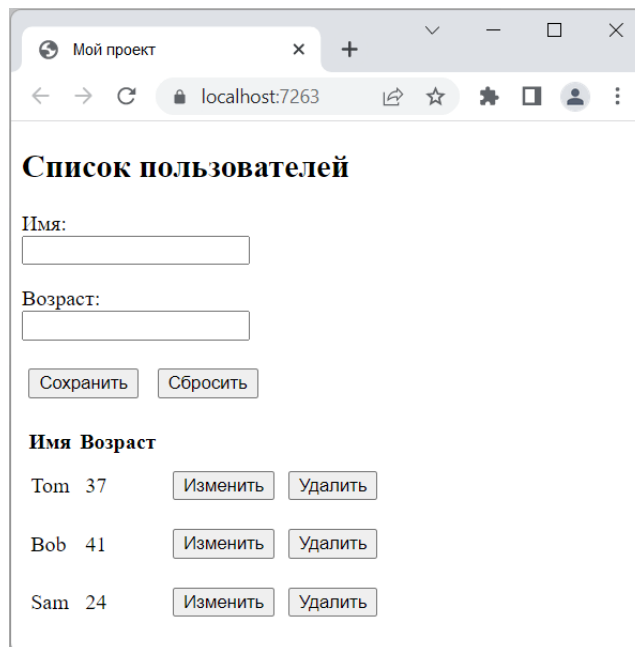
async function editUser(userId, userName, userAge) {
  const response = await fetch("api/users", {
    method: "PUT",
    headers: {
      "Accept": "application/json",
      "Content-Type": "application/json" },
    body: JSON.stringify({
      id: userId,
      name: userName,
      age: parseInt(userAge, 10)
    })
  });
  if (response.ok === true) {
    const user = await response.json();
    document.querySelector(

```

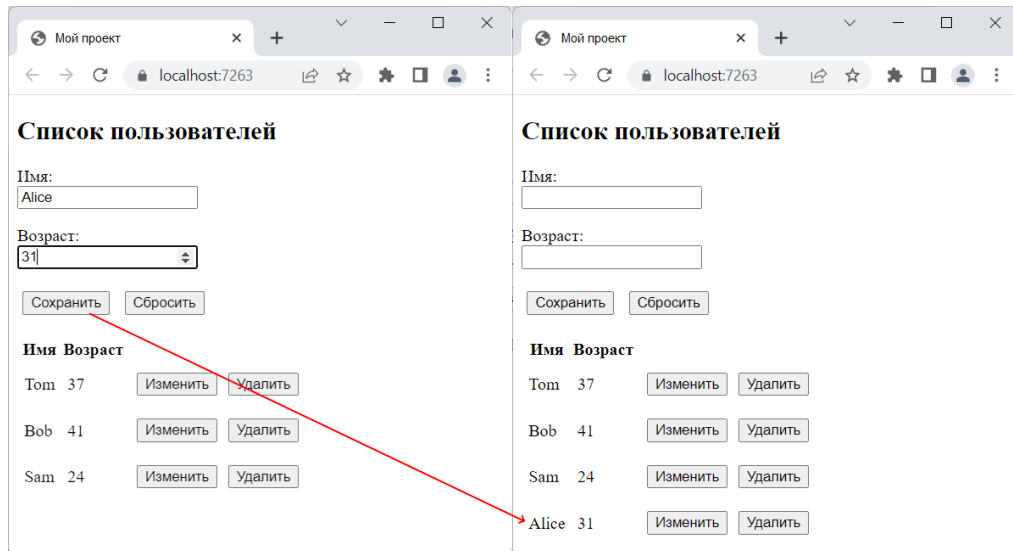
```
        `tr[data-rowid=  
            '${user.id}']`).replaceWith(row(user));  
    }  
    else {  
        const error = await response.json();  
        console.log(error.message);  
    }  
}
```

И функция `deleteUser()` посылает на сервер запрос типа DELETE на удаление пользователя, и при успешном удалении на сервере удаляет объект по `id` из списка объектов `Person`.

Теперь запустим проект, и по умолчанию приложение отправит браузеру веб-страницу **index.html**, которая загрузит список объектов:



После этого мы сможем выполнять все базовые операции с пользователями – получение, добавление, изменение, удаление. Например, добавим нового пользователя:



3. Методика и порядок выполнения работы

3.1. Учебная задача

Изучите код, представленный в теоретической части лабораторной работы.

3.2. Индивидуальное задание

1. Проанализируйте технологии, рассмотренные в теоретической части лабораторной работы.

2. Модифицируйте приложение, полученное в ходе выполнения предыдущих лабораторных работ. Для этого необходимо внедрить следующие возможности: разработать API и реализовать обработку запросов различного типа (GET/POST/PUT/DELETE) для выполнения операций с данными.

4. Контрольные вопросы

Что такое API в контексте ASP.NET Core?

Какие принципы лежат в основе RESTful API?

Что такое URI в контексте API?

Какие HTTP методы используются в REST API?

Что такое модель в контексте API?

Что такое контроллер в контексте API?

Что такое действие контроллера?

Что такое сериализация и десериализация данных в контексте API?

Какие преимущества предоставляет REST API?

Как настроить маршрутизацию в ASP.NET Core для API?

ЛАБОРАТОРНАЯ РАБОТА 8. МЕХАНИЗМ DEPENDENCY INJECTION

1. Цель и задачи

Цель лабораторной работы: изучить принципы и подходы к внедрению зависимостей в веб-приложениях на платформе ASP.NET Core.

Задачи:

1. Изучение основных понятий и терминов, связанных с внедрением зависимостей.
2. Изучение различных типов внедрения зависимостей (Constructor Injection, Method Injection, Property Injection).
3. Ознакомление с инструментами и библиотеками, используемыми для внедрения зависимостей в ASP.NET Core.

2. Теоретическое обоснование

2.1. Внедрение зависимостей и IServiceCollection

Dependency injection (DI) или внедрение зависимостей представляет механизм, который позволяет сделать взаимодействующие в приложении объекты слабосвязанными. Такие объекты связаны между собой через абстракции, например, через интерфейсы, что делает всю систему более гибкой, более адаптируемой и расширяемой.

В центре подобного механизма находится понятие **зависимость** - некоторая сущность, от которой зависит другая сущность. Например:

```
class Logger
{
    public void Log(string message) =>
        Console.WriteLine(message);
}
class Message
{
    Logger logger = new Logger();
    public string Text { get; set; } = "";
    public void Print() => logger.Log(Text);
}
```

Здесь сущность Message, которая представляет некоторое сообщение, зависит от другой сущности - Logger, которая представляет логгер. В методе Print() класса Message имитируется логгирование текста сообщения путем вызова у объекта Logger метода Log, который выводит сообщение на консоль. Однако здесь класс Message тесно связан с классом Logger. Класс Message отвечает за создание объекта Logger. Это имеет ряд недостатков. Прежде всего, если мы захотим вместо класса Logger использовать другой тип логгера, например, логгировать в файл, а не на консоль, то нам придется менять класс Message. Один класс не составит труда поменять, но если в проекте таких классов много, то поменять во всех класс Logger на другой будет труднее. Кроме того, класс Logger может иметь свои зависимости, которые тоже может потребоваться поменять. В итоге такими системами сложнее управлять и сложнее тестировать.

Чтобы отвязать объект Logger от класса Message, мы можем создать абстракцию, которая будет представлять логгер, и передавать ее извне в объект Message:

```
interface ILogger
{
    void Log(string message);
}
class Logger : ILogger
{
    public void Log(string message) =>
        Console.WriteLine(message);
}
class Message
{
    ILogger logger;
    public string Text { get; set; } = "";
    public Message(ILogger logger)
    {
        this.logger = logger;
    }
    public void Print() => logger.Log(Text);
}
```

Теперь класс Message не зависит от конкретной реализации класса Logger - это может быть любая реализация интерфейса ILogger. Кроме того, создание объекта логгера выносится во внешний код. Класс Message больше

ничего не знает о логгере кроме того, что у него есть метод `Log`, который позволяет логгировать его текст.

Тем не менее остается проблема управления подобными зависимостями, особенно если это касается больших приложений. Нередко для установки зависимостей в подобных системах используются специальные контейнеры - IoC-контейнеры (Inversion of Control). Такие контейнеры служат своего рода фабриками, которые устанавливают зависимости между абстракциями и конкретными объектами и, как правило, управляют созданием этих объектов.

Преимуществом ASP.NET Core в этом отношении является то, что фреймворк уже по умолчанию имеет встроенный контейнер внедрения зависимостей, который представлен интерфейсом **IServiceProvider**. А сами зависимости еще называются сервисами, собственно поэтому контейнер можно назвать **провайдером сервисов**. Этот контейнер отвечает за сопоставление зависимостей с конкретными типами и за внедрение зависимостей в различные объекты.

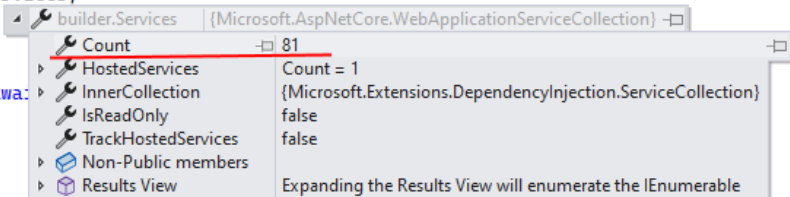
2.2. Установка встроенных сервисов фреймворка

За управление сервисами в приложении в классе **WebApplicationBuilder** определено свойство **Services**, которое представляет объект **IServiceCollection** - коллекцию сервисов:

```
WebApplicationBuilder builder
    = WebApplication.CreateBuilder();
IServiceCollection allServices
    = builder.Services; // коллекция сервисов
```

И даже если мы не добавляем в эту коллекцию никаких сервисов, **IServiceCollection** уже содержит ряд сервисов по умолчанию

```
var builder = WebApplication.CreateBuilder();
var allServices = builder.Services;
var app = builder.Build();
app.Run(async (context) => await...);
app.Run();
```



Property	Value
Count	81
HostedServices	Count = 1
InnerCollection	{Microsoft.Extensions.DependencyInjection.ServiceCollection}
IsReadOnly	false
TrackHostedServices	false
Non-Public members	
Results View	Expanding the Results View will enumerate the IEnumerable

Как видно на скриншоте, в коллекции `IServiceCollection` 81 сервис, который мы можем использовать в приложении. Это такие сервисы, как **`ILogger<T>`**, **`ILoggerFactory`**, **`IWebHostEnvironment`** и ряд других. Они добавляются по умолчанию инфраструктурой ASP.NET Core. И мы их можем использовать в различных частях приложения.

2.3. Информация о сервисах

Каждый сервис в коллекции `IServiceCollection` представляет объект **`ServiceDescriptor`**, который несет некоторую информацию. В частности, наиболее важные свойства этого объекта:

- **`ServiceType`**: тип сервиса
- **`ImplementationType`**: тип реализации сервиса
- **`Lifetime`**: жизненный цикл сервиса

Например, получим все сервисы, которые добавлены в приложение:

```
using System.Text;
var builder = WebApplication.CreateBuilder();
var services = builder.Services;
var app = builder.Build();

app.Run(async context =>
{
    var sb = new StringBuilder();
    sb.Append("<h1>Все сервисы</h1>");
    sb.Append("<table>");
    sb.Append(
"<tr><th>Тип</th><th>Lifetime</th><th>Реализация</th></tr>");
    foreach (var svc in services)
    {
        sb.Append("<tr>");
        sb.Append($"<td>{svc.ServiceType.FullName}</td>");
        sb.Append($"<td>{svc.Lifetime}</td>");
        sb.Append($"<td>{svc.ImplementationType?.FullName}</td>"
);
        sb.Append("</tr>");
    }
    sb.Append("</table>");
    context.Response.ContentType = "text/html;charset=utf-8";
    await context.Response.WriteAsync(sb.ToString());
});
app.Run();
```

Тип	Lifetime	Реализация
Microsoft.Extensions.Hosting.IHostingEnvironment	Singleton	
Microsoft.Extensions.Hosting.IHostEnvironment	Singleton	
Microsoft.Extensions.Hosting.IHostBuilderContext	Singleton	
Microsoft.Extensions.Configuration.IConfiguration	Singleton	
Microsoft.Extensions.Hosting.IApplicationLifetime	Singleton	
Microsoft.Extensions.Hosting.IHostApplicationLifetime	Singleton	Microsoft.Extensions.Hosting.Internal.ApplicationLifetime
Microsoft.Extensions.Hosting.IHostLifetime	Singleton	Microsoft.Extensions.Hosting.Internal.ConsoleLifetime
Microsoft.Extensions.Hosting.IHost	Singleton	
Microsoft.Extensions.Options.IOptions`1	Singleton	Microsoft.Extensions.Options.UnnamedOptionsManager`1
Microsoft.Extensions.Options.IOptionsSnapshot`1	Scoped	Microsoft.Extensions.Options.OptionsManager`1
Microsoft.Extensions.Options.IOptionsMonitor`1	Singleton	Microsoft.Extensions.Options.OptionsMonitor`1

2.4. Регистрация встроенных сервисов ASP.NET Core

Кроме ряда подключаемых по умолчанию сервисов ASP.NET Core имеет еще ряд встроенных сервисов, которые мы можем подключать в приложение при необходимости. Все сервисы и компоненты middleware, которые предоставляются ASP.NET по умолчанию, регистрируются в приложение с помощью методов расширений IServiceCollection, имеющих общую форму Add[название_сервиса].

Например:

```
var builder = WebApplication.CreateBuilder();
builder.Services.AddMvc();
```

Для объекта IServiceCollection определено ряд методов расширений, которые начинаются на Add, как, например, AddMvc(). Эти методы добавляют в объект IServiceCollection соответствующие сервисы. Например, AddMvc() добавляет в приложение сервисы MVC, благодаря чему мы сможем их использовать в приложении.

2.5. Создание сервисов

Фреймворк ASP.NET Core предоставляет ряд встроенных сервисов, которые мы можем использовать. Но также мы можем создавать свои собственные сервисы. Рассмотрим, как это сделать.

Определим новый интерфейс **ITimeService**, который предназначен для получения времени:

```
interface ITimeService
{
    string GetTime();
}
```

И также определим два класса, которые будут реализовать данный интерфейс. Первый класс будет называться **ShortTimeService** и будет возвращать текущее время в формате hh:mm(то есть часы и минуты):

```
// время в формате hh:mm
class ShortTimeService : ITimeService
{
    public string GetTime() =>
        DateTime.Now.ToShortTimeString();
}
```

Второй класс будет называться **LongTimeService** – он будет возвращать время в формате hh:mm:ss:

```
// время в формате hh:mm:ss
class LongTimeService : ITimeService
{
    public string GetTime()
        => DateTime.Now.ToLongTimeString();
}
```

Теперь добавим в коллекцию сервисов сервис **ITimeService** и используем его в приложении:

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddTransient<ITimeService,
ShortTimeService>();

var app = builder.Build();

app.Run(async context =>
{
    var timeService = app.Services.GetService<ITimeService>();
    await context.Response.WriteAsync($"Time:
{timeService?.GetTime()}");
});

app.Run();

interface ITimeService
{
    string GetTime();
}
// время в формате hh:mm
```

```

class ShortTimeService : ITimeService
{
    public string GetTime() =>
DateTime.Now.ToShortTimeString();
}
// время в формате hh:mm:ss
class LongTimeService : ITimeService
{
    public string GetTime() => DateTime.Now.ToLongTimeString();
}

```

Здесь надо выделить два момента. Во-первых, добавление сервиса в коллекцию сервисов приложения:

```

builder.Services.AddTransient<ITimeService,
ShortTimeService>();

```

Благодаря вызову `AddTransient<ITimeService, ShortTimeService>()` система на место объектов интерфейса `ITimeService` будет передавать экземпляры класса `ShortTimeService`.

Кроме того, сервисы добавляются до создания объекта `WebApplication` методом `Build()` объекта `WebApplicationBuilder`:

```

// добавление сервисов
builder.Services.AddTransient<ITimeService,
ShortTimeService>();
// создание объекта WebApplication
var app = builder.Build();

```

После добавления сервиса его можно получить и использовать в любой части приложения. Для получения сервиса могут применяться различные способы в зависимости от ситуации. В данном случае используется свойство `app.Services`, которое предоставляет провайдер сервисов - объект `IServiceProvider`. Для получения сервиса у провайдера сервиса вызывается метод `GetService()`, который типизируется типом сервиса:

```

var timeService = app.Services.GetService<ITimeService>();

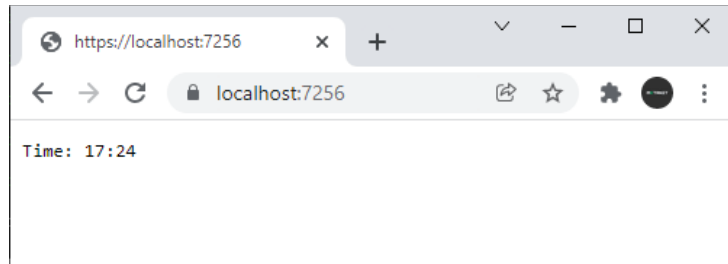
```

После получения сервиса мы можем использовать его.

```

await context.Response.WriteAsync(
    $"Time: {timeService?.GetTime()}");

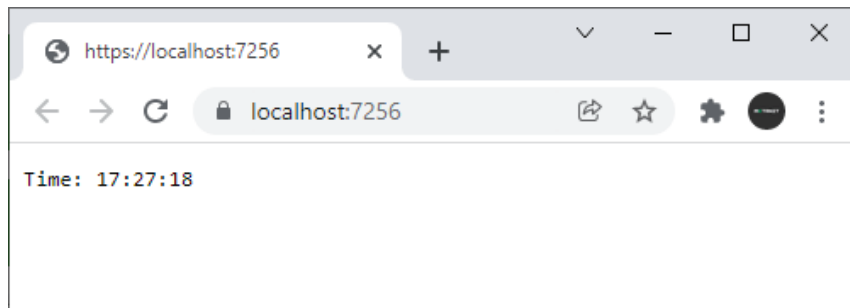
```



Поскольку метод `AddTransient` установил зависимость между `ITimeService` и `ShortTimeService`, то в браузере выводится текущее время в формате "hh:mm". Мы можем поменять тип, сопоставляемый с `ITimeService`:

```
builder.Services.AddTransient<ITimeService, LongTimeService>();
```

И в этом случае мы увидим другое сообщение.



2.6. Сервис как конкретный класс

При этом необязательно разделять определение сервиса в виде интерфейса и его реализацию. Сам термин "сервис" в данном случае может представлять любой объект, функциональность которого может использоваться в приложении.

Например, определим новый класс `TimeService`:

```
public class TimeService
{
    public string GetTime()
        => DateTime.Now.ToShortTimeString();
}
```

Данный класс определяет один метод `GetTime()`, который возвращает текущее время.

Используем этот класс в качестве сервиса:

```
var builder = WebApplication.CreateBuilder();
builder.Services.AddTransient<TimeService>();
var app = builder.Build();
```

```

app.Run(async context =>
{
    var timeService = app.Services.GetService<TimeService>();
    await context.Response.WriteAsync(
        $"Time: {timeService?.GetTime()}");
});

app.Run();

public class TimeService
{
    public string GetTime() =>
        DateTime.Now.ToShortTimeString();
}

```

Для добавления сервиса в эту коллекцию применяется метод `AddTransient()`:

```
builder.Services.AddTransient<TimeService>();
```

После добавления сервиса мы его можем получить и использовать в любой части приложения.

2.7. Расширения для добавления сервисов

Нередко для сервисов создают собственные методы добавления в виде методов расширения для интерфейса `IServiceCollection`. Например, создадим подобный метод для сервиса `TimeService`:

```

public static class ServiceProviderExtensions
{
    public static void AddTimeService(
        this IServiceCollection services)
    {
        services.AddTransient<TimeService>();
    }
}

```

И теперь используем этот метод:

```

var builder = WebApplication.CreateBuilder();

builder.Services.AddTimeService();

var app = builder.Build();
app.Run(async context =>
{
    var timeService = app.Services.GetService<TimeService>();
    context.Response.ContentType =
        "text/html; charset=utf-8";

    await context.Response.WriteAsync(

```

```
        $"Текущее время: {timeService?.GetTime()}");  
    });  
  
    app.Run();  
  
    public class TimeService  
    {  
        public string GetTime() =>  
            DateTime.Now.ToShortTimeString();  
    }  
  
    public static class ServiceProviderExtensions  
    {  
        public static void AddTimeService(  
            this IServiceCollection services)  
        {  
            services.AddTransient<TimeService>();  
        }  
    }  
}
```

3. Методика и порядок выполнения работы

3.1. Учебная задача

Изучите код, представленный в теоретической части лабораторной работы.

3.2. Индивидуальное задание

1. Проанализируйте технологии, рассмотренные в теоретической части лабораторной работы.

2. Модифицируйте приложение, полученное в ходе выполнения предыдущих лабораторных работ. Для этого необходимо внедрить следующие возможности: реализовать инверсию зависимостей, реализовать сервис.

4. Контрольные вопросы

1. Что такое внедрение зависимостей (Dependency Injection, DI) в контексте ASP.NET Core?

2. Какие преимущества предоставляет внедрение зависимостей для разработки приложений?

3. Какие типы внедрения зависимостей существуют в ASP.NET Core?
4. Что такое `ServiceCollection` и как он используется для регистрации сервисов в ASP.NET Core?
5. Какие методы регистрации сервисов существуют в `ServiceCollection`?
6. Что такое `Constructor Injection`, `Method Injection` и `Property Injection`?
7. Какие инструменты и библиотеки используются для внедрения зависимостей в ASP.NET Core?
8. Приведите пример использования внедрения зависимостей в реальном проекте.

ЛАБОРАТОРНАЯ РАБОТА 9. МАРШРУТИЗАЦИЯ

1. Цель и задачи

Цель лабораторной работы: изучить принципы и методы маршрутизации запросов в веб-приложениях.

Задачи лабораторной работы:

1. Ознакомление с основными понятиями и терминами маршрутизации.
2. Изучение различных типов маршрутизации (конфигурационная маршрутизация, атрибутная маршрутизация, ручная маршрутизация).
3. Ознакомление с концепцией IRouteConstraint и его использованием для настройки правил маршрутизации.
4. Рассмотрение примеров использования различных типов маршрутизации в реальных проектах.
5. Изучение методов настройки маршрутов для разных контроллеров и действий.

2. Теоретическое обоснование

2.1. Конечные точки. Метод Map

Система маршрутизации отвечает за сопоставление входящих запросов с маршрутами и на основании результатов сопоставления выбирает для обработки запроса определенную **конечную точку** приложения. **Конечная точка** или **endpoint** представляет некоторый код, который обрабатывает запрос. По сути конечная точка объединяет шаблон маршрута, которому должен соответствовать запрос, и обработчик запроса по этому маршруту.

ASP.NET Core по умолчанию предоставляет простой и удобный функционал для создания конечных точек. Ключевым типом в этом функционале является интерфейс **Microsoft.AspNetCore.Routing.IEndpointRouteBuilder**. Он определяет ряд методов для добавления конечных точек в приложение. И поскольку класс

WebApplication также реализует данный интерфейс, то соответственно все методы интерфейса мы можем вызывать и у объекта **WebApplication**.

Для использования системы маршрутизации в конвейер обработки запроса добавляются два встроенных компонента **middleware**:

- **Microsoft.AspNetCore.Routing.EndpointMiddleware** добавляет в конвейер обработки запроса конечные точки. Добавляется в конвейер с помощью метода **UseEndpoints()**
- **Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware** добавляет в конвейер обработки запроса функциональность сопоставления запросов и маршрутов. Данный **middleware** выбирает конечную точку, которая соответствует запросу и которая затем обрабатывает запрос. Добавляется в конвейер с помощью метода **UseRouting()**

Причем обычно не требуется явным образом подключать эти два компонента **middleware**. Объект **WebApplicationBuilder** автоматически сконфигурирует конвейер таким образом, что эти два **middleware** добавляются при использовании конечных точек.

2.2. Метод **Map**

Самым простым способом определения конечной точки в приложении является метод **Map**, который реализован как метод расширения для типа **IEndpointRouteBuilder**. Он добавляет конечные точки для обработки запросов типа **GET**. Данный метод имеет три версии:

```
public static RouteHandlerBuilder Map (
    this IEndpointRouteBuilder endpoints,
    RoutePattern pattern, Delegate handler);
public static IEndpointConventionBuilder Map (
    this IEndpointRouteBuilder endpoints,
    string pattern,
    RequestDelegate requestDelegate);
public static RouteHandlerBuilder Map (
    this IEndpointRouteBuilder endpoints,
    string pattern,
    Delegate handler);
```

В всех трех реализациях этот метод в качестве параметра **pattern** принимает шаблон маршрута, которому должен соответствовать запрос. Данный параметр может представлять тип **RoutePattern** или **string**.

Последний параметр представляет действие, которое будет обрабатывать запрос. Это может быть делегат типа **RequestDelegate**, либо делегат **Delegate**.

Стоит отметить, что не стоит путать этот метод с одноименным методом **Map()**, который реализован как метод расширения для типа **IApplicationBuilder**

Например, определим следующее приложение:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

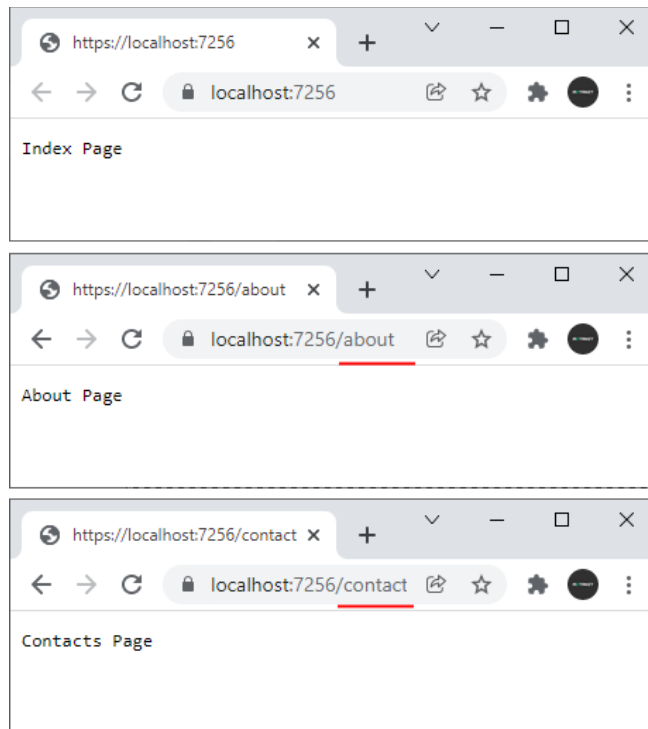
app.Map("/", () => "Index Page");
app.Map("/about", () => "About Page");
app.Map("/contact", () => "Contacts Page");

app.Run();
```

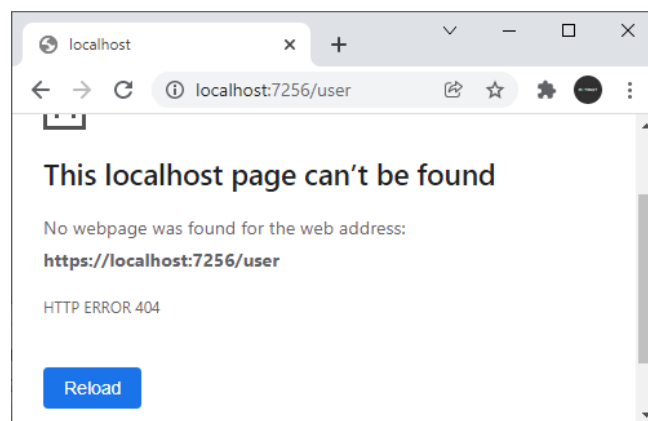
Здесь определено три конечных точки с помощью трех методов **app.Map()**. Первый вызов добавляет конечную точку, которая будет обрабатывать запрос по пути **"/**. В качестве обработчика выступает действие `1 () => "Index Page"`

Результат этого действия - строка **"Index Page"** – это то, что будет отправляться в ответ клиенту.

Аналогично второй и третий вызовы метода **Map** добавляют конечные точки для обработки запросов по путям **"/about"** и **"/contact"**:



Если путь запроса не соответствует ни одной из конечных точек, то приложение отправит ошибку 404:



Выше в примере обработчики маршрутов возвращали строки, но в принципе это может быть любое значение, например:

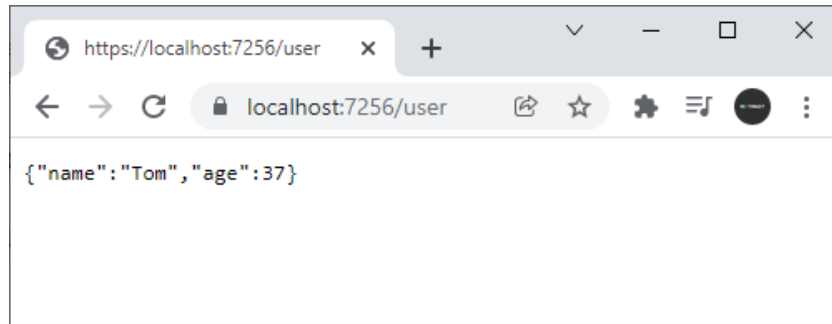
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map("/", () => "Index Page");
app.Map("/user", () => new Person("Tom", 37));

app.Run();

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

Здесь обработчик запроса второй конечной точки возвращает в ответ объект `Person`. По умолчанию подобные объекты при отправке сериализуются в JSON:



В принципе можно ничего из обработчика не возвращать, просто выполнять некоторые действия:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map("/", () => "Index Page");
app.Map("/user", () => Console.WriteLine("Request Path: /user"));

app.Run();
```

В данном случае в обработчике второй конечной точки просто логируем на консоль некоторую информацию.

При необходимости обработчик маршрута можно вынести в полноценный метод:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map("/", IndexHandler);
app.Map("/user", UserHandler);

app.Run();

string IndexHandler()
{
    return "Index Page";
}
Person UserHandler()
{
    return new Person("Tom", 37);
}
record class Person(string Name, int Age);
```

В примерах выше обработчик запроса представлял делегат `Delegate`. Если же необходимо получить полный доступ к контексту `HttpContext`, то

МОЖНО ИСПОЛЬЗОВАТЬ ДРУГУЮ ВЕРСИЮ МЕТОДА, КОТОРАЯ ПРИНИМАЕТ ДЕЛЕГАТ

RequestDelegate:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map("/", () => "Index Page");
app.Map("/about", async (context) =>
{
    await context.Response.WriteAsync("About Page");
});
app.Run();
```

2.3. Получение всех маршрутов приложения

ASP.NET Core позволяет легко получить все имеющиеся в приложении конечные точки. Так, определим следующий код:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map("/", () => "Index Page");
app.Map("/about", () => "About Page");
app.Map("/contact", () => "Contacts Page");

app.MapGet("/routes", (IEnumerable<EndpointDataSource>
endpointSources) =>
    string.Join("\n", endpointSources.SelectMany(
        source => source.Endpoints)));

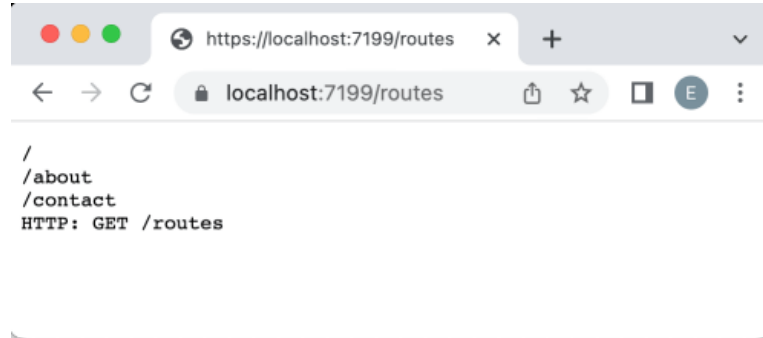
app.Run();
```

Здесь определены четыре конечных точки. Три первых конечных точки стандартные. Поэтому рассмотрим четвертую конечную точку, которая обрабатывает запросы по маршруту "/routes" и которая и будет выводить список всех конечных точек.

Через механизм внедрения зависимостей в обработчик маршрута четвертой конечной точки передается объект **IEnumerable<EndpointDataSource>** - некоторый набор данных о конечных точках. Каждый отдельный элемент этого набора – объект **EndpointDataSource**, который хранит набор конечных точек в свойстве-списке **Endpoints**. Каждая конечная точка в этом списке представляет класс **Endpoint**

С помощью метода `endpointSources.SelectMany()` выбираем из коллекции `Endpoints` все конечные точки. С помощью метода `Join()` они склеиваются в одну строку и разделяются переводом строки `\n`.

В итоге мы увидим в браузере список из четырех конечных точек



При необходимости можно получить более детальную и подробную информацию по каждой конечной точке

```
using System.Text;

var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map("/", () => "Index Page");
app.Map("/about", () => "About Page");
app.Map("/contact", () => "Contacts Page");

app.MapGet("/routes",
    (IEnumerable<EndpointDataSource> endpointSources) =>
    {
        var sb = new StringBuilder();
        var endpoints = endpointSources.SelectMany(es =>
es.Endpoints);
        foreach (var endpoint in endpoints)
        {
            sb.AppendLine(endpoint.DisplayName);

            // получим конечную точку как RouteEndpoint
            if (endpoint is RouteEndpoint routeEndpoint)
            {
                sb.AppendLine(routeEndpoint.RoutePattern.RawText);
            }

            // получение метаданных
            // данные маршрутизации
            // var routeNameMetadata =
            // endpoint.Metadata.Of<Microsoft.AspNetCore.
            // Routing.RouteNameMetadata>().FirstOrDefault();
            // var routeName = routeNameMetadata?.RouteName;
            // данные http - поддерживаемые типы запросов
            // var httpMethodsMetadata =
```

```
        // endpoint.Metadata.GetType<HttpMethodMetadata>().  
        // FirstOrDefault();  
        //var httpMethods = httpMethodsMetadata?.HttpMethods;  
// [GET, POST, ...]  
    }  
    return sb.ToString();  
});  
  
app.Run();
```

3. Методика и порядок выполнения работы

3.1. Учебная задача

Изучите код, представленный в теоретической части лабораторной работы.

3.2. Индивидуальное задание

1. Проанализируйте технологии, рассмотренные в теоретической части лабораторной работы.

2. Модифицируйте приложение, полученное в ходе выполнения предыдущих лабораторных работ. Для этого необходимо внедрить следующие возможности: управление механизмом маршрутизации.

4. Контрольные вопросы

1. Что такое маршрутизация в ASP.NET Core?
2. Какие типы маршрутизации существуют в ASP.NET Core?
3. Что такое IRouteConstraint и для чего оно используется?
4. Как настроить маршруты для разных контроллеров и действий?
5. Какие инструменты и библиотеки используются для маршрутизации в ASP.NET Core?

ЛАБОРАТОРНАЯ РАБОТА 10. ЛОГГИРОВАНИЕ

1. Цель и задачи

Цель лабораторной работы: изучить принципы и инструменты для регистрации событий и ошибок в веб-приложениях.

Задачи лабораторной работы:

1. Ознакомление с основными понятиями и терминами логгирования.
2. Изучение различных типов логгеров (ConsoleLogger, DebugLogger, EventLogLogger, FileLogger, SntpLogger).
3. Анализ ILogger и его использование для регистрации событий.
4. Изучение методов настройки логгеров для разных уровней логгирования (информация, отладка, ошибка, предупреждение).

2. Теоретическое обоснование

2.1. Класс Logger

ASP.NET Core имеет встроенную поддержку логгирования, что позволяет применять логгирование с минимальными вкраплениями кода в функционал приложения.

Для логгирования данных нам необходим объект **ILogger<T>**. По умолчанию среда ASP NET Core через механизм внедрения зависимостей уже предоставляет нам такой объект. Его можно получить как и любую другую зависимость в приложении. Также этот объект можно получить через свойство **Logger** объекта **WebApplication**.

Например, используем встроенный логгер для логгирования на консоль приложения:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Run(async (context) =>
{
    // пишем на консоль информацию
    app.Logger.LogInformation(
        $"Processing request {context.Request.Path}");
});
```

```

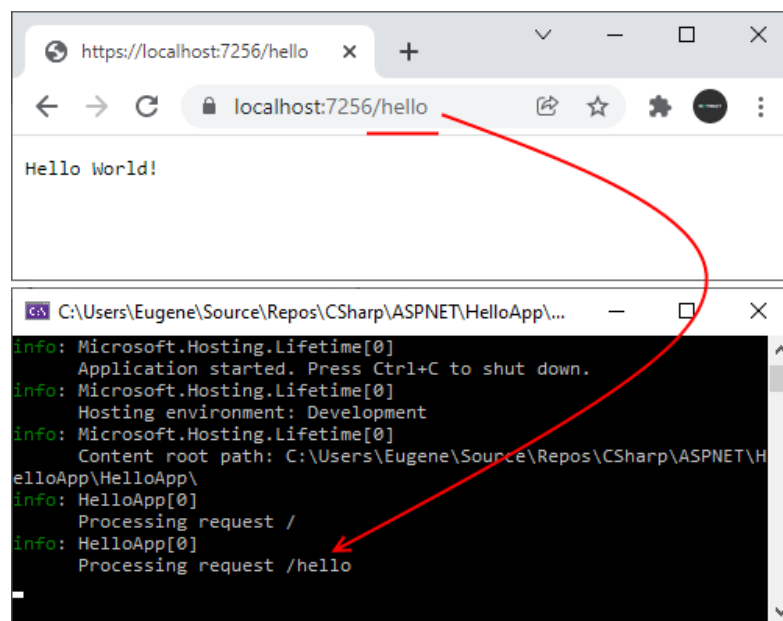
        await context.Response.WriteAsync("Hello World!");
    });

    app.Run();

```

В данном случае через свойство **app.Logger** получаем встроенный логгер и с помощью его метода `logger.LogInformation` передаем на консоль некоторую информацию.

При обращении к приложению с помощью следующего запроса `http://localhost:xxxxx/hello` на консоль будет выведена информация, переданная логгером:



2.2. Категория логгера

При создании логгера для него указывается категория. Обычно в качестве категории логгера выступает класс, в котором используется логгер. В этом случае логгер типизируется классом-категории. Например, логгер, для которого в качестве категории выступает класс `Program`:

```
ILogger<Program>
```

Категория задает текстовую метку, с которой ассоциируется сообщение логгера, и в выводе лога мы ее можем увидеть.

```

C:\Users\Eugene\Source\Repos\CSharp\ASPNET\HelloApp\...
Now listening on: http://localhost:5256
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\Eugene\Source\Repos\CSharp\ASPNET\H
elloApp\HelloApp\
info: HelloApp[0]
      Processing request /
info: HelloApp[0]
      Processing request /hello

```

Где это может быть полезно? Например, у нас есть несколько классов `middleware`, где ведется логгирование. Указывая в качестве категории текущий класс, в последствии в логе мы можем увидеть, в каком классе именно было создано данное сообщение лога. Поэтому, как правило, в качестве категории указывается текущий класс, но в принципе это необязательно.

2.3. Получение логгера через внедрение зависимостей

Поскольку логгер добавляется в сервисы приложения, то мы можем получить его как и любой другой сервис через систему внедрения зависимостей. Например:

```

var builder = WebApplication.CreateBuilder();
var app = builder.Build();

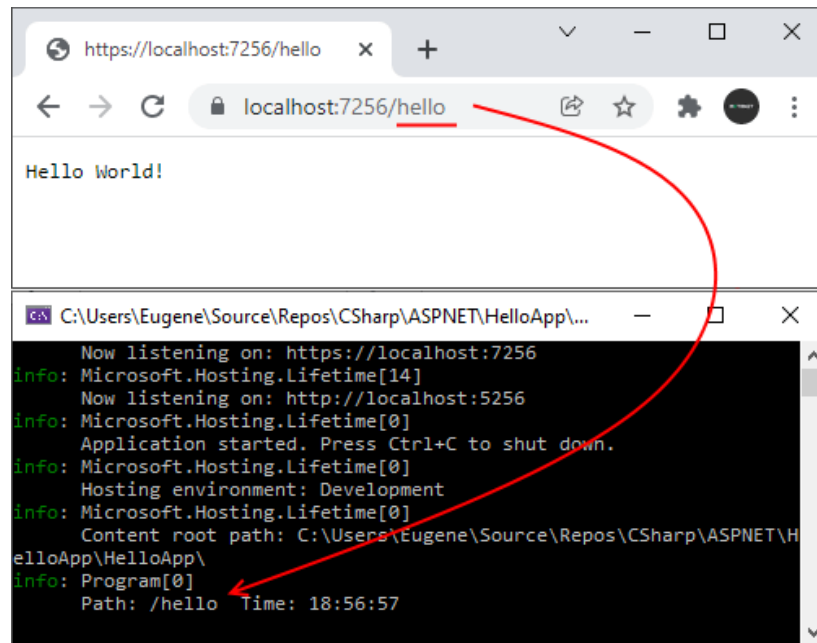
app.Map("/hello", (ILogger<Program> logger) =>
{
    logger.LogInformation(
        $"Path: /hello Time: {DateTime.Now.ToLongTimeString()}");
    return "Hello World";
});

app.Run();

```

В данном случае при обращении по адресу `"/hello"` работает конечная точка, в обработчике которой через механизм внедрения зависимостей можно получить объект логгера. Стоит учитывать, что в этом случае для логгера надо определить категорию. Здесь в качестве категории применяется класс `Program` (неявный класс, в котором и запускается приложение).

В самом обработчике логгер выводит на консоль путь запроса и время запроса:



2.4. Уровни и методы логгирования

При настройке логгирования мы можем установить уровень детализации информации с помощью одного из значений перечисления **LogLevel**. Всего мы можем использовать следующие значения:

- **Trace**: используется для вывода наиболее детализированных сообщений. Подобные сообщения могут нести важную информацию о приложении и его строении, поэтому данный уровень лучше использовать при разработке, но никак не при публикации
- **Debug**: для вывода информации, которая может быть полезной в процессе разработки и отладки приложения
- **Information**: уровень сообщений, позволяющий просто отследить поток выполнения приложения
- **Warning**: используется для вывода сообщений о неожиданных событиях, например, ошибках, которые не останавливают выполнение приложения, но в то же время должны быть исследованы
- **Error**: для вывода информации об ошибках и исключениях, которые возникли при текущей операции и которые не могут быть обработаны

- Critical: уровень критических ошибок, которые требуют немедленной реакции – ошибками операционной системы, потерей данных в БД, переполнение памяти диска и т.д.
- None: вывод информации в лог не применяется

Для вывода соответствующего уровня информации у объекта ILogger определены соответствующие методы расширения:

- LogDebug()
- LogTrace()
- LogInformation()
- LogWarning()
- LogError()
- LogCritical()

Так, в примере выше для вывода информации на консоль использовался метод LogInformation().

Вывод сообщений уровня Trace по умолчанию отключен.

Каждый такой метод имеет несколько перегрузок, которые могут принимать ряд различных параметров:

- string data: строковое сообщение для лога
- int eventId: числовой идентификатор, который связан с логом. Идентификатор должен быть статическим и специфическим для определенной части логируемых событий.
- string format: строковое сообщения для лога, которое может содержать параметры
- object[] args: набор параметров для строкового сообщения
- Exception error: логируемый объект исключения

Также для логгирования определен общий метод **Log()**, который позволяет определить уровень логгера через один из параметров:

```
logger.Log(
    LogLevel.Information,
    $"Requested Path: {context.Request.Path}");
```

При стандартном логгировании на консоль для каждого уровня/метода определен своя метка и цветовой маркер, которые позволяют сразу выделить сообщение соответствующего уровня. Например, при запуске следующего кода:

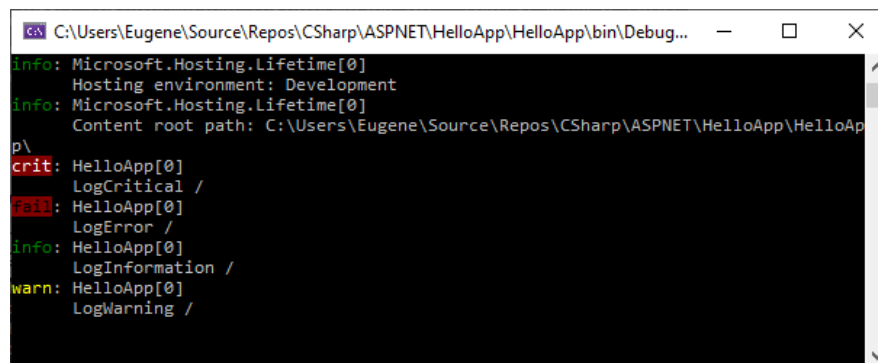
```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Run(async (context) =>
{
    var path = context.Request.Path;
    app.Logger.LogCritical($"LogCritical {path}");
    app.Logger.LogError($"LogError {path}");
    app.Logger.LogInformation($"LogInformation {path}");
    app.Logger.LogWarning($"LogWarning {path}");

    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

мы получим следующий лог на консоль:



```
C:\Users\Eugene\Source\Repos\CSharp\ASPNET\HelloApp\HelloApp\bin\Debug...
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\Eugene\Source\Repos\CSharp\ASPNET\HelloApp\HelloApp\p\
crit: HelloApp[0]
      LogCritical /
err: HelloApp[0]
      LogError /
info: HelloApp[0]
      LogInformation /
warn: HelloApp[0]
      LogWarning /
```

2.5. Фабрика логгера и провайдеры логгирования

В примерах в прошлой теме мы получали объект логгера, который добавляется через DI. Но мы можем также использовать фабрику логгера для его создания:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

ILoggerFactory loggerFactory = LoggerFactory.Create(
    builder => builder.AddConsole());
ILogger logger = loggerFactory.CreateLogger<Program>();
app.Run(async (context) =>
{
    logger.LogInformation($"Requested {context.Request.Path}");
    Path:
```

```

        await context.Response.WriteAsync("Hello World!");
    });

    app.Run();

```

В данном случае с помощью метода **LoggerFactory.Create** создается фабрика логгера в виде объекта **ILoggerFactory**. В качестве параметра метод принимает делегат, который устанавливает некоторые настройки логгирования. В частности, метод **AddConsole()** объекта **ILoggingBuilder** устанавливает вывод сообщений лога на консоль. Затем метод **CreateLogger()** фабрики собственно создает логгер:

```
ILogger logger = loggerFactory.CreateLogger<Program>();
```

Метод **CreateLogger()** типизируется классом, который представляет категорию. В данном случае это класс **Program**, в котором неявно выполняется данный код. Но в качестве альтернативы название категории можно передать в метод в качестве параметра в виде строки:

```
ILogger logger = loggerFactory.CreateLogger(
    "WebApplication");
```

В итоге мы получим тот же вывод сообщений на консоль. Но преимущество использования фабрики логгеров состоит в том, что мы можем дополнительно настроить различные параметры логгирования, в частности, провайдер логгирования.

Получение фабрики логгера через **dependency injection**

Как и логгер, фабрика логгера доступна в приложении в виде сервиса, соответственно ее можно получить через механизм внедрения зависимостей:

```

var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Map("/hello", (ILoggerFactory loggerFactory)=>{

    // создаем логгер с категорией "MapLogger"
    ILogger logger = loggerFactory.CreateLogger("MapLogger");
    // логируем некоторое сообщение
    logger.LogInformation(
        $"Path: /hello   Time: {
            DateTime.Now.ToLongTimeString()}");
    return "Hello World!";
});

app.Run();

```

2.6. Провайдеры логгирования

В примере выше логгирование шло на консоль. Вообще путь логгирования определяется провайдером логгирования. По умолчанию ASP.NET Core предоставляет следующие провайдеры:

- **Console**: вывод информации на консоль. Устанавливается методом **AddConsole()**
- **Debug**: использует для ведения записей лога класс **System.Diagnostics.Debug** и в частности его метод **Debug.WriteLine**. Соответственно все записи лога мы можем увидеть в окне Output в Visual Studio. Устанавливается методом **AddDebug()**. Стоит отметить, что данный способ работает только при запуске проекта в режиме отладки
- **EventSource**: на Windows введет логгирование в лог [ETW](#) (Event Tracing for Windows), для просмотра которого может использоваться инструмент PerfView (или аналогичный инструменты). Хотя данный провайдер задумывался как кроссплатформенный, для Linux и MacOS пока назначение лога не определено. Устанавливается методом **AddEventSourceLogger()**
- **EventLog**: записывает в Windows Event Log, соответственно работает только при запуске на Windows. Устанавливается методом **AddEventLog()**

Например, вместо консоли зададим вывод лога в окне Output в Visual Studio:

```
var builder = WebApplication.CreateBuilder();
var app = builder.Build();

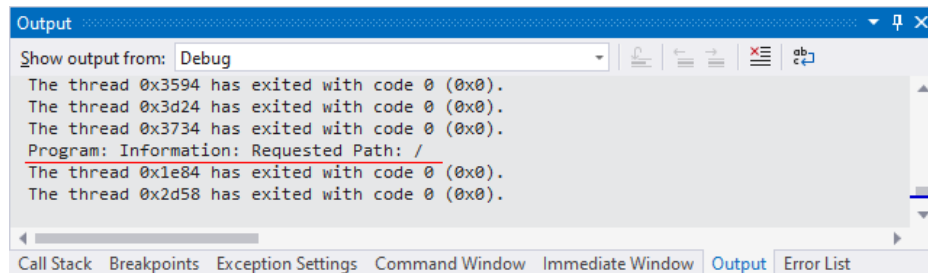
var loggerFactory = LoggerFactory.Create(builder =>
    builder.AddDebug());
ILogger logger = loggerFactory.CreateLogger<Program>();
app.Run(async (context) =>
{
    logger.LogInformation(
        $"Requested Path: {context.Request.Path}");
});
```

```

        await context.Response.WriteAsync("Hello World!");
    });

app.Run();

```



3. Методика и порядок выполнения работы

3.1. Учебная задача

Изучите код, представленный в теоретической части лабораторной работы.

3.2. Индивидуальное задание

1. Проанализируйте технологии, рассмотренные в теоретической части лабораторной работы.

2. Модифицируйте приложение, полученное в ходе выполнения предыдущих лабораторных работ. Для этого необходимо внедрить следующие возможности: реализация механизма логирования для своего приложения.

4. Контрольные вопросы

1. Что такое логирование и зачем оно нужно в ASP.NET Core?
2. Какие типы логгеров доступны в ASP.NET Core?
3. Как использовать ILogger для регистрации событий?
4. Какие методы настройки логгеров существуют в ASP.NET Core?
5. Какие инструменты и библиотеки используются для логирования в ASP.NET Core?
6. В чём разница между Debug и Information уровнями логирования?

7. Как настроить логгирование для разных сред (например, разработки, тестирования и производства)?
8. Что такое Serilog и как его использовать для логирования в ASP.NET Core?
9. Как использовать NLog для логирования в ASP.NET Core?
10. Какие преимущества и недостатки использования разных логгеров в ASP.NET Core?

**ПРИЛОЖЕНИЕ 1. ПРИМЕРНЫЕ ПРЕДМЕТНЫЕ ОБЛАСТИ
ДЛЯ РЕАЛИЗАЦИИ СТУДЕНТАМИ В ЛАБОРАТОРНЫХ РАБОТАХ**

1. Управление персоналом. Учет кадров, сотрудников, отработанного времени.
2. Управление проектами.
3. Управление взаимоотношениями с клиентами (CRM).
4. Управление взаимоотношениями с поставщиками (SRM).
5. Управление цепочками поставок (SCM).
6. Управление запасами.
7. Управление производством.
8. Управление качеством.
9. Управление финансами.
10. Управление рисками.
11. Удаленное управление технологическим оборудованием.
12. Электронная коммерция. Интернет-магазин.
13. Управление взаимоотношениями с клиентами (CRM).
14. Управление качеством.
15. Сервис по генерации обучающих выборок.
16. Сервис для обучения нейронных сетей.
17. Сервис для генерации фейковых данных.

СПИСОК ЛИТЕРАТУРЫ

1. Шкляр, Л. Архитектура веб-приложений : принципы, протоколы, практика / Леон Шкляр, Рич Розен ; [пер. с англ. М. А. Райтмана]. – Москва : Эксмо, 2011. – 640 с. : ил., табл. – (Высший уровень) (Top level). – Загл. и авт. ориг.: Web application architecture / Leon Shklar, Rich Rosen. – Кн. фактически изд. в 2010 г. – ISBN 978-5-699-44273-7

2. Основы Web-технологий [Электронный ресурс]: учебное пособие/ П.Б. Храмов [и др.].— Электрон. текстовые данные.— М.: БИНОМ. Лаборатория знаний, Интернет-Университет Информационных Технологий (ИНТУИТ), 2007.— 374 с.— Режим доступа: <http://www.iprbookshop.ru/22422>.— ЭБС «IPRbooks»3. Боресков, А. Основы работы с технологией CUDA / А.В. Боресков, А.А. Харламов – М.: ДМК Пресс, 2010. – 232 с.

3. Буренин С.Н. Сервис-ориентированные информационные системы и базы данных [Электронный ресурс]: учебный практикум/ Буренин С.Н.— Электрон. текстовые данные.— М.: Московский гуманитарный университет, 2014.— 120 с.— Режим доступа: <http://www.iprbookshop.ru/39683>.— ЭБС «IPRbooks»

4. Фримен, А. ASP.NET MVC 5 с примерами на C# 5.0 для профессионалов. / А. Фримен. Пер. Ю. Артеменко. – Издательство: Вильямс, 2015. – 736 с.

5. Фримен, А. Query 2.0 для профессионалов. / А. Фримен. Пер. А. Гузикевич. – Издательство: Вильямс, 2015. – 1040 с.

Список дополнительной литературы

6 Язык программирования C# 5.0 и платформа .NET 4.5. 6-е изд. Пер. с англ. / Э. Троелсен. – М.:Изд. «Издательский дом Вильямс». 2013. – 1312 с.

7. Албахари Дж. C# 5.0. Справочник. Полное описание языка / Дж. Албахари, Б. Албахари. – М.:Изд. «Издательский дом Вильямс». 2013. – 1008 с.

8. Стиллмен Э. Изучаем С#. 3-е изд. / Э. Стиллмен, Дж. Грин. – СПб.: Питер. 2014. – 816 с.
9. Флентов, М. Библия С#. 2-е изд. / М. Флентов. – СПб.: БХВ-Петербург, 2011. – 560 с.