**Лабораторная работа**

**Обработка HTTP запросов средствами ASP.NET Core. Сохранение состояния. Кэширование.**

**Цель работы.** Ознакомиться c методами обработкой HTTP средствами ASP.NET Core, методами сохранения состояния приложения и повышение производительности приложений путем использования разных видов кэширования.

**1.ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

### Классы 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.

Создание приложения по умолчанию фактически начинается с класса WebApplicationBuilder. Исходный код доступен по адресу [WebApplicationBuilder.cs](https://github.com/dotnet/aspnetcore/blob/main/src/DefaultBuilder/src/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.

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/dotnet/aspnetcore/blob/main/src/DefaultBuilder/src/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
* IEndpointRouteBuilder: 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 заущено, и мы сможем к нему обращаться:

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

При необходимости с помощью метода StopAsync() можно программным сособом 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();  // через 10 секунд завершаем выполнение приложения |

**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.



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: логгирует 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.

**Методы Use, Run и делегат RequestDelegate**

Для конфигурации конвейера обработки запроса применяются методы Run, Map и Use. Для рассмотрения этих методов создадим проект ASP.NET Core по типу Empty.

**Метод Run**

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

|  |
| --- |
| IApplicationBuilder.Run(RequestDelegate handler) |

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

В качестве параметра метод Run принимает делегат RequestDelegate. Этот делегат имеет следующее определение:

|  |  |
| --- | --- |
| 1 | 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 IT"));  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.

После запуска проекта будет запущено приложение откроется браузер, который выполнит запрос к приложению и получит обратно строку "Hello IT".

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 IT")); |

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

|  |
| --- |
| var builder = WebApplication.CreateBuilder();  var app = builder.Build();  app.Run(HandleRequst);  app.Run();    async Task HandleRequst(HttpContext context)  {      await context.Response.WriteAsync("Hello IT 2");  } |

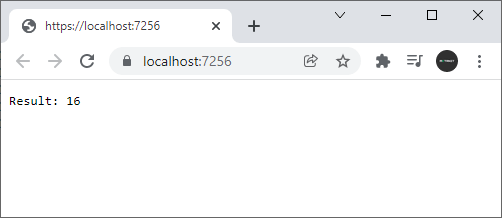
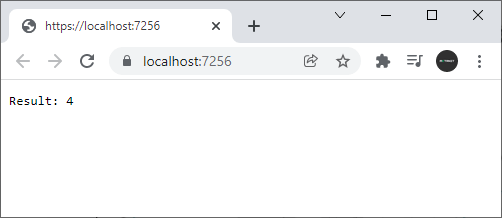
### Жизненный цикл middleware

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 х 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.

**Метод Use**

Метод Use также добавляет компоненты middleware и обрабатывает запрос, но в нем может быть вызван следующий в конвейере запроса компонент middleware. Например:

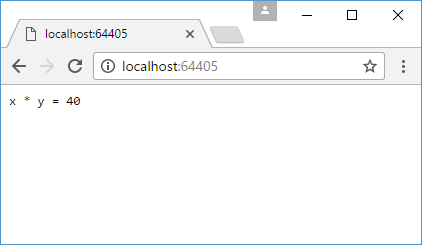
|  |
| --- |
| int x = 5;      int y = 8;      int z = 0;      app.Use(async (context, next) =>      {          z = x \* y;          await next.Invoke();      });        app.Run(async (context) =>      {          await context.Response.WriteAsync($"x \* y = {z}");      }); |

В данном случае мы используем перегрузку метода Use, которая в качестве параметров принимает контекст запроса - объект HttpContext и делегат Func<Task>, который представляет собой ссылку на следующий в конвейере компонент middleware.

Метод app.Use реализует простейшую задачу - умножение двух чисел и затем передает обработку запроса следующим компонентам middleware в конвейере.

То есть при вызове await next.Invoke() обработка запроса перейдет к тому компоненту, который установлен в методе app.Run().

В итоге в веб-браузере мы увидим следующее сообщение:



Если бы мы не использовали вызов await next.Invoke() или закомментировали бы его, то обращения к следующему компоненту в конвейере не произошло бы.

Однако в большинстве случаев мы можем использовать не просто методы Use, а методы расширений app.UseXXX, например, UseStaticFiles() или UseMvc().

При использовании метода Use и передаче выполнения следующему делегату следует учитывать, что не рекомендуется вызывать метод next.Invoke после метода Response.WriteAsync(). Компонент middleware должен либо генерировать ответ с помощью Response.WriteAsync, либо вызывать следующий делегат посредством next.Invoke, но не выполнять оба этих действия одновременно.

То есть к примеру следующая обработка запроса не рекомендуется:

|  |
| --- |
| app.Use(async (context, next) =>      {          await context.Response.WriteAsync("<p>Hello world!</p>");          await next.Invoke();      });        app.Run(async (context) =>      {          // await Task.Delay(10000); можно поставить задержку          await context.Response.WriteAsync("<p>Good bye, World...</p>");      }); |

**Выполнение app.Use**

Если компоненты middleware в app.Use использует вызов next.Invoke() для передачи обработки дальше по конвейеру, то выполнение такого компонента фактически делится на две части: до next.Invoke() и после next.Invoke(). Например:

|  |
| --- |
| int x = 2;      app.Use(async (context, next) =>      {          x = x \* 2;      // 2 \* 2 = 4          await next.Invoke();    // вызов app.Run          x = x \* 2;      // 8 \* 2 = 16          await context.Response.WriteAsync($"Result: {x}");      });        app.Run(async (context) =>      {          x = x \* 2;  //  4 \* 2 = 8          await Task.FromResult(0);      }); |

Здесь определена переменная x, которая равна 2. Последующие вызовы компонентов middleware увеличивают ее значение в два раза. Каким образом будет происходить обработка запроса:

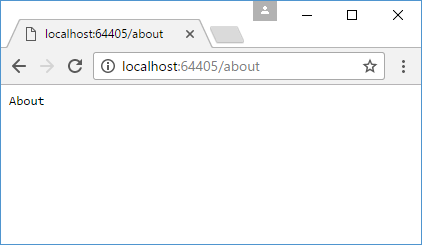
1. Вызов компонента app.Use
2. Увеличение переменной x в два раза: x = x \* 2;. Теперь х равно 4.
3. Вызов await next.Invoke(). Управление переходит следующему компоненту в конвейере - к app.Run.
4. Увеличение переменной x в два раза: x = x \* 2;. Теперь х равно 8.
5. Метод app.Run закончил свою работу, и упрвление обработкой возвращается к app.Use. Начинает выполняться та часть кода, которая идет после await next.Invoke().
6. Увеличение переменной x в два раза: x = x \* 2;. Теперь х равно 16.
7. Отправка ответа клиенту с помощью вызова await context.Response.WriteAsync($"Result: {x}")

### Метод Map

Метод Map (и методы расширения MapXXX()) применяется для сопоставления пути запроса с определeнным делегатом, который будет обрабатывать запрос по этому пути. Например:

|  |  |
| --- | --- |
|  | app.Map("/index", Index);      app.Map("/about", About);        app.Run(async (context) =>      {          await context.Response.WriteAsync("Page Not Found");      });  }    private static void Index(IApplicationBuilder app)  {      app.Run(async context =>      {          await context.Response.WriteAsync("Index");      });  }  private static void About(IApplicationBuilder app)  {      app.Run(async context =>      {          await context.Response.WriteAsync("About");      }); |

Теперь обращения к приложению типа http://localhost:xxxx/about будут обрабатываться с помощью метода About, а запросы типа http://localhost:xxxx/index - методом Index. А все остальные запросы будут обрабатываться делегатом из app.Run().



### Вложенные методы Map

Метод Map может иметь вложенные методы Map, которые обрабатывают подмаршруты. Например:

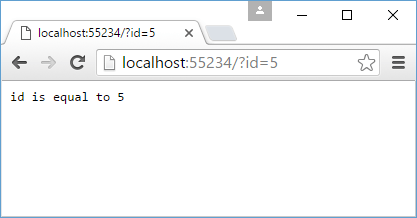
|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | app.Map("/home", home =>      {          home.Map("/index", Index);          home.Map("/about", About);      });        app.Run(async (context) =>      {          await context.Response.WriteAsync("Page Not Found");      });  }    private static void Index(IApplicationBuilder app)  {      app.Run(async context =>      {          await context.Response.WriteAsync("Index");      });  }  private static void About(IApplicationBuilder app)  {      app.Run(async context =>      {          await context.Response.WriteAsync("About");      }); |

Теперь метод About будет обарабатывать запрос не http://localhost:xxxx/about, а http://localhost:xxxx/home/about

### Метод MapWhen

Похожим образом работает метод MapWhen(). Он принимает в качестве параметра делегат Func<HttpContext, bool> и обрабатывает запрос, если функция, передаваемая в качестве параметра возвращает true.

|  |  |
| --- | --- |
|  | app.MapWhen(context => {            return context.Request.Query.ContainsKey("id") &&                  context.Request.Query["id"] == "5";      }, HandleId);        app.Run(async (context) =>      {          await context.Response.WriteAsync("Good bye, World...");      });  }    private static void HandleId(IApplicationBuilder app)  {      app.Run(async context =>      {          await context.Response.WriteAsync("id is equal to 5");      }); |



В данном случае если в запросе указан параметр id и он имеет значение 5, то запрос обрабатывается функцией HandleId(). К подобным запросам будут относиться, например, запрос http://localhost:55234/?id=5 или http://localhost:55234/product?id=5&name=phone, так как обе строки запроса содержат параметр id равный 5. А все остальные запросы также будут обрабатываться делегатом, передаваемым в метод app.Run().

**Состояние приложения. Куки. Сессии.**

<https://metanit.com/sharp/aspnet6/8.2.php>

<https://metanit.com/sharp/aspnet6/8.3.php>

**Кэширование с помощью MemoryCache.**

<https://metanit.com/sharp/aspnet6/17.1.php>

**2. ЗАДАНИЕ**

Используя ранее разработанные объектную модель для доступа к данным в заданной предметной области разработать простое ASP.NET Core приложение.

1. С использование методов Run, Map и Use разработать:
   1. компоненты промежуточного уровня (middleware) и встроить их в конвейер обработки HTTP запроса с целью кэширования 20 записей из каждой таблицы базы данных заданной предметной области с помощью встроенного инструмента кэширования - объекта IMemoryCache. Данные в кэше хранить неизменными в течение 2\*N+240 секунд, где N- номер вашего варианта.
   2. собственную систему маршрутизации входящих запросов:

* если URL адрес входящего запроса содержит \info – выводить в выходной поток для отображения браузером информацию о клиенте и выходить из конвейера обработки запроса;
* если URL адрес входящего запроса содержит \table (где table – имя таблицы из базы данных) – выводить в выходной поток для отображения браузером с использование метода Response.WriteAsync кэшированную информацию из соответствующей таблицы базы данных и выходить из конвейера обработки запроса;
* если URL адрес входящего запроса содержит \searchform1 или \searchform2 – выводить в выходной поток для отображения браузером с использование метода Response.WriteAsync формы для поиска информации из базы данных и выходить из конвейера обработки запроса;

форма должна содержать, как минимум: одно поле, одного поле со списком, один список, одну кнопку;

* в противном случае (URL адрес входящего запроса не содержит перечисленных выше элементов) - продолжать обрабатывать другие компоненты конвейера обработки запросов и передавать управление системе маршрутизации MVC фреймворка;

1. Реализовать сохранение состояния элементов одной формы одной страницы с использованием куки (\searchform1).
2. Реализовать сохранение состояния элементов одной формы одной страницы в виде одного объекта специальной структуры с использованием объекта Session (\searchform2).
3. Осуществить заполнение элементов формы при их загрузке данными ранее сохранненными в объекте Session и куки (\searchform1, \searchform2 ).
4. Разместить выполненный проект в репозитории на GitHub, создать README.md файл.
5. Используя средство GitHub Actions, написать рабочий процесс, который будет осуществлять компиляцию проекта под две разные платформы при любом изменении в репозитории.
6. Отредактировать README.md файл опубликованного проекта, вставив в него код для создания эмблемы состояния рабочего процесса (status badge), показывающей, чем в данный момент завершился рабочий процесс.

**3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ**

1. Начать рекомендуется с создания проекта ASP.NET Core и переноса в него инфраструктуры для доступа к данным базы данных из предыдущей работы.

2. После этого разрабатывается сервис для кэширования данных с использованием MemoryCache (см. <https://metanit.com/sharp/aspnet6/17.1.php>).

Пример кода класса сервиса:

using FuelStation.Data;

using FuelStation.Models;

using Microsoft.Extensions.Caching.Memory;

using System;

using System.Collections.Generic;

using System.Linq;

namespace FuelStation.Services

{

public class CachedTanksService

{

private FuelsContext db;

private IMemoryCache cache;

private int \_rowsNumber;

public CachedTanksService(FuelsContext context, IMemoryCache memoryCache)

{

db = context;

cache = memoryCache;

\_rowsNumber = 20;

}

public IEnumerable<Tank> GetTanks()

{

return db.Tanks.Take(\_rowsNumber).ToList();

}

public void AddTanks(string cacheKey)

{

IEnumerable<Tank> tanks=db.Tanks.Take(\_rowsNumber);

cache.Set(cacheKey, tanks, new MemoryCacheEntryOptions

{

AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5)

});

}

public IEnumerable<Tank> GetTanks(string cacheKey)

{

IEnumerable<Tank> tanks = null;

if (!cache.TryGetValue(cacheKey, out tanks))

{

tanks = db.Tanks.Take(\_rowsNumber).ToList();

if (tanks != null)

{

cache.Set(cacheKey, tanks,

new MemoryCacheEntryOptions().SetAbsoluteExpiration(TimeSpan.FromMinutes(5)));

}

}

return tanks;

}

}

}

3. С использованием методов Run, Map и Use выполняется п. 2.1.2

Пример подключения сервиса:

public void ConfigureServices(IServiceCollection services)

{

// внедрение зависимости для доступа к БД с использованием EF

string connection = Configuration.GetConnectionString("SqlServerConnection");

services.AddDbContext<FuelsContext>(options => options.UseSqlServer(connection));

services.AddDistributedMemoryCache();

services.AddSession();

**// внедрение зависимости CachedTanksService**

**services.AddTransient<CachedTanksService>();**

...

}

и использования сервиса:

app.Run( (context) =>

{

**CachedTanksService cachedTanksService = context.RequestServices.GetService<CachedTanksService>();**

IEnumerable<Tank> tanks = cachedTanksService.GetTanks("Tanks20");

return context.Response.WriteAsync(tanks.FirstOrDefault().TankMaterial.ToString());

});

4. Вывод HTML- таблиц в п. 2.1.2 осуществляется путем циклического вывода в выходной поток с использованием метода Response.WriteAsync каждой отдельной строки таблицы, формируемой динамически c использованием значений соответствующей коллекций DBSet EntityFramework.

app.Run( (context) =>

{

CachedTanksService cachedTanksService = context.RequestServices.GetService<CachedTanksService>();

IEnumerable<Tank> tanks = cachedTanksService.GetTanks("Tanks20");

string HtmlString = "<HTML><HEAD>" +

"<TITLE>Емкости</TITLE></HEAD>" +

"<META http-equiv='Content-Type' content='text/html; charset=utf-8 />'"+

"<BODY><H1>Список емкостей</H1>" +

"<TABLE BORDER=1>";

HtmlString += "<TH>";

HtmlString += "<TD>Код</TD>";

HtmlString += "<TD>Материал</TD>";

HtmlString += "<TD>Тип</TD>";

HtmlString += "<TD>Объем</TD>";

HtmlString += "</TH>";

foreach (var tank in tanks)

{

HtmlString += "<TR>";

HtmlString += "<TD>" + tank.TankID + "</TD>";

HtmlString += "<TD>" + tank.TankMaterial + "</TD>";

HtmlString += "<TD>" + tank.TankType + "</TD>";

HtmlString += "<TD>" + tank.TankVolume + "</TD>";

HtmlString += "</TR>";

}

HtmlString += "</TABLE></BODY></HTML>";

return context.Response.WriteAsync(HtmlString);

});

5. Примером динамической генерации HTML страницы, содержащей формы с параметром, считываемым из GET запроса, с использование метода Response.WriteAsync, может служить следующий фрагмент кода:

app.Run(async (context) =>

{

string firstname = "";

firstname = context.Request.Query["firstname"];

string strresponse = "<html><body><form action = / >" +

"First name:<br><input type = 'text' name = 'firstname' value = "+firstname+">" +

"<br>Last name:<br><input type = 'text' name = 'lastname' value = 'Mouse' >" +

"<br><br><input type = 'submit' value = 'Submit' ></form></body></html>";

await context.Response.WriteAsync(strresponse);

});

6. Инструменты разработчика в браузерах Chrome, Firefox вызываются с использованием сочетания клавиш Ctrl+Shift+I. Для мониторинга HTTP запросов нужно выбрать вкладку Сеть (Network).

**Пример**

<https://github.com/Olgasn/FuelStation/tree/RequestProcessing>

**4. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать:

* Ссылку на GitHub репозиторий.
* Исходный код класса Program.cs (Startup.cs) с разработанными компонентами middleware.
* Исходные классы классов (методов) для создания объектов, сохраняемых в куки и Session, и задание значений элементов формы.
* Содержимое yml файла рабочего процесса.

**5. КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Определение и примеры Web серверов.
2. Характеристика веб-сервера Kestrel. Настройка веб-сервера Kestrel.
3. Общая характеристика HTTP протокола.
4. Опишите структуру класса (Startup.cs), который производит настройку конфигурации ASP.NET Core приложения.
5. Дайте характеристику конвейеру обработки HTTP запросов в ASP.NET Core.
6. Объект HttpRequest.
7. Способы создания компонентов для обработки HTTP запросов внутри конвейера их обработки в ASP.NET Core.
8. Дайте характеристику методам Use, Run, Map, используемым для конфигурирования стека обработки http запроса.
9. Средства для взаимодействия ASP.NET Core приложения со средой, в которой оно запущено.
10. Проблема сохранения состояния Web-приложения. Общая характеристика способов ее решения для ASP.NET Core приложений.
11. Характеристика объекта Session. Достоинства и недостатки использования этого объекта для хранения состояния приложения.
12. Как сохранить объект сложной структуры в объекте Session. Как получить этот объект из Session.
13. Характеристика методов работы с куки средствами ASP.NET Core.
14. Как сохранить объект в куки средствами ASP.NET Core. Как получить этот объект из куки.
15. Кэширования данных с помощью встроенного инструмента - объекта IMemoryCache.
16. Настройка кэширования страниц с использованием атрибута метода контроллера ResponseCache.
17. Создание и использования профилей кэширования.
18. Способы кэширование статических файлов в ASP.NET Core.