**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**



**МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

***Факультет Информационных технологий***

***Кафедра Информатики и информационных технологий***

**направление подготовки**

**09.03.02 «Информационные системы и технологии»**

**ЛАБОРАТОРНАЯ РАБОТА №** 17

**Дисциплина: «Backend»**

**Тема:** *Изучение кэширования в веб-приложении на основе ASP.NET Core*

**Выполнил: студент группы: 231-339**

Карапетян Нвер Каренович\_\_\_\_\_\_\_\_\_\_\_

(Фамилия И.О.)



**Дата, подпись:** 04.05.25 \_\_\_\_\_\_\_\_\_\_\_\_

(Дата)(Подпись)

**Проверил:** \_\_*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_*

(Фамилия И.О., степень, звание) (Оценка)

**Дата, подпись** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *\_\_\_\_\_\_\_\_\_\_\_\_\_*

(Дата)(Подпись)

**Москва**

**2025**

Цель:

Ознакомиться с механизмами кеширования в ASP.NET Core для оптимизации производительности веб-приложений.

Задачи:

* Изучить основные типы кэширования доступные в ASP.NET Core: внутренний кэш, кэш памяти, кэш диска, распределенный кэш.
* Реализовать простые примеры использования каждого типа кэширования в веб-приложении.
* Протестировать работу кэша при различных сценариях, включая частые запросы, изменение данных и истечение времени жизни кэша.
* Оценить влияние использования кеширования на производительность и время ответа веб-приложения.

Ход работы

Кэширование позволяет значительно снизить нагрузку на базу данных и улучшить время отклика веб-приложения за счет хранения результатов дорогостоящих операций (например, запросов к БД) в более быстром хранилище. В ASP.NET Core доступны четыре основных механизма кэширования: встроенный in-memory cache (памятный кэш), распределенный кэш (distributed cache), диск-базированный кэш на уровне приложения и кеширование HTTP-ответов (response cache). В ходе этой лабораторной работы были последовательно реализованы и протестированы примеры каждого из этих видов.

Настройка приложения и сервисов

Перед тем как приступить к контроллеру, в файле **Program.cs** были добавлены все необходимые сервисы и настроено логирование. Сначала подключаются службы кэширования:

Листинг 1. Подключение через систему Dependency Injection сервисов кэширования.

builder.Services.AddMemoryCache();   // Встроенный In-Memory кэш

builder.Services.AddDistributedMemoryCache();   // Распределенный кэш

builder.Services.AddResponseCaching();   // Кэширование HTTP-ответов

После этого очищаются стандартные провайдеры логирования и настраивается Serilog, чтобы выводить сообщения в консоль и файлы:

Листинг 2. Настройка сервисов логирования.

builder.Logging.ClearProviders();

builder.Logging.AddConsole();

builder.Logging.AddDebug();

Log.Logger = new LoggerConfiguration()

    .MinimumLevel.Debug()

    .WriteTo.Console()

    .WriteTo.File("Logs/log-.txt", rollingInterval: RollingInterval.Day)

    .WriteTo.File(new Serilog.Formatting.Json.JsonFormatter(), "Logs/structured-.json", rollingInterval: RollingInterval.Day)

    .CreateLogger();

builder.Host.UseSerilog();

И последним этапом настройки приложения является подключение соответствующих middleware для Response Caching и Serilog Request Logging:

Листинг 3. Подключение необходимых middleware.

app.UseSerilogRequestLogging();

app.UseResponseCaching();

Реализация In-Memory Cache

В методе GetProducts() контроллера сначала проверяется, есть ли в IMemoryCache список товаров по ключу "ProductsList". Если данные найдены, в лог записывается сообщение об успешном попадании в кэш, и возвращается сохраненный список. При первом обращении кэш пуст, поэтому список загружается из базы и сохраняется в память на 30 секунд:

Листинг 4. GET-запрос с реализацией кэширования во внутренней памяти.

// GET: api/products

[HttpGet]

public async Task<ActionResult<List<Product>>> GetProducts()

{

    if (\_memoryCache.TryGetValue(ProductsCacheKey, out List<Product> products))

    {

        \_logger.LogWarning($"MemoryCache: найдено в кэше {products.Count} товаров");

    }

    else

    {

        \_logger.LogWarning("MemoryCache: читаем из базы данных...");

        products = await \_context.Products.ToListAsync();

        \_memoryCache.Set(ProductsCacheKey, products, TimeSpan.FromSeconds(30));

        \_logger.LogWarning($"MemoryCache: закэшировано {products.Count} товаров на 30 секунд");

    }

    return Ok(products);

}

Эта схема отлично подходит для тех данных, которые часто запрашиваются и редко меняются в пределах 30 секунд (или любого другого промежутка времени).

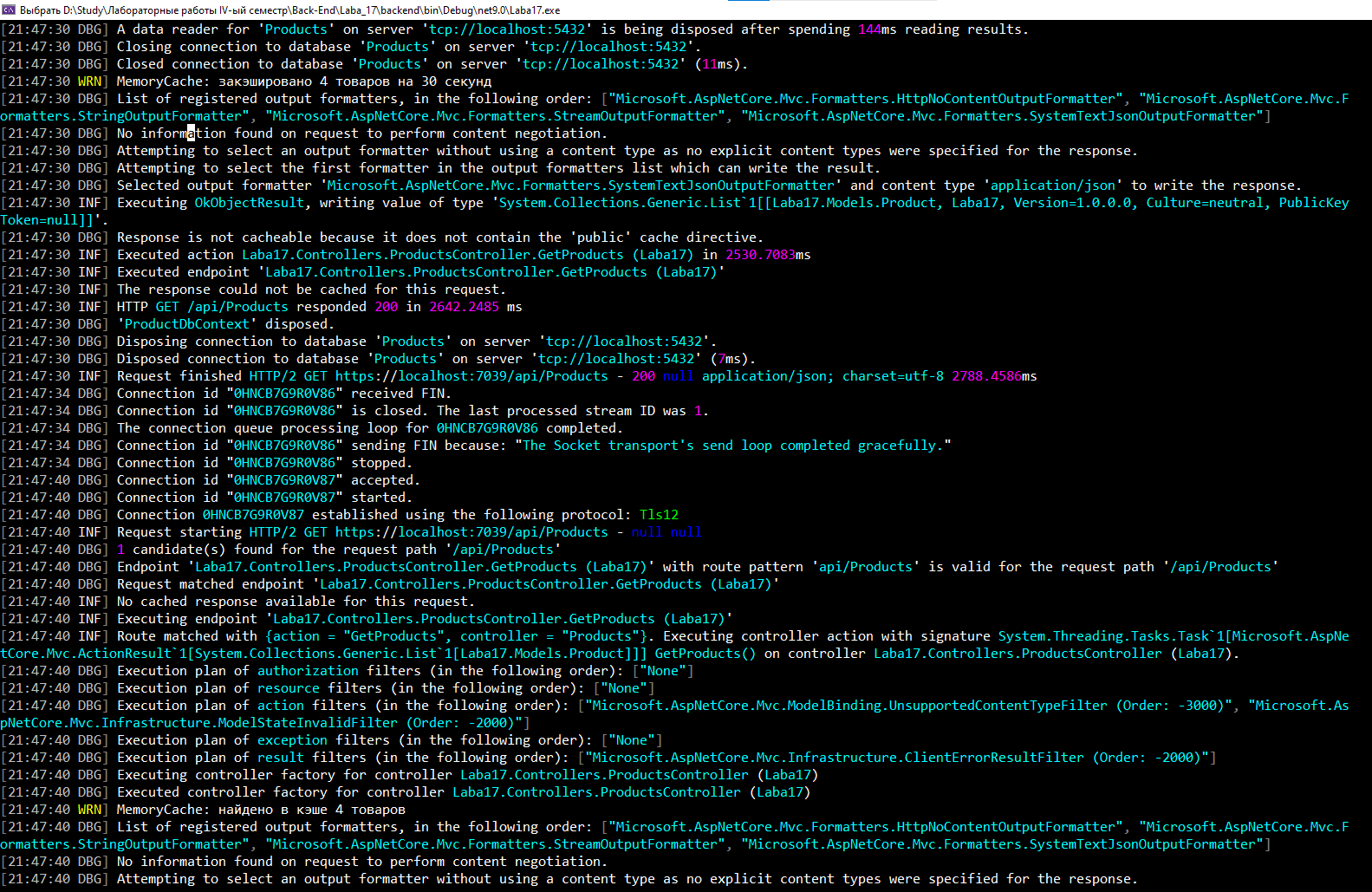


Рисунок . Примеры консольных логов (строки, начинающиеся с желтой метки WRN).

Распределенный кэш (Distributed Cache)

Для метода GetProductById(int id) использован IDistributedCache. Первоначально пытаемся получить сериализованный JSON-объект товара по ключу, формируемому как "Product\_{id}". При отсутствии в кэше считываем товар из базы, сериализуем его в JSON и сохраняем в распределенный кэш на 60 секунд. При повторном запросе данные будут доставаться не из БД, а из кэша:

Листинг 5. GET-запрос с кэшированием данных о товаре в распределенном кэше.

// GET: api/products/{id}

[HttpGet("{id}")]

public async Task<ActionResult<Product>> GetProductById(int id)

{

    string cacheKey = $"Product\_{id}";

    Product product = null;

    var cached = await \_distributedCache.GetStringAsync(cacheKey);

    if (!string.IsNullOrEmpty(cached))

    {

        \_logger.LogWarning($"DistributedCache: найден в распределенном кэше товар с ключом {cacheKey}");

        product = JsonSerializer.Deserialize<Product>(cached)!;

    }

    else

    {

        \_logger.LogWarning($"DistributedCache: в кэше не найден продукт по ключу {cacheKey}. Читаем из базы данных...");

        product = await \_context.Products.FirstOrDefaultAsync(p => p.Id == id);

        if (product is null)

            return NotFound("Продукт не найден");

        var options = new DistributedCacheEntryOptions

        {

            AbsoluteExpirationRelativeToNow = TimeSpan.FromSeconds(60)

        };

        await \_distributedCache.SetStringAsync(cacheKey, JsonSerializer.Serialize(product), options);

        \_logger.LogWarning($"DistributedCache: продукт с ключом {cacheKey} закэширован на 60 секунд");

    }

    return Ok(product);

}

Такой подход позволяет масштабировать приложение в нескольких экземплярах: все они будут обращаться к единому распределенному хранилищу (в примере — в памяти, но в продакшене это может быть Redis).

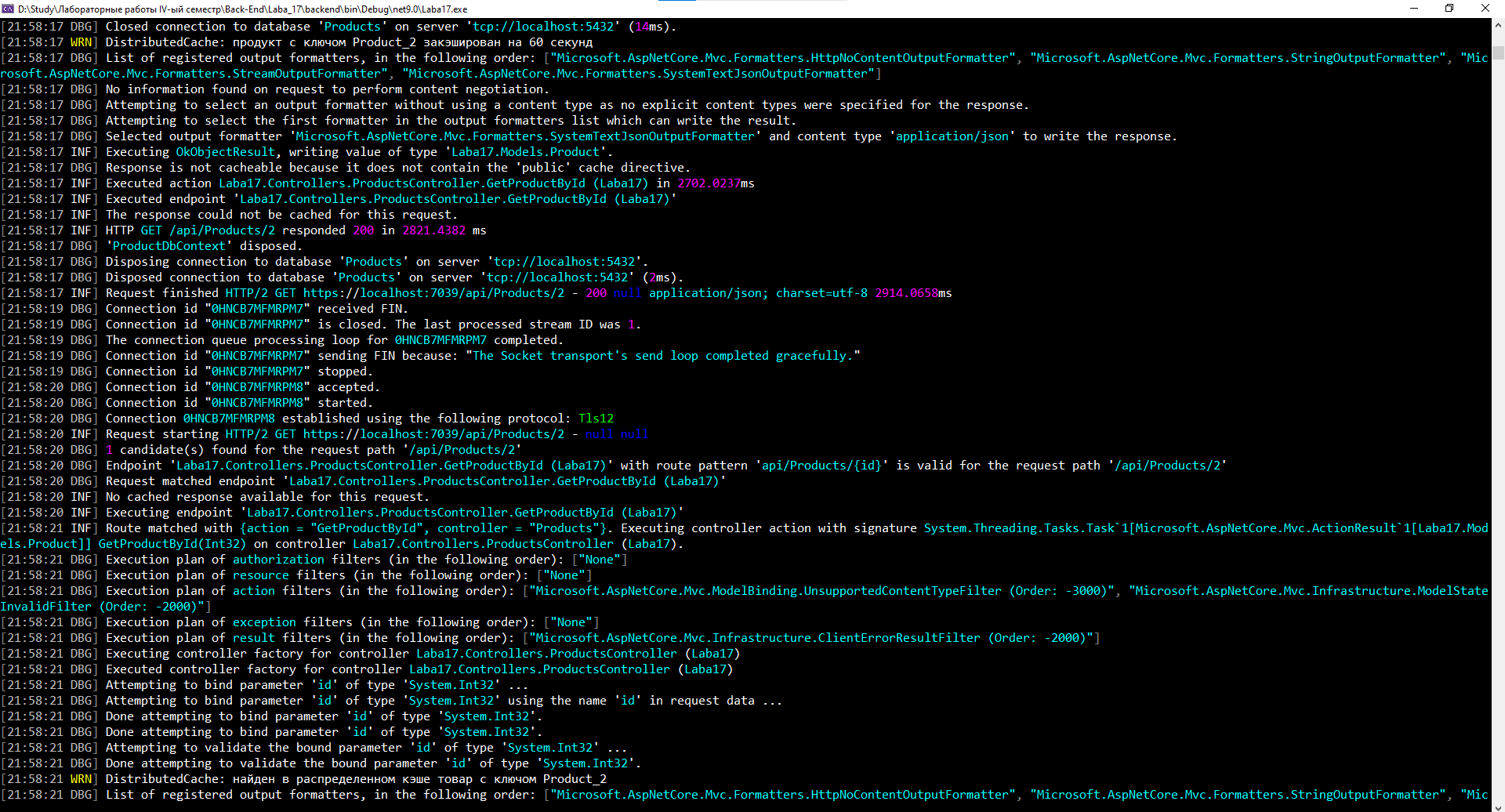


Рисунок . Примеры консольных логов (строки, начинающиеся с желтой метки WRN).

Кэширование на диске

Для демонстрации файлового кэша реализован метод GetProductByIdDiskCache(int id), который хранит результат в файле CacheFiles/product\_{id}.json. Если файл существует и моложе 45 секунд, данные читаются из файла, иначе снова запрашиваются из БД и сохраняются в файл:

Листинг 6. GET-запрос с кэшированием данных о товаре на диске в формате JSON-файлов.

// GET: api/products/disk/{id}

[HttpGet("disk/{id}")]

public async Task<ActionResult<Product>> GetProductByIdDiskCache(int id)

{

    var cacheFile = Path.Combine("CacheFiles", $"product\_{id}.json");

    Product product = null;

    if (System.IO.File.Exists(cacheFile) && DateTime.UtcNow - System.IO.File.GetCreationTimeUtc(cacheFile) < TimeSpan.FromSeconds(45))

    {

        \_logger.LogWarning($"DiskCache: найден файл {cacheFile}");

        var json = await System.IO.File.ReadAllTextAsync(cacheFile);

        product = JsonSerializer.Deserialize<Product>(json);

    }

    else

    {

        \_logger.LogWarning($"DiskCache: в кэш диске не найден продукт (ID = {id}). Читаем из базы данных...");

        product = await \_context.Products.FirstOrDefaultAsync(p => p.Id == id);

        if (product is null)

            return NotFound("Продукт не найден");

        Directory.CreateDirectory("CacheFiles");

        await System.IO.File.WriteAllTextAsync(cacheFile, JsonSerializer.Serialize(product));

        \_logger.LogWarning($"DiskCache: записан файл {cacheFile}");

    }

    return Ok(product);

}

Файловый кэш удобен для данных, которые не изменяются слишком часто и должны сохраняться между перезапусками приложения.

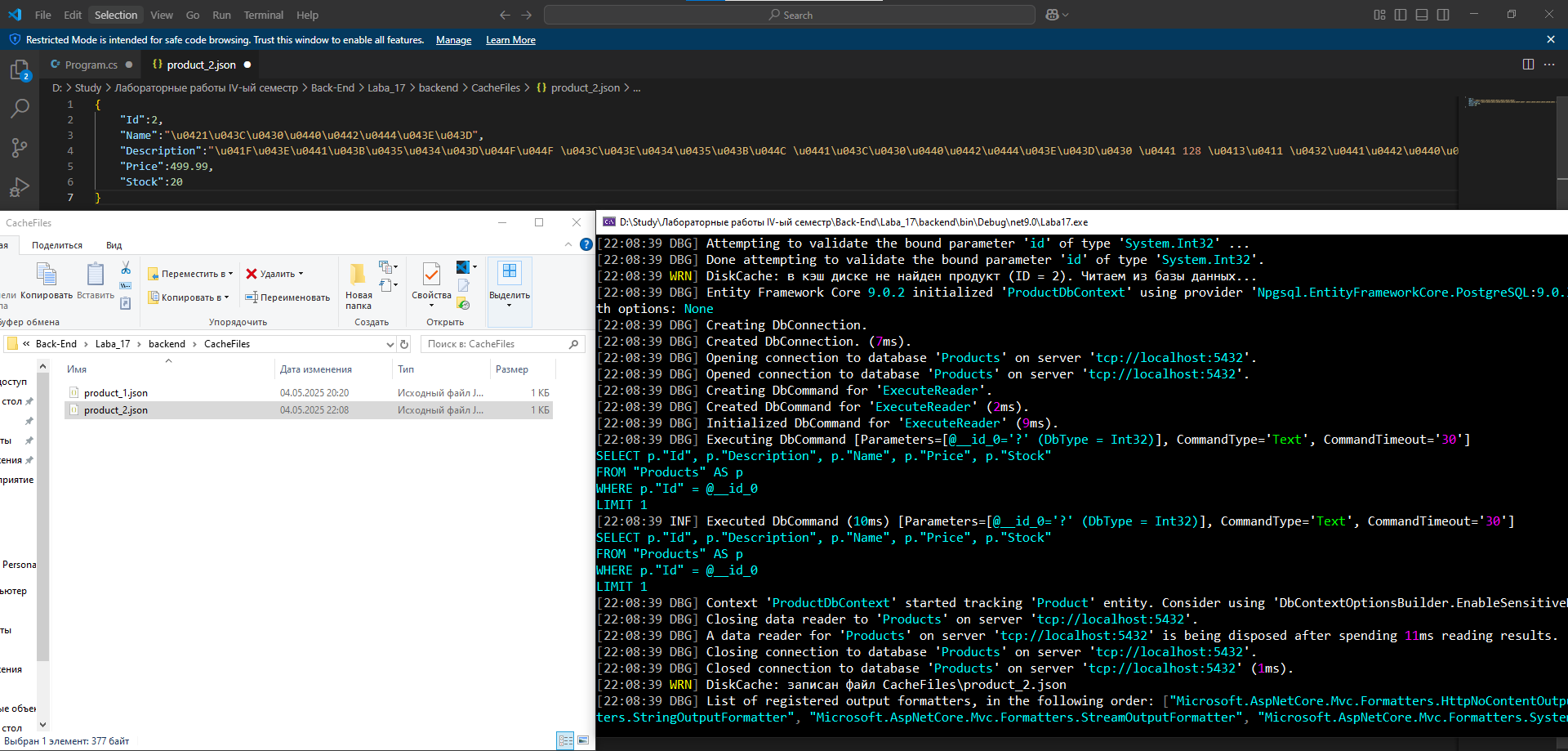


Рисунок . Демонстрация содержимого подобного JSON-файла. Примеры консольных логов.

Кэширование HTTP-ответов (Response Caching)

HTTP-кэширование позволяет браузеру или прокси хранить готовый ответ. В методе GetProductsResponseCache() добавлен атрибут [ResponseCache] с параметром Duration = 20 секунд. NoStore = false означает, что заголовок Cache-Control **не** будет содержать директиву no-store. Если бы NoStore = true, сервер бы добавил Cache-Control: no-store, запрещая любому узлу кэшировать ответ вовсе. Поскольку мы устанавливаем NoStore = false, кэширование разрешено, и в HTTP-заголовках мы увидим, например: Cache-Control: public, max-age=20. При первом запросе ответ формируется и кэшируется, а при повторном в течение 20 секунд сервер не будет выполнять бизнес-логику, а просто выдаст сохраненный ответ:

Листинг 7. GET-запрос с кэшированием HTTP-ответа.

// GET: api/products/response

[HttpGet("response")]

[ResponseCache(Duration = 20, Location = ResponseCacheLocation.Any, NoStore = false)]

public async Task<ActionResult<List<Product>>> GetProductsResponseCache()

{

    \_logger.LogWarning($"ResponseCache: формируется свежий ответ в {DateTime.UtcNow}");

    var products = await \_context.Products.ToListAsync();

    return Ok(products);

}

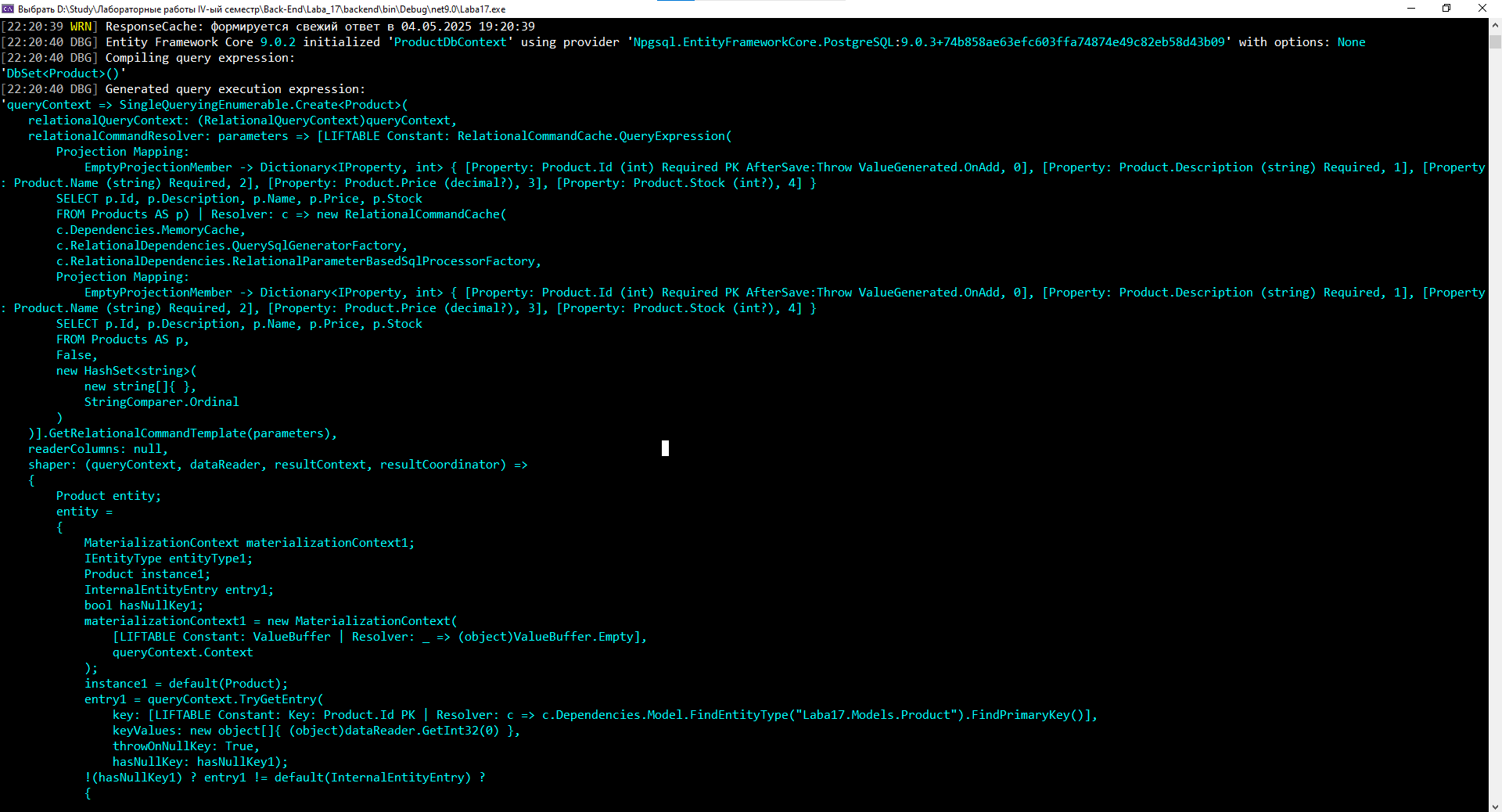


Рисунок . Примеры консольных логов.