# Стратегический отчет: Архитектура Middleware и Паттерны Внедрения Зависимостей в ASP.NET Core (Неделя 6, День 2)

## 1. Введение: Смена парадигмы от Игрового Цикла к Конвейеру Запросов

Переход от разработки на Unity к созданию корпоративных бэкенд-систем на базе ASP.NET Core представляет собой фундаментальный сдвиг в ментальной модели инженера. Этот переход выходит далеко за рамки синтаксических различий C# или смены IDE. Он требует полной деконструкции привычных паттернов управления состоянием, жизненным циклом объектов и потоком выполнения. В экосистеме Unity разработчик оперирует в рамках детерминированного **Игрового Цикла (Game Loop)**, где время дискретно и разбито на кадры, а объекты (GameObjects) часто живут от начала сцены до её завершения, сохраняя свое состояние в полях классов между вызовами Update().

Второй день шестой недели обучения, согласно утвержденному плану 1, посвящен одной из самых критических и сложных тем для понимания Unity-разработчиком: **архитектуре Middleware (посредников) и интеграции с контейнером Внедрения Зависимостей (Dependency Injection)**. В отличие от Unity, где состояние часто глобально и изменяемо (например, через статические менеджеры или Singleton-паттерны), веб-сервер ASP.NET Core оперирует в стохастической среде, обрабатывая тысячи конкурентных HTTP-запросов. Каждый запрос — это атомарное событие, рождающееся и умирающее за миллисекунды. В этой среде попытка применить привычные паттерны сохранения состояния в полях класса (например, сохранение HttpContext в приватную переменную) приводит к катастрофическим последствиям: состоянию гонки (Race Conditions), утечкам памяти и утечкам данных между пользователями.2

Данный отчет представляет собой исчерпывающее теоретическое руководство для второго дня обучения. Он призван не просто объяснить *как* написать Middleware, но и *почему* архитектура устроена именно так, какие глубинные механизмы CLR (Common Language Runtime) задействованы при активации компонентов, и как избежать архитектурных ловушек, специфичных для разработчиков с бэкграундом в GameDev. Мы проанализируем внутреннее устройство IApplicationBuilder, механизмы ActivatorUtilities, различия в аллокации памяти между Convention-based и Factory-based подходами, а также влияние этих паттернов на производительность и совместимость с Native AOT в.NET 8.4

### 1.1. Онтологическое различие сред выполнения

Для глубокого понимания материала необходимо провести четкую границу между средой выполнения Unity и средой ASP.NET Core.

| **Характеристика** | **Unity (Game Loop)** | **ASP.NET Core (Request Pipeline)** |
| --- | --- | --- |
| **Единица исполнения** | Кадр (Frame). Синхронное обновление всех объектов сцены каждые ~16 мс. | Запрос (Request). Асинхронная обработка события прихода данных в сокет. |
| **Потоковая модель** | Однопоточная (Main Thread) для API движка. Корутины для псевдо-асинхронности. | Многопоточная (ThreadPool). Тысячи параллельных потоков обслуживают разные запросы. |
| **Жизненный цикл** | Длительный (Stateful). Объекты живут, пока загружена сцена. | Эфемерный (Stateless). Объекты (контроллеры, scoped-сервисы) живут миллисекунды. |
| **Управление зависимостями** | Service Locator (GetComponent, FindObjectOfType), Ручная привязка в Инспекторе. | Dependency Injection (DI). Инверсия управления, где контейнер создает и уничтожает объекты. |
| **Риски состояния** | Глобальное состояние удобно и часто безопасно (в рамках Main Thread). | Глобальное состояние (Singleton) требует синхронизации и опасно (Race Conditions). |

Анализ таблицы показывает, что интуиция Unity-разработчика, выработанная годами ("сохрани ссылку в поле при старте и используй в Update"), в контексте ASP.NET Core превращается в анти-паттерн. Middleware в ASP.NET Core — это не аналог MonoBehaviour, который висит на объекте и обновляется. Это скорее функциональный фильтр, через который протекает поток данных. Понимание этого различия является ключом к успешному освоению материала второго дня.1

## 2. Анатомия Middleware: Два пути активации

В экосистеме ASP.NET Core существует два фундаментально различных подхода к созданию Middleware: **Convention-based (основанный на соглашениях)** и **Factory-based (основанный на фабриках и интерфейсе IMiddleware)**. Выбор между ними — это не вопрос эстетики, а вопрос управления памятью, производительности и архитектурной чистоты.

### 2.1. Convention-based Middleware: Магия рефлексии и Синглтоны

Исторически первый и наиболее распространенный подход. Он опирается на "утиную типизацию" (duck typing): если класс выглядит как Middleware, он работает как Middleware.

#### 2.1.1. Механизм работы и жизненный цикл

При использовании Convention-based подхода класс Middleware не реализует никаких интерфейсов. Вместо этого он должен соблюдать строгие соглашения:

1. Иметь публичный конструктор, принимающий делегат RequestDelegate (обычно именуемый next).
2. Иметь метод Invoke или InvokeAsync, принимающий первым параметром HttpContext и возвращающий Task.

Критически важный аспект, который часто упускают новички: **Convention-based Middleware инстанцируется только один раз за все время жизни приложения**.7

Когда вы вызываете app.UseMiddleware<MyMiddleware>() в Program.cs:

1. Фреймворк использует рефлексию для анализа конструктора класса.
2. С помощью ActivatorUtilities создается экземпляр класса.
3. Этот экземпляр сохраняется в конвейере как **Singleton**.

Это означает, что конструктор вызывается ровно один раз — при старте приложения. Все последующие миллионы запросов будут проходить через *один и тот же* экземпляр класса, вызывая его метод InvokeAsync.

#### 2.1.2. Проблема внедрения зависимостей (Method Injection)

Поскольку класс Middleware является Синглтоном, вы **не можете** внедрять в его конструктор сервисы со временем жизни **Scoped** (например, DbContext или Unit of Work). Если вы попытаетесь это сделать, произойдет одно из двух:

* **Ошибка валидации (в Development):** InvalidOperationException: Cannot consume scoped service '...' from singleton '...'.9
* **Захваченная зависимость (Captive Dependency):** Scoped-сервис будет создан один раз и навсегда привязан к Middleware. Он никогда не будет уничтожен, что приведет к утечке памяти и использованию устаревшего соединения с БД.8

**Решение:** Для Convention-based Middleware используется паттерн **Method Injection**. Зависимости внедряются не в конструктор, а в сигнатуру метода InvokeAsync.

C#

// Пример Convention-based Middleware  
public class LoggingMiddleware  
{  
 private readonly RequestDelegate \_next;  
   
 // В конструктор внедряются только Singleton-сервисы!  
 public LoggingMiddleware(RequestDelegate next)  
 {  
 \_next = next;  
 }  
  
 // Scoped-сервисы внедряются в метод InvokeAsync  
 // Фреймворк резолвит их из HttpContext.RequestServices для каждого запроса  
 public async Task InvokeAsync(HttpContext context, ILogger<LoggingMiddleware> logger)  
 {  
 logger.LogInformation("Request started");  
 await \_next(context);  
 }  
}

#### 2.1.3. Производительность и оптимизации

Внутренняя реализация UseMiddleware в ASP.NET Core выполняет сложную работу по оптимизации. При старте приложения фреймворк не просто находит метод InvokeAsync через рефлексию. Он компилирует дерево выражений (Expression Tree), создавая оптимизированный делегат.11

Это означает, что вызов InvokeAsync во время выполнения происходит практически так же быстро, как прямой вызов метода, без накладных расходов на рефлексию. Более того, так как объект Middleware создается один раз, отсутствует аллокация памяти в куче (Heap) на каждый запрос. Это делает Convention-based подход идеальным для высоконагруженных сценариев (High Frequency Trading, RTB-системы), где важна нулевая аллокация (Zero Allocation).12

### 2.2. Factory-based Middleware: Строгая типизация и IMiddleware

С появлением ASP.NET Core 2.0+ и развитием DI-контейнера был представлен альтернативный подход, решающий проблемы неочевидности Convention-based метода. Этот подход основан на реализации интерфейса IMiddleware и использовании фабрики IMiddlewareFactory.13

#### 2.2.1. Механизм активации

Класс Middleware реализует интерфейс IMiddleware, который обязывает реализовать метод InvokeAsync(HttpContext context, RequestDelegate next).

Ключевое отличие: Экземпляр такого Middleware создается заново для каждого HTTP-запроса (если зарегистрирован как Transient) или один раз на запрос (если Scoped).13

Процесс активации:

1. Вы регистрируете Middleware в DI-контейнере: services.AddTransient<MyFactoryMiddleware>().
2. В Program.cs вызываете app.UseMiddleware<MyFactoryMiddleware>().
3. При каждом запросе конвейер обращается к IMiddlewareFactory.
4. Фабрика запрашивает у провайдера услуг (IServiceProvider) новый экземпляр Middleware.
5. После выполнения запроса фабрика освобождает (Release) экземпляр (вызывается Dispose, если реализован IDisposable).

#### 2.2.2. Преимущества для Unity-разработчика

Этот подход ментально ближе к тому, как Unity-разработчики привыкли работать с префабами (Instantiate). Создается новый объект, выполняет работу и уничтожается.

Главное преимущество: Возможность внедрения Scoped-зависимостей прямо в конструктор. Так как сам Middleware живет один запрос, он может безопасно принимать другие сервисы, живущие один запрос (например, DbContext). Это делает код чище и понятнее, следуя принципу Explicit Dependencies Principle.11

C#

// Пример Factory-based Middleware  
public class AuthenticationMiddleware : IMiddleware  
{  
 private readonly AppDbContext \_dbContext;  
  
 // Scoped-сервис безопасно внедряется в конструктор!  
 public AuthenticationMiddleware(AppDbContext dbContext)  
 {  
 \_dbContext = dbContext;  
 }  
  
 public async Task InvokeAsync(HttpContext context, RequestDelegate next)  
 {  
 var user = await \_dbContext.Users.FindAsync(context.User.Identity.Name);  
 //... логика...  
 await next(context);  
 }  
}

#### 2.2.3. Плата за удобство: Аллокации

Factory-based подход имеет цену. Создание нового экземпляра класса на каждый запрос означает дополнительную нагрузку на Garbage Collector (GC). Хотя.NET GC (Gen 0) работает чрезвычайно быстро, при нагрузках в десятки тысяч запросов в секунду (RPS) это может создать заметное давление на память (GC Pressure).15

В сценариях, где производительность критична, Unity-разработчик должен понимать этот компромисс: удобство архитектуры против сырой производительности.

### 2.3. Сравнительная матрица архитектурных подходов

| **Характеристика** | **Convention-based** | **Factory-based (IMiddleware)** |
| --- | --- | --- |
| **Интерфейс** | Нет (Duck Typing) | IMiddleware |
| **Жизненный цикл** | Singleton (Один на все приложение) | Transient/Scoped (Один на запрос) |
| **Внедрение Scoped-сервисов** | Только в метод InvokeAsync (Method Injection) | В конструктор (Constructor Injection) |
| **Потокобезопасность полей** | **НЕТ**. Поля класса разделяются всеми потоками. | **ДА**. Поля принадлежат только текущему запросу. |
| **Регистрация в DI** | Не требуется (создается через ActivatorUtilities) | **Обязательна** (services.AddTransient<T>()) |
| **Совместимость с Native AOT** | Ограниченная (требует настройки тримминга из-за рефлексии) | **Полная** (строгая типизация) 5 |
| **Аллокация памяти** | Нулевая (Zero-Allocation) на запрос | Аллокация объекта Middleware + замыканий на каждый запрос |

## 3. Внедрение Зависимостей: Ловушки и Жизненные Циклы

Третий день плана 1 фокусируется на DI, но теоретическая база должна быть заложена уже сейчас. Для Unity-разработчика DI-контейнер (IServiceProvider) — это "черный ящик", который заменяет привычный GetComponent. Понимание времени жизни сервисов критически важно для предотвращения утечек памяти.

### 3.1. Иерархия Жизненных Циклов (Lifetimes)

В ASP.NET Core существует три основных времени жизни, каждый из которых имеет прямой аналог или контраналог в мире Unity.18

1. **Transient (Эфемерный)**
   * **Механика:** Контейнер создает *новый* экземпляр каждый раз, когда кто-то его запрашивает.
   * **Аналогия Unity:** Instantiate(prefab). Объект создается, используется и забывается.
   * **Применение:** Легковесные сервисы без состояния, валидаторы, стратегии.
2. **Scoped (В рамках запроса)**
   * **Механика:** Контейнер создает экземпляр один раз для данного HTTP-запроса. Все компоненты (контроллеры, сервисы), участвующие в обработке *этого* запроса, получат один и тот же объект. В конце запроса объект уничтожается (Dispose).
   * **Аналогия Unity:** Отсутствует прямой аналог. Можно сравнить с объектом, существующим только в рамках загрузки сцены (Scene Context), но умирающим сразу после.
   * **Применение:** DbContext (Entity Framework), Unit of Work, сервисы с данными текущего пользователя. Это основной "рабочий лошадь" веб-разработки.
3. **Singleton (Одиночка)**
   * **Механика:** Создается один раз при первом обращении и живет до остановки приложения.
   * **Аналогия Unity:** DontDestroyOnLoad, статические классы, GameManager.
   * **Применение:** Кэширование (IMemoryCache), конфигурация, сервисы-синглтоны.
   * **Риск:** Должен быть абсолютно потокобезопасным (Thread-Safe), так как к нему обращаются тысячи потоков одновременно.

### 3.2. Анти-паттерн: Захваченные Зависимости (Captive Dependencies)

Это наиболее коварная ошибка, которую совершают 90% разработчиков, переходящих на.NET Core.

Сценарий катастрофы:

Представьте, что у вас есть Singleton-сервис GameServerStatus, который должен логировать активность пользователей в базу данных. База данных доступна через AppDbContext, который является Scoped-сервисом (так как соединение с БД не должно висеть вечно).

Если разработчик внедрит AppDbContext в конструктор GameServerStatus:

1. При старте приложения создается GameServerStatus (Singleton).
2. Для его создания DI-контейнер вынужден создать экземпляр AppDbContext.
3. Этот экземпляр AppDbContext передается в конструктор Синглтона и сохраняется в приватном поле.
4. Синглтон живет вечно. Следовательно, AppDbContext тоже живет вечно.

**Последствия:**

* **Утечка памяти:** DbContext кэширует (Change Tracking) все сущности, которые через него проходят. Поскольку он не уничтожается, кэш растет бесконечно, пока сервер не упадет с OutOfMemoryException (Gen 2 Heap exhaustion).7
* **Состояние гонки:** DbContext не потокобезопасен. Когда два параллельных запроса попытаются записать лог через этот Синглтон, приложение упадет с исключением конкурентного доступа.

#### 3.2.1. Механизм обнаружения (ValidateScopes)

В.NET Core встроен механизм защиты. При запуске в среде Development (и только в ней!) дефолтный билдер контейнера устанавливает флаг ValidateScopes = true.

При попытке запуска приложения, если контейнер обнаружит, что Scoped-сервис внедряется в Singleton, он выбросит исключение при старте:

System.InvalidOperationException: Cannot consume scoped service 'AppDbContext' from singleton 'GameServerStatus'..9

**Важно:** В среде Production (Release build) эта проверка **отключена** ради производительности. Поэтому ошибка может не проявиться на локальной машине (если разработчик случайно отключил валидацию или использует сторонний контейнер), но "убьет" продакшн-сервер.

#### 3.2.2. Паттерн решения: IServiceScopeFactory

Как использовать Scoped-сервис внутри Singleton'а легально? Нужно создавать область видимости (Scope) вручную. Этот паттерн Unity-разработчик должен заучить наизусть.

C#

public class SingletonWorker  
{  
 private readonly IServiceScopeFactory \_scopeFactory;  
  
 // Внедряем не сам сервис, а фабрику областей видимости  
 public SingletonWorker(IServiceScopeFactory scopeFactory)  
 {  
 \_scopeFactory = scopeFactory;  
 }  
  
 public void DoWork()  
 {  
 // Создаем "искусственный" запрос (Scope)  
 using (var scope = \_scopeFactory.CreateScope())  
 {  
 // Теперь мы можем запросить Scoped-сервис у провайдера этого scope  
 var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();  
   
 dbContext.SaveChanges();  
 }   
 // При выходе из using, scope.Dispose() уничтожит dbContext.   
 // Утечки памяти нет.  
 }  
}

Этот код демонстрирует принцип: "Создал — Использовал — Уничтожил". Это эквивалент создания временного объекта в Unity для выполнения задачи и немедленного его уничтожения.21

## 4. Потокобезопасность и HttpContext: Главный парадокс

Для Unity-разработчика работа с состоянием обычно выглядит просто: есть transform.position, который можно менять в Update(). Никто другой не изменит его в середине кадра (в большинстве случаев). В ASP.NET Core такой гарантии нет, и HttpContext является источником постоянных ошибок.

### 4.1. Парадокс HttpContext

HttpContext — это объект, инкапсулирующий все данные об одном конкретном HTTP-запросе (заголовки, тело, пользователь, ответ). Казалось бы, он должен быть доступен везде.

Однако, HttpContext не является потокобезопасным.2

Анти-паттерн "Поле класса":

В Unity часто пишут:

C#

private Player \_player; // Поле класса, доступно всем методам  
void Update() { \_player.Move(); }

Попытка перенести это в Middleware (Convention-based) приводит к катастрофе:

C#

public class UnsafeMiddleware  
{  
 private readonly RequestDelegate \_next;  
 private HttpContext \_currentContext; // КАТАСТРОФИЧЕСКАЯ ОШИБКА  
  
 public UnsafeMiddleware(RequestDelegate next)  
 {  
 \_next = next;  
 }  
  
 public async Task InvokeAsync(HttpContext context)  
 {  
 \_currentContext = context; // Записываем контекст текущего пользователя в поле Синглтона  
   
 await SomeAsyncOperation(); // Поток освобождается, приходит второй запрос...  
   
 // Когда управление вернется, \_currentContext может быть перезаписан другим запросом!  
 // Пользователь А получит данные Пользователя Б.  
 await \_next(\_currentContext);   
 }  
}

В Convention-based Middleware (Синглтон) поля класса являются общими для **всех** одновременных запросов. Запись \_currentContext = context в одном потоке немедленно перезаписывает данные для всех остальных потоков. Это классический **Race Condition**, который в вебе приводит к утечке персональных данных (Cross-talk).24

**Правило:** В Singleton-компонентах (Middleware, Services) никогда нельзя сохранять данные, специфичные для запроса, в поля класса. Данные должны передаваться только через аргументы методов или использоваться внутри локальных переменных метода.

### 4.2. IHttpContextAccessor: Полезный, но опасный инструмент

Иногда доступ к HttpContext нужен глубоко в сервисах, куда неудобно прокидывать его через параметры. Для этого существует IHttpContextAccessor.

Он использует механизм AsyncLocal<T> для хранения данных. Это аналог ThreadLocal, но работающий корректно с асинхронными вызовами (await), когда поток может меняться в процессе выполнения запроса.

Однако, использование IHttpContextAccessor имеет свою цену в производительности и считается "code smell" в чистой архитектуре, так как создает скрытую зависимость от веб-контекста в слоях бизнес-логики. Кроме того, доступ к HttpContext вне цикла обработки запроса (например, в фоновом потоке после завершения запроса) приведет к NullReferenceException или непредсказуемому поведению, так как объект контекста может быть уже утилизирован (Recycled) сервером Kestrel для повторного использования.2

## 5. Middleware в Эпоху.NET 8: Новые Горизонты

В 2024-2025 годах (актуальность.NET 8/9) подходы к Middleware продолжают эволюционировать. Неделя 6 плана обучения включает работу с новейшими возможностями платформы.

### 5.1. Native AOT и Будущее Middleware

С выходом.NET 8 Microsoft активно продвигает Native AOT (Ahead-of-Time compilation) — компиляцию в нативный машинный код без JIT. Это критически важно для облачных сред (AWS Lambda, Azure Functions), так как сокращает время холодного старта и потребление памяти.5

**Влияние на Middleware:**

* **Convention-based Middleware:** Сильно полагается на рефлексию и генерацию кода в рантайме (через ActivatorUtilities). Это делает его проблематичным для AOT (требует сложных настроек тримминга).
* **Factory-based Middleware (IMiddleware):** Полностью совместим с AOT, так как использует явные интерфейсы и типизированные регистрации в DI.
* **Вывод:** Если вы строите микросервис, нацеленный на AOT, предпочтение следует отдавать IMiddleware или новому функциональному стилю регистрации Middleware через делегаты.

### 5.2. IExceptionHandler вместо Middleware

До.NET 8 стандартным способом глобальной обработки ошибок было написание собственного Middleware c try-catch блоком вокруг await next(context).

В.NET 8 представлен интерфейс IExceptionHandler.28

**Преимущества нового подхода:**

* **Разделение ответственности:** Вместо одного гигантского try-catch можно зарегистрировать несколько реализаций IExceptionHandler (например, один для БД ошибок, другой для валидации).
* **Удобство DI:** Обработчики регистрируются как обычные сервисы в DI.
* **Интеграция:** Используется встроенное UseExceptionHandler, которое теперь умеет работать с этими интерфейсами, предоставляя стандартизированный ответ (Problem Details).

### 5.3. Short-Circuit Routing (Короткое замыкание)

Еще одна новинка.NET 8 — возможность "короткого замыкания" маршрутов.

Традиционно, запрос проходит через весь конвейер Middleware (Logging -> Auth -> Compression ->...), даже если это запрос к favicon.ico или /health.

Метод .ShortCircuit() позволяет указать, что для определенных маршрутов не нужно запускать весь тяжелый конвейер, а нужно сразу отдать ответ.1

**Пример:**

C#

app.MapGet("/health", () => Results.Ok("Healthy")).ShortCircuit();

Для Unity-разработчика это аналог оптимизации физики: "Если объект далеко, не считай коллизии". Это существенно экономит CPU на служебных запросах.

## 6. Синтез и Рекомендации к Практике

Завершая теоретический обзор второго дня, можно сформулировать свод правил ("золотой стандарт") для Unity-разработчика, пишущего Middleware:

1. **Презумпция Хаоса:** Всегда пишите код так, как будто его вызовут 1000 раз одновременно. Никакого изменяемого состояния в полях класса (кроме readonly зависимостей).
2. **Закон Жизненного Цикла:**
   * Нужен DbContext в Middleware? -> Используй IMiddleware (Factory-based) или Method Injection в InvokeAsync. Никогда не суй его в конструктор Convention-based Middleware.
   * Нужен DbContext в Singleton-сервисе? -> Используй IServiceScopeFactory.
3. **Выбор Модели:**
   * Для простых задач (логирование, заголовки) -> **Convention-based** (быстрее, меньше памяти).
   * Для сложной логики с зависимостями -> **Factory-based** (безопаснее, чище, AOT-ready).
4. **Управление Исключениями:** Используй IExceptionHandler в.NET 8+ вместо самописных try-catch блоков в Middleware, если это возможно.

Переход от Unity к ASP.NET Core — это переход от управления долгоживущими объектами к управлению потоками данных. Middleware — это трубы этого потока. Если труба "запоминает" воду, которая через нее прошла (сохраняет состояние), она неизбежно лопнет под давлением. Задача второго дня — научиться строить трубы, которые пропускают поток максимально быстро, чисто и безопасно.

## 7. Приложения: Таблицы данных и Сравнения

### Таблица 7.1. Сравнение производительности и аллокаций памяти

12

| **Тип Middleware** | **Startup Cost (Инициализация)** | **Runtime Allocation (GC Gen 0)** | **Throughput Impact (RPS)** | **AOT Readiness** |
| --- | --- | --- | --- | --- |
| **Inline (app.Use)** | Низкая | Средняя (Замыкания/Closures) | Высокая | Средняя |
| **Convention-based** | Высокая (Рефлексия + Компиляция) | **Нулевая** (Zero-Allocation) | **Максимальная** | Низкая (Требует тримминг) |
| **Factory-based (IMiddleware)** | Низкая | Высокая (Новый объект на каждый запрос) | Средняя (-5%...-10%) | **Высокая** |

### Таблица 7.2. Шпаргалка миграции Unity -> ASP.NET Core DI

| **Концепция Unity** | **Концепция ASP.NET Core** | **Опасность / Комментарий** |
| --- | --- | --- |
| FindObjectOfType<T>() | Constructor Injection | Не используйте Service Locator (GetService) в коде. |
| DontDestroyOnLoad | Singleton Lifetime | Осторожно с потокобезопасностью! |
| Instantiate(prefab) | Transient Lifetime | Создается новый при каждой инъекции. |
| (Нет прямого аналога) | Scoped Lifetime | Живет один HTTP-запрос. Самый частый выбор для бизнес-логики. |
| public static var | (Избегать) | Используйте Singleton-сервисы вместо статики для тестируемости. |
| Coroutine | async/await | await не блокирует поток, в отличие от yield return. |

Этот документ закладывает фундамент для выполнения лабораторных работ "Инспектор Конвейера" и "API Key Guard", предусмотренных планом обучения.

#### Источники

1. План обучения ASP.NET Core: Неделя 6
2. ASP.NET Core Best Practices | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/best-practices?view=aspnetcore-10.0>
3. Access HttpContext in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/http-context?view=aspnetcore-10.0>
4. What's New in .NET 8: Breaking Down the Most Powerful Features - DotNet Wisdom, дата последнего обращения: декабря 5, 2025, <https://dotnetwisdom.co.uk/2025/04/02/whats-new-in-net-8-breaking-down-the-most-powerful-features/>
5. What's new in ASP.NET Core in .NET 8 | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/release-notes/aspnetcore-8.0?view=aspnetcore-10.0>
6. Unity game engine adopting .NET Core - devclass, дата последнего обращения: декабря 5, 2025, <https://devclass.com/2022/05/19/unity-game-engine-adopting-net-core/>
7. In Middleware why is this value only set once? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/67913104/in-middleware-why-is-this-value-only-set-once>
8. Write custom ASP.NET Core middleware - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/write?view=aspnetcore-10.0>
9. Dependency injection guidelines - .NET | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-guidelines>
10. Using Scoped Services in Middleware: Pitfalls, Solutions, and Testing in ASP.NET Core, дата последнего обращения: декабря 5, 2025, <https://ercanerdogan.medium.com/using-scoped-services-in-middleware-pitfalls-solutions-and-testing-in-asp-net-core-b79871ea0999>
11. Styles of Writing ASP.NET Core Middleware - Steve Talks Code, дата последнего обращения: декабря 5, 2025, <https://stevetalkscode.co.uk/middleware-styles>
12. Performance benchmark and requests per second comparison between ASP .net core, Java Spring and Python Django : r/dotnet - Reddit, дата последнего обращения: декабря 5, 2025, <https://www.reddit.com/r/dotnet/comments/1f00ii5/performance_benchmark_and_requests_per_second/>
13. Factory-based middleware activation in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/extensibility?view=aspnetcore-10.0>
14. AspNetCore.Docs/aspnetcore/fundamentals/middleware/extensibility.md at main · dotnet/AspNetCore.Docs · GitHub, дата последнего обращения: декабря 5, 2025, <https://github.com/dotnet/AspNetCore.Docs/blob/main/aspnetcore/fundamentals/middleware/extensibility.md>
15. Factory-Based Middleware - by Sina Riyahi - Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@Sina-Riyahi/factory-based-middleware-88314b0eba84>
16. Creating conventional and factory-based Middleware for .NET Core | by Felipe Ramos, дата последнего обращения: декабря 5, 2025, <https://codingdistilled.medium.com/creating-conventional-and-factory-based-middleware-for-net-core-a36751187ca3>
17. Does dependency injection in ASP.NET Core add performance overhead? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/69708519/does-dependency-injection-in-asp-net-core-add-performance-overhead>
18. Understanding Service Lifetimes in ASP.NET Core .NET 8: Transient, Scoped, and Singleton | by Ravi Patel | Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@ravipatel.it/understanding-service-lifetimes-in-asp-net-core-net-8-transient-scoped-and-singleton-fd48752fab4b>
19. Middleware in ASP.NET Core: How to Extend and Optimize Your API - Treblle, дата последнего обращения: декабря 5, 2025, <https://treblle.com/blog/middleware-aspnet-core-extend-optimize-api>
20. Dependency injection validation error in ASP.NET Core projects - DEV Community, дата последнего обращения: декабря 5, 2025, <https://dev.to/alechka/dependency-injection-validation-error-in-aspnet-core-projects-3d7j>
21. ASP.Net Core 2 ServiceProviderOptions.ValidateScopes Property - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/50198609/asp-net-core-2-serviceprovideroptions-validatescopes-property>
22. The ASP.NET Core Dependency Injection System | ABP.IO, дата последнего обращения: декабря 5, 2025, <https://abp.io/community/articles/the-asp.net-core-dependency-injection-system-3vbsdhq8>
23. Avoiding threading issues in ASP.NET Core, дата последнего обращения: декабря 5, 2025, <https://www.poppastring.com/blog/avoiding-threading-issues-in-aspnet-core>
24. HttpContext .NET core saving instance in Middleware - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/38854690/httpcontext-net-core-saving-instance-in-middleware>
25. Locking Middleware or Service to prevent race conditions - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/66606979/locking-middleware-or-service-to-prevent-race-conditions>
26. Memory management and patterns in ASP.NET Core | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/performance/memory?view=aspnetcore-10.0>
27. ASP.NET Core support for Native AOT - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/native-aot?view=aspnetcore-10.0>
28. Handle errors in ASP.NET Core | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/error-handling?view=aspnetcore-10.0>
29. Enhanced exception management utilizing with IExceptionHandler in .NET Core 8 - Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@jaimin_99136/enhanced-exception-management-utilizing-with-iexceptionhandler-in-net-core-8-d041e0c4f96a>
30. Leveraging IExceptionHandler for Effective Error Handling in ASP.NET Core 8.0 - Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@malarsharmila/leveraging-iexceptionhandler-for-effective-error-handling-in-asp-net-core-8-0-5e994ed4e9b4>