# Стратегический отчет: Архитектурная трансформация и учебный план (Неделя 6)

## 1. Введение и архитектурный контекст

Настоящий отчет представляет собой исчерпывающее руководство по реализации шестой недели обучения в рамках программы перехода Unity-разработчиков в сферу Enterprise.NET Backend разработки. Основываясь на базовом документе «Разработка бэкенда: план обучения C#» 1, данный этап является критическим рубежом в ментальной трансформации инженера. Если предыдущие фазы фокусировались на синтаксисе и алгоритмической базе CLR, то шестая неделя посвящена архитектурному скелету любого веб-приложения на платформе.NET — конвейеру обработки запросов (Middleware Pipeline) и системе внедрения зависимостей (Dependency Injection).

Для разработчика, привыкшего к детерминированному игровому циклу (Game Loop) в Unity, переход к асинхронной, событийной модели обработки HTTP-запросов требует фундаментального сдвига парадигмы. В Unity управление состоянием часто осуществляется через синглтоны или прямые ссылки между компонентами (GetComponent), живущими в памяти на протяжении всей сцены. В ASP.NET Core мы оперируем объектами, время жизни которых зачастую ограничено миллисекундами одного запроса, а управление их созданием и уничтожением делегировано IoC-контейнеру.

Цель данного отчета — не просто предоставить список тем, а развернуть детальную техническую доктрину, объясняющую *почему* архитектура устроена именно так, какие подводные камни ожидают Unity-разработчика (например, захваченные зависимости) и как использовать новейшие возможности.NET 8 (Keyed Services, Short-Circuit Routing) для построения высокопроизводительных систем.

### 1.1. Сравнительный анализ сред выполнения: Unity vs ASP.NET Core

Понимание различий в среде выполнения является ключом к успешному освоению материала шестой недели. В таблице ниже приведен детальный анализ архитектурных расхождений, диктующих необходимость изучения Middleware и DI.

| **Характеристика** | **Unity (Game Dev)** | **ASP.NET Core (Enterprise Backend)** | **Архитектурное следствие для Недели 6** |
| --- | --- | --- | --- |
| **Точка входа** | События жизненного цикла (Awake, Start, Update), вызываемые движком 60+ раз в секунду. | Единый метод Main -> Host.Run, слушающий сокет и запускающий Pipeline для каждого запроса. | Необходимость понимания Middleware как аналога "покадровой" обработки, но для каждого запроса индивидуально.1 |
| **Управление зависимостями** | Service Locator (GetComponent, FindObjectOfType), Singleton (Instance), ручная привязка в инспекторе. | Constructor Injection (DI), инверсия управления, строгое время жизни (Lifetime). | Отказ от статического доступа. Изучение IServiceProvider как фабрики объектов.2 |
| **Жизненный цикл объектов** | Определяется сценой или вызовом Destroy(). Объекты часто живут долго (Stateful). | Определяется DI-контейнером (Transient, Scoped, Singleton). Объекты часто живут один запрос (Stateless). | Риск утечек памяти при неправильном выборе Lifetime (Captive Dependency).3 |
| **Многопоточность** | Однопоточный API движка (Main Thread). Coroutines для асинхронности. | Истинная многопоточность (ThreadPool). Тысячи параллельных запросов. | Критичность потокобезопасности Синглтонов. HttpContext не потокобезопасен.4 |

## 2. Детальный учебный план: Неделя 6 (20 часов)

Программа разбита на 5 дней по 4 часа интенсивной работы. Каждый день включает теоретический блок с глубоким погружением во внутреннее устройство платформы, протоколы взаимодействия с AI-ментором (Gemini/ChatGPT) для парного программирования и практические лабораторные работы.

### День 1: Анатомия Конвейера (Middleware Mechanics)

**Цель:** Понять модель выполнения «Russion Doll» (Матрешка), делегаты запросов и отличие Use от Run.

#### 2.1. Теоретический базис: От Loop к Pipeline

[ASP.NET Core Конвейер: Теория и Практика день 1](https://docs.google.com/document/d/1DGNc2HIReKQJ9ReN-7QwAjA2Zmg40NORsC3PfxY5QXg/edit?usp=drive_link)

В основе ASP.NET Core лежит концепция конвейера (Pipeline). В отличие от Unity, где логика разбросана по сотням скриптов MonoBehaviour, выполняющихся в произвольном (или настроенном) порядке внутри кадра, в ASP.NET Core вся логика обработки запроса выстроена в строгую последовательность компонентов Middleware.5

Каждый Middleware представляет собой компонент, который получает HttpContext и ссылку на следующий компонент в цепи (RequestDelegate next). Это реализация паттерна «Цепочка обязанностей» (Chain of Responsibility).

**Ключевые механизмы:**

1. **Двунаправленный поток:** Запрос проходит через middleware *дважды*. Сначала на пути "вниз" (от клиента к серверу), затем, после выполнения await next(), на пути "вверх" (от сервера к клиенту). Это позволяет одному и тому же компоненту замерить время выполнения всего запроса или перехватить исключение, возникшее на любом этапе.5
   * *Аналогия с Unity:* Это похоже на вложенность вызовов методов, где Update() родительского объекта вызывает Update() дочернего, и может выполнить код до и после этого вызова.
2. **app.Use против app.Run:**
   * app.Use подразумевает передачу управления дальше. Он принимает делегат (context, next).
   * app.Run является терминальным middleware. Он не принимает next и завершает конвейер. Все, что зарегистрировано после app.Run, никогда не выполнится.7 Это критически важно для понимания порядка выполнения.
3. **Short-Circuiting (Короткое замыкание):** Если middleware решит не вызывать next(), цепочка прерывается, и запрос начинает путь обратно. Это используется для валидации, кэширования или, например, в.NET 8, для оптимизации статических маршрутов через .ShortCircuit(), минуя тяжелую логику.8

#### 2.2. Протокол взаимодействия с AI (День 1)

Для закрепления материала используется метод "Сократического диалога" с AI.

* **Роль AI:** Senior.NET Architect.
* **Контекст:** Студент — Unity Developer, изучающий Middleware.
* **Промт для студента:**

"Действуй как мой ментор. Я понимаю метод Update в Unity. Объясни мне app.Use в терминах вложенности вызовов.

* 1. Напиши код, где 3 middleware вызывают друг друга. Добавь логирование 'До' и 'После' вызова next.
  2. Спроси меня, в каком порядке появятся логи в консоли. (Ожидаемый ответ: 1-Before, 2-Before, 3-Before, 3-After, 2-After, 1-After).
  3. Что произойдет, если второе middleware не вызовет await next()? Как это соотносится с return в начале метода Update?"

#### 2.3. Практическое задание: "Инспектор Конвейера"

**Задача:** Создать приложение без контроллеров, используя только Program.cs и лямбда-выражения, для анализа HTTP-запросов.

1. **Setup:** Создать пустой проект dotnet new web.
2. **Logger Middleware:** Написать middleware, которое логирует метод (GET/POST) и путь запроса.
3. **Timer Middleware:** Использовать Stopwatch. Запустить его *до* await next(), остановить *после* и записать время выполнения в заголовки ответа (context.Response.Headers.Add("X-Execution-Time",...)).
4. **Error Handler:** Обернуть вызов next в блок try-catch. В блоке catch перехватить исключение, залогировать его и вернуть клиенту JSON с кодом 500. *Важно:* Это должно быть *первое* middleware в конвейере, чтобы ловить ошибки всех последующих.10
5. **Chaos Middleware:** Написать компонент, который с вероятностью 50% бросает исключение. Поместить его в конец цепи.

### День 2: Продвинутые Middleware и Фабрики

**Цель:** Перейти от inline-кода к переиспользуемым классам, понять разницу между Convention-based и Factory-based middleware, изучить потокобезопасность HttpContext.

#### 2.4. Теоретический базис: Жизненный цикл Middleware

[ASP.NET Core Middleware: Теория и Практика день 2](https://docs.google.com/document/d/1WWz3RbTlQ45KmYyDH7stlz9dZzhbnaovO8gYJldnrfE/edit?usp=drive_link)

В Unity компоненты создаются и уничтожаются динамически. В ASP.NET Core большинство middleware создаются один раз при старте приложения (Singleton). Это создает серьезную ловушку для новичков.

1. **Convention-based Middleware:**
   * Определяется через класс с методом InvokeAsync(HttpContext context).
   * Конструктор принимает RequestDelegate next.
   * **Проблема:** Класс инстанцируется *один раз* (Singleton). Если вы попытаетесь внедрить в конструктор зависимость со временем жизни Scoped (например, DbContext), вы получите ошибку или утечку памяти (Captive Dependency). Scoped-зависимости нужно внедрять в сигнатуру метода InvokeAsync.7
2. **Factory-based Middleware (IMiddleware):**
   * Реализует интерфейс IMiddleware.
   * Инстанцируется *для каждого запроса* (если зарегистрирован как Transient/Scoped) через IMiddlewareFactory.
   * Позволяет безопасно внедрять Scoped-сервисы прямо в конструктор.12
3. **Потокобезопасность HttpContext:**
   * В Unity мы привыкли к глобальному Time.deltaTime. В ASP.NET Core HttpContext уникален для запроса. Его *нельзя* сохранять в поля Singleton-класса, так как при параллельных запросах данные перезапишутся (Race Condition). Это классическая ошибка Unity-разработчиков.4

#### 2.5. Протокол взаимодействия с AI (День 2)

* **Роль AI:** Code Reviewer & Performance Engineer.
* **Промт для студента:**

"Я написал middleware для проверки API ключа.

* 1. Проанализируй мой код (предоставить Convention-based код с инъекцией Scoped сервиса в конструктор). Почему это вызовет ошибку 'Cannot resolve scoped service from root provider'?
  2. Перепиши этот код, используя IMiddleware. Объясни, как фабрика middleware создает новый экземпляр класса на каждый HTTP-запрос, в отличие от стандартного подхода.
  3. Сравни это с Instantiate префаба в Unity. Является ли IMiddleware аналогом создания нового GameObject для обработки каждого столкновения?"

#### 2.6. Практическое задание: "API Key Guard"

**Задача:** Реализовать систему аутентификации через кастомное middleware.

1. Создать интерфейс IKeyValidator (Scoped) и его реализацию, которая проверяет наличие ключа в базе (эмуляция через List<string>).
2. Реализовать ApiKeyMiddleware через интерфейс IMiddleware (Factory-based).
3. В конструкторе получить IKeyValidator.
4. В методе InvokeAsync:
   * Проверить заголовок X-API-KEY.14
   * Если ключа нет или он невалиден — вернуть 401 Unauthorized и *не вызывать* next (Short-circuit).
   * Если валиден — вызвать next.
5. Зарегистрировать middleware в Program.cs. Обратить внимание на необходимость регистрации самого класса middleware в DI контейнере (services.AddTransient<ApiKeyMiddleware>()).15

### День 3: Фундамент Dependency Injection

**Цель:** Изучить IoC-контейнер, сервисные жизненные циклы (Lifetimes) и отличие от Service Locator.

#### 2.7. Теоретический базис: Инверсия управления

[Подробный разбор Dependency Injection в .NET день 3](https://docs.google.com/document/d/15d1ISolMPpNfpLpvccczLa0WLNFnmPEn8wYelxVH6wo/edit?usp=drive_link)

Unity разработчики привыкли "искать" зависимости: GetComponent, FindObjectOfType, Singleton.Instance. Это паттерн Service Locator. В Enterprise-разработке это считается анти-паттерном, так как скрывает зависимости класса и усложняет тестирование.1

В ASP.NET Core используется Constructor Injection. Мы "объявляем" зависимости в конструкторе, и DI-контейнер (IServiceProvider) сам их находит и предоставляет.

Жизненные циклы (Lifetimes) 17:

1. **Transient (AddTransient):**
   * *Механика:* Новый экземпляр при каждом запросе (инъекции).
   * *Аналогия Unity:* Instantiate() каждый раз, когда нужен объект. Легковесный, без состояния.
   * *Применение:* Конвертеры, валидаторы, стратегии.
2. **Scoped (AddScoped):**
   * *Механика:* Один экземпляр на весь HTTP-запрос. Все компоненты в рамках одного запроса получают тот же объект. Уничтожается в конце запроса.
   * *Аналогия Unity:* Объект LevelManager, существующий только пока грузится уровень, и доступный всем скриптам загрузки.
   * *Применение:* Context БД (Entity Framework), Unit of Work, данные сессии пользователя.19
3. **Singleton (AddSingleton):**
   * *Механика:* Один экземпляр на всё время работы приложения. Создается при первом обращении.
   * *Аналогия Unity:* DontDestroyOnLoad или статический класс.
   * *Риски:* Обязан быть потокобезопасным (Thread-safe), так как тысячи запросов могут обращаться к нему одновременно.20

#### 2.8. Протокол взаимодействия с AI (День 3)

* **Роль AI:**.NET Internals Expert.
* **Промт для студента:**

"Помоги мне визуализировать Lifetimes.

* 1. Дай мне код примера с генерацией GUID в конструкторе сервиса.
  2. Я хочу внедрить этот сервис дважды в один контроллер и еще раз в другой сервис, который тоже внедрен в контроллер.
  3. Предскажи вывод GUID для Transient, Scoped и Singleton режимов при двух последовательных HTTP-запросах.
  4. Объясни, почему Singleton GUID не меняется, и как это влияет на потребление памяти (LOH vs SOH) по сравнению с созданием тысяч Transient объектов."

#### 2.9. Практическое задание: "Лаборатория Жизненных Циклов"

**Задача:** Эмпирическим путем доказать поведение контейнера.

1. Создать сервис OperationService с свойством Id (Guid), генерируемым в конструкторе.
2. Зарегистрировать его тремя способами: как ITransientOp, IScopedOp, ISingletonOp.
3. Создать "вложенный" сервис ReportingService, который также принимает эти три интерфейса в конструктор.
4. Создать Контроллер, который принимает все три интерфейса *плюс* ReportingService.
5. В методе контроллера вывести JSON, сравнивающий Id экземпляров, полученных напрямую в контроллер, и Id экземпляров внутри ReportingService.
6. Сделать несколько запросов к API и зафиксировать паттерны изменения ID.18

### День 4: Продвинутый DI и Анти-паттерны

[ASP.NET Core DI: Четвертый день, Шестая неделя](https://docs.google.com/document/d/1nwigyeROjK5Mn52B-kn63q2aaGOae2QUIR8Nsqt-je8/edit?usp=drive_link)

**Цель:** Разобраться с Captive Dependencies, Keyed Services (.NET 8) и ручным управлением Scope.

#### 2.10. Теоретический базис: Ловушки и новые возможности

Captive Dependency (Захваченная зависимость):

Самая частая и опасная ошибка. Если внедрить Scoped-сервис (например, DbContext) в Singleton-сервис, то Scoped-сервис станет пленником Синглтона. Он не будет уничтожен после завершения запроса.

* *Последствия:* DbContext будет накапливать кэшированные сущности бесконечно, пока не кончится память (Memory Leak). Кроме того, при параллельном доступе к Синглтону возникнут исключения конкурентного доступа к БД.3
* *Решение:* Внедрять IServiceScopeFactory в Синглтон. Создавать scope вручную: using (var scope = \_scopeFactory.CreateScope()) {... }.3

Keyed Services (.NET 8):

До.NET 8 реализация паттерна "Стратегия" (выбор реализации в рантайме) требовала "костылей" или сложных фабрик. Unity-разработчики часто используют enum или теги для выбора поведения.

В.NET 8 появилась нативная поддержка: builder.Services.AddKeyedSingleton<IInterface, Implementation>("key"). Инъекция происходит через атрибут ``. Это позволяет элегантно переключать реализации (например, разные провайдеры уведомлений).20

#### 2.11. Протокол взаимодействия с AI (День 4)

* **Роль AI:** Senior Debugger.
* **Промт для студента:**

"У меня есть Singleton сервис BackgroundWorker, который должен сохранять данные в БД.

* 1. Напиши код с ошибкой Captive Dependency (внедрение DbContext в конструктор Singleton).
  2. Объясни, почему встроенная валидация ValidateScopes выбросит исключение при запуске в Development режиме.
  3. Перепиши код, используя IServiceScopeFactory, чтобы корректно создавать Scope для каждой операции сохранения.
  4. Покажи, как использовать Keyed Services из.NET 8, чтобы внутри этого воркера выбирать алгоритм сжатия данных ('zip' или 'gzip') в зависимости от настроек."

#### 2.12. Практическое задание: "Мульти-тенантный Уведомитель"

**Задача:** Реализовать сервис уведомлений с выбором канала доставки и безопасным фоновым воркером.

1. **Keyed Services:**
   * Определить интерфейс INotifier (метод Send).
   * Реализовать SmsNotifier и EmailNotifier.
   * Зарегистрировать их как Keyed Services с ключами "sms" и "email".23
   * Создать контроллер, который принимает параметр ?type=sms и использует IServiceProvider.GetRequiredKeyedService<INotifier>(type) для отправки.
2. **Scope Factory:**
   * Создать фоновый сервис HostedService (Singleton), который каждую минуту пишет лог в "Базу Данных" (эмуляция Scoped репозитория).
   * Реализовать безопасный доступ к репозиторию через создание области видимости CreateScope внутри метода ExecuteAsync.20

### День 5: Синтез — Построение Мини-Фреймворка

**Цель:** Объединить знания Middleware и DI для создания собственной микро-архитектуры. Интеграция Routing.

#### 2.13. Теоретический базис: Routing как Middleware

[Создание мини-фреймворка ASP.NET Core день 5](https://docs.google.com/document/d/1qRUV8F419DXwCIvjnDOJyahvRnxbSIEOMioVKrLvBJA/edit?usp=drive_link)

В Unity навигация часто жестко закодирована. В ASP.NET Core маршрутизация (Routing) — это просто два middleware:

1. UseRouting: Анализирует URL и находит подходящий Endpoint (метаданные), но не запускает его. Он кладет результат поиска в HttpContext.
2. UseEndpoints: Берет найденный Endpoint из контекста и исполняет его делегат.24

Это разделение позволяет вклиниться *между* ними. Например, middleware авторизации может проверить атрибуты [Authorize] на найденном Endpoint *до* того, как он будет запущен.

Short-Circuiting в.NET 8:

Новая фича .ShortCircuit() позволяет указать маршрутам (например, robots.txt или healthcheck), что они должны выполняться немедленно, минуя остальной конвейер (аутентификацию, логирование и т.д.). Это критически важно для высоконагруженных систем.8

#### 2.14. Протокол взаимодействия с AI (День 5)

* **Роль AI:** Framework Architect.
* **Промт для студента:**

"Я хочу понять архитектуру MapGet.

* 1. Объясни, как UseRouting и UseEndpoints взаимодействуют через HttpContext. Где хранятся метаданные выбранного маршрута?
  2. Как работает .ShortCircuit() в.NET 8? На каком этапе конвейера происходит отсечение лишних middleware?
  3. Помоги мне спроектировать структуру для финального задания: мне нужно сделать API без контроллеров, используя только MapPost, DI и кастомные Middleware."

#### 2.15. Практическое задание: "Собственный Web Framework"

**Задача:** Собрать все знания недели в едином приложении.

1. Создать приложение с использованием **Minimal APIs** (без контроллеров).
2. **DI:** Зарегистрировать сервис IWeatherService (Scoped), который возвращает прогноз.
3. **Middleware Pipeline:**
   * GlobalExceptionMiddleware (в начале).
   * PerformanceMetricMiddleware (замер времени).
   * UseRouting (стандартный).
   * **Custom Auth Middleware:** Должен проверять наличие токена, но *только* для маршрутов, помеченных специальным мета-тегом (нужно прочитать context.GetEndpoint()?.Metadata).
   * UseEndpoints (стандартный).
4. **Endpoints:**
   * GET /health: Добавить .ShortCircuit(), чтобы не триггерить логику метрик и авторизации.9
   * GET /weather: Требует авторизации. Возвращает данные из сервиса.
5. **Integration:** Убедиться, что Custom Auth Middleware корректно блокирует /weather без токена, но пропускает /health даже без токена (благодаря ShortCircuit или проверке метаданных).

## 3. Документация Google Drive

В рамках задания необходимо создать следующую структуру документов в Google Drive для студента. Это обеспечит системный подход к накоплению знаний.

### Структура папок и файлов

* 📂 **Week\_06\_Middleware\_and\_DI**
  + 📄 **00\_Learning\_Plan\_Week6.md** (Копия данного отчета).
  + 📂 **01\_Theory**
    - 📄 **Pipeline\_Architecture\_Diagram.pdf** (Визуализация потока Use/Run/Map).
    - 📄 **DI\_Lifetimes\_CheatSheet.pdf** (Сравнительная таблица Transient/Scoped/Singleton с примерами из Unity).
    - 📄 **NET8\_New\_Features.md** (Keyed Services, Short Circuits).
  + 📂 **02\_AI\_Prompts**
    - 📄 **Gemini\_Mentor\_Protocol.txt** (Стартовый промт для бота).
    - 📄 **Code\_Review\_Checklist.md** (Чек-лист для проверки Captive Dependencies).
  + 📂 **03\_Labs**
    - 📂 **Day1\_PipelineInspector**
    - 📂 **Day3\_LifeCycles**
    - 📂 **Day5\_MiniFramework**
  + 📄 **04\_Final\_Assessment\_Brief.docx** (ТЗ для финальных задач).

### Стартовый промт для бота (Gemini/ChatGPT)

Ниже приведен текст промта, который студент должен скопировать в чат с AI для начала недели.

SYSTEM PROMPT:

Ты — Senior.NET Backend Architect и ментор для Unity-разработчика, переходящего на ASP.NET Core (.NET 8).

Твоя задача — объяснять концепции бэкенда, используя аналогии из GameDev (Unity), и помогать с ревью кода.

**Контекст: Неделя 6 — Middleware и Dependency Injection.**

**Правила взаимодействия:**

1. **Аналогии:**
   * Сравнивай Middleware Pipeline с Game Loop, но подчеркивай отсутствие цикла и наличие цепочки делегатов.
   * Сравнивай Transient с Instantiate().
   * Сравнивай Singleton с DontDestroyOnLoad, но всегда предупреждай о Thread Safety (Locking), чего обычно нет в Unity Main Thread.
2. **Архитектурный надзор:**
   * При ревью кода ищи "Captive Dependencies" (Scoped внутри Singleton). Это критическая ошибка.
   * Следи за блокирующими вызовами (.Result). В бэкенде это "убийство" ThreadPool, в отличие от фриза кадра в Unity.
3. **Стиль обучения:**
   * Не давай готовый код сразу. Задавай наводящие вопросы ("Что произойдет с памятью, если этот сервис будет жить вечно?").
   * Используй терминологию.NET 8 (Keyed Services, Short Circuits).

Стартовое сообщение:

Привет! На этой неделе мы ломаем твое представление о жизненном цикле приложения. Забудь про Update(). В бэкенде жизнь — это один быстрый запрос. Готов разобрать, как устроена "Матрешка" Middleware? Напиши "Готов", и я дам первую задачу на визуализацию потока выполнения.

## 4. Финальные практические задачи (Capstone Tasks)

Эти задачи предназначены для выполнения в конце недели (выходные) для консолидации знаний. Они требуют интеграции всех изученных тем.

### Задача А: Rate Limiting Middleware (Алгоритмическая)

Контекст: Ваш API атакуют боты. Нужно ограничить количество запросов с одного IP.

Требования:

1. Написать RateLimitMiddleware.
2. Использовать IMemoryCache (Singleton) для хранения счетчиков запросов по IP.
3. Реализовать алгоритм "Fixed Window" (например, 10 запросов в минуту).
4. Если лимит превышен: вернуть статус 429 Too Many Requests и заголовки Retry-After.
5. **Важно:** Использовать Short-Circuiting (не вызывать next), чтобы не тратить ресурсы сервера на обработку заблокированного запроса.
6. *Unity-аналогия:* Это как кулдаун (Cooldown) на способность, но глобальный для каждого игрока (IP).

### Задача Б: Plugin System через DI (Архитектурная)

Контекст: Приложение должно поддерживать экспорт данных в разные форматы, выбираемые пользователем.

Требования:

1. Создать интерфейс IDataExporter с методом Export(object data).
2. Реализовать JsonExporter, XmlExporter, CsvExporter.
3. Использовать **.NET 8 Keyed Services** для регистрации:
   * builder.Services.AddKeyedTransient<IDataExporter, JsonExporter>("json");
   * и так далее.
4. Создать Endpoint POST /export/{format}.
5. Внутри обработчика использовать инъекцию IKeyedServiceProvider (или атрибут ``) для получения нужной реализации экспортера на основе параметра URL.
6. Обработать случай, когда формат не поддерживается (DI контейнер выбросит исключение — его нужно перехватить и вернуть 400 Bad Request).

### Задача В: "Охота на призраков" (Отладка и Тестирование)

Контекст: В продакшене обнаружена утечка памяти и странное поведение данных (пользователи видят чужие данные).

Требования:

1. Написать код, который намеренно создает **Captive Dependency**: Синглтон-сервис кэширования, который получает Scoped DbContext в конструктор.
2. Написать интеграционный тест (используя WebApplicationFactory), который делает два запроса с разными данными.
3. Убедиться (assert), что второй запрос получает устаревшие или чужие данные из-за того, что DbContext не был пересоздан.
4. Исправить код, внедрив IServiceScopeFactory в Синглтон и создавая область видимости (CreateScope) для каждой операции кэширования.3
5. Убедиться, что тест проходит успешно.

## 5. Анализ и выводы

Переход от Unity к Enterprise Backend на шестой неделе достигает своей кульминации. Мы заменяем интуитивно понятную, но ограниченную модель игрового цикла на мощную, масштабируемую архитектуру конвейера и DI.

1. **Отсутствие состояния (Statelessness):** Главный инсайт недели — сервер "забывает" все о клиенте между запросами. Состояние выносится в БД или Кэш, а Middleware лишь обрабатывает транзитные данные.
2. **Дисциплина памяти:** В Unity Instantiate — дорогая операция, и мы используем пулинг объектов. В.NET Core создание тысяч Transient-объектов (Gen0) является нормой и работает экстремально быстро. Однако ошибки с Lifetime (Gen2/LOH) в Синглтонах наказываются жестче — остановкой всего приложения (GC Pause) под нагрузкой.
3. **Сила Middleware:** Понимание того, что маршрутизация, аутентификация и даже обработка ошибок — это просто "кубики" (делегаты), дает разработчику полный контроль над потоком выполнения, недостижимый в стандартных высокоуровневых фреймворках.

Строгое выполнение данного плана, особенно лабораторных работ по Captive Dependencies и Keyed Services, гарантирует, что к концу недели разработчик будет готов к созданию production-ready архитектуры на.NET 8, понимая цену каждой строки кода в Program.cs.

#### Источники

1. Разработка бэкенда: план обучения C#
2. Understanding IServiceCollection and IServiceProvider in ASP.NET Core: A Complete Guide to Dependency Injection | by parsa panahpoor | Medium, дата последнего обращения: декабря 4, 2025, <https://medium.com/@parsapanahpoor/understanding-iservicecollection-and-iserviceprovider-in-asp-net-f798c4adef70>
3. The ASP.NET Core Dependency Injection System | ABP.IO, дата последнего обращения: декабря 4, 2025, <https://abp.io/community/articles/the-asp.net-core-dependency-injection-system-3vbsdhq8>
4. ASP.NET Core Best Practices | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/best-practices?view=aspnetcore-10.0>
5. Write custom ASP.NET Core middleware - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/write?view=aspnetcore-10.0>
6. ASP.NET Core Middleware Pipeline | Web API (.NET 8) Ep 5 - YouTube, дата последнего обращения: декабря 4, 2025, <https://www.youtube.com/watch?v=SkkMdn2pLC0>
7. ASP.NET Core Middleware | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-10.0>
8. Introduction to Short Circuit Routes in .NET 8 | by sharmila subbiah | Medium, дата последнего обращения: декабря 4, 2025, <https://medium.com/@malarsharmila/introduction-to-short-circuit-routes-in-net-8-3f5781c277bc>
9. Introduction to ShortCircuit and MapShortCircuit in .NET 8 - DEV Community, дата последнего обращения: декабря 4, 2025, <https://dev.to/moh_moh701/introduction-to-shortcircuit-and-mapshortcircuit-in-net-8-12ml>
10. Implementing Global Error Handling via Middleware in .NET | by Shahar Shokrani | Medium, дата последнего обращения: декабря 4, 2025, <https://shaharsho.medium.com/implementing-global-error-handling-via-middleware-in-net-805b101fe1f4>
11. Creating conventional and factory-based Middleware for .NET Core | by Felipe Ramos, дата последнего обращения: декабря 4, 2025, <https://codingdistilled.medium.com/creating-conventional-and-factory-based-middleware-for-net-core-a36751187ca3>
12. Styles of Writing ASP.NET Core Middleware - Steve Talks Code, дата последнего обращения: декабря 4, 2025, <https://stevetalkscode.co.uk/middleware-styles>
13. Factory-based middleware activation in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/extensibility?view=aspnetcore-10.0>
14. Using API Key Authentication To Secure ASP.NET Core Web API - C# Corner, дата последнего обращения: декабря 4, 2025, <https://www.c-sharpcorner.com/article/using-api-key-authentication-to-secure-asp-net-core-web-api/>
15. Why do some middleware need to be registered with `ConfigureServices` while others don't? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/72965835/why-do-some-middleware-need-to-be-registered-with-configureservices-while-othe>
16. Dependency Injection Anti-Patterns Killing Your ASP.NET Core Apps - Arg Software, дата последнего обращения: декабря 4, 2025, <https://arg-software.medium.com/dependency-injection-anti-patterns-killing-your-asp-net-core-apps-502f08d85d95>
17. Mastering Dependency Injection in .NET 8: Best Practices and Proven Patterns for Cleaner Code - DEV Community, дата последнего обращения: декабря 4, 2025, <https://dev.to/leandroveiga/mastering-dependency-injection-in-net-8-best-practices-and-proven-patterns-for-cleaner-code-1feh>
18. Understanding Service Lifetimes in ASP.NET Core .NET 8: Transient, Scoped, and Singleton | by Ravi Patel | Medium, дата последнего обращения: декабря 4, 2025, <https://medium.com/@ravipatel.it/understanding-service-lifetimes-in-asp-net-core-net-8-transient-scoped-and-singleton-fd48752fab4b>
19. Dependency injection in ASP.NET Core | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-10.0>
20. Dependency injection - .NET - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection>
21. Dependency injection guidelines - .NET | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-guidelines>
22. Introducing Keyed Services in .NET 8: A Cleaner Way to Handle Multiple Implementations, дата последнего обращения: декабря 4, 2025, <https://gizemcifguvercin.medium.com/introducing-keyed-services-in-net-8-a-cleaner-way-to-handle-multiple-implementations-4af4bf549cd6>
23. Keyed Services in .NET – Advanced Dependency Injection Techniques - codewithmukesh, дата последнего обращения: декабря 4, 2025, <https://codewithmukesh.com/blog/keyed-services-dotnet-advanced-di/>
24. Routing in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-10.0>
25. Mastering the Highway System of ASP.NET Core: An In-Depth Journey Through Routing and Middleware | by Bhargava Koya - Medium, дата последнего обращения: декабря 4, 2025, <https://medium.com/@bhargavkoya56/mastering-the-highway-system-of-asp-net-core-an-in-depth-journey-through-routing-and-middleware-40d6ccede925>