# Теоретический фундамент внедрения зависимостей в ASP.NET Core: Архитектурный переход для разработчиков Unity (Неделя 6, День 3)

## 1. Введение: Смена парадигмы управления состоянием

Переход от разработки интерактивных приложений реального времени на движке Unity к построению высоконагруженных серверных систем на платформе ASP.NET Core знаменует собой фундаментальный сдвиг в ментальной модели инженера. Этот переход выходит далеко за рамки синтаксических различий или изучения новых библиотек; он требует полной реструктуризации понимания того, как программное обеспечение управляет жизненным циклом объектов, памятью и потоком выполнения данных. В контексте третьего дня шестой недели учебного плана, который посвящен основам внедрения зависимостей (Dependency Injection, DI), мы сталкиваемся с критическим архитектурным рубежом: отказом от императивного контроля над созданием объектов в пользу декларативного управления через контейнеры инверсии управления (Inversion of Control, IoC).1

В среде Unity разработчик привыкает к детерминированному, цикличному миру. Игровой цикл (Game Loop) диктует жесткий ритм выполнения: Awake, Start, Update, LateUpdate. Объекты, как правило, пространственно ориентированы — они существуют в рамках Сцены, имеют координаты и сохраняют свое состояние на протяжении тысяч кадров. В этой парадигме управление зависимостями часто сводится к активному поиску: скрипт "протягивает руку" и берет то, что ему нужно, используя методы вроде GetComponent, FindObjectOfType или обращаясь к глобальным статическим экземплярам через паттерн Singleton (например, GameManager.Instance). Это подход "Service Locator", где потребитель диктует условия поиска.2

ASP.NET Core оперирует в принципиально иной реальности. Здесь нет единого цикла, который тикает 60 раз в секунду. Вместо этого система работает как гигантский асинхронный конвейер, обрабатывающий тысячи независимых HTTP-запросов одновременно. Время жизни объектов здесь измеряется не кадрами или минутами, а микросекундами. Объект может быть создан для обработки одного конкретного запроса от пользователя из Токио, выполнить свою функцию и быть уничтоженным сборщиком мусора еще до того, как завершится создание соседнего объекта для пользователя из Нью-Йорка. В таком хаотичном, многопоточном окружении попытка управлять связями между объектами вручную или через статические ссылки неминуемо ведет к состоянию гонки (Race Conditions), утечкам памяти и невозможности тестирования кода.4

Данный отчет представляет собой исчерпывающее теоретическое руководство по механике внедрения зависимостей в.NET 8, разработанное специально для инженеров с бэкграундом в Unity. Мы детально разберем, как "черный ящик" IoC-контейнера создает объекты, почему GetService считается анти-паттерном в Enterprise-разработке, как Server GC отличается от сборщика мусора в Unity и какие новые возможности, такие как Keyed Services, предлагает современная экосистема.NET.

## 2. От Service Locator к Inversion of Control: Эволюция доступа

### 2.1 Философия "Вытягивания" в GameDev

Чтобы глубоко осознать необходимость Dependency Injection, необходимо сначала деконструировать паттерны, укоренившиеся в практике разработки на Unity. Экосистема MonoBehaviour исторически поощряет модель "вытягивания" (pulling) зависимостей. Когда скрипту управления персонажем требуется доступ к системе инвентаря, разработчик пишет код, который активно ищет эту систему. Это может быть локальный поиск через GetComponent<Inventory>(), глобальное сканирование иерархии через FindObjectOfType<Inventory>() или обращение к статическому полю InventoryManager.Instance.6

Этот подход, известный как Service Locator, обладает низкой когнитивной нагрузкой в контексте небольших проектов. Сцена в Unity выступает в роли естественного контейнера состояния: все объекты визуализированы, их иерархия очевидна, и "поиск" зависимости интуитивно понятен. Производительность таких операций, особенно при кэшировании в методе Start, часто не является узким местом благодаря однопоточной природе основного игрового цикла.6 Однако архитектурная цена такого удобства — жесткая связность (Tight Coupling). Класс PlayerController не может существовать в изоляции; он неявно требует наличия InventoryManager в сцене. Попытка протестировать логику движения персонажа в отрыве от системы инвентаря становится нетривиальной задачей, требующей воссоздания сложной сцены и моков глобальных объектов.8

### 2.2 Императив "Вталкивания" в Enterprise

В мире корпоративной разработки на платформе.NET доминирует принцип инверсии зависимостей (Dependency Inversion Principle, буква D в аббревиатуре SOLID). Согласно этому принципу, модули верхнего уровня не должны зависеть от модулей нижнего уровня; оба типа модулей должны зависеть от абстракций.5 Вместо того чтобы класс самостоятельно искал свои зависимости, он декларирует свои потребности через конструктор. Это модель "вталкивания" (pushing).

Когда мы определяем конструктор public OrderService(IDatabase database, ILogger logger), мы заключаем контракт. Мы говорим: "Я не знаю, откуда возьмется база данных или логгер, я не знаю, как они создаются, но я отказываюсь работать без них". Ответственность за поиск, создание и управление жизненным циклом этих зависимостей перекладывается на третью сторону — IoC-контейнер (в терминологии.NET — IServiceProvider).4

Этот сдвиг парадигмы обеспечивает несколько критических преимуществ для бэкенд-систем:

Во-первых, это тестируемость. Поскольку зависимости передаются извне, в unit-тестах мы можем легко подменить реальную базу данных на заглушку (Mock), которая имитирует поведение БД в памяти. В Unity с использованием GetComponent или синглтонов это потребовало бы сложной архитектуры или использования специальных фреймворков для инъекций, которые часто считаются "избыточными" для игр.9

Во-вторых, это управление жизненным циклом. В веб-сервере, обрабатывающем десятки тысяч запросов в секунду, ручное управление созданием и уничтожением объектов (вызов new и Dispose) чревато ошибками. Контейнер берет на себя эту рутинную, но критически важную работу, гарантируя, что объект подключения к базе данных будет закрыт ровно в тот момент, когда завершится обработка HTTP-запроса, предотвращая утечки ресурсов.11

### 2.3 Анти-паттерн Service Locator в.NET

Несмотря на то, что IServiceProvider технически позволяет запрашивать сервисы напрямую через метод GetService<T>() (что функционально эквивалентно Service Locator), в современном.NET это считается строгим анти-паттерном.13

Использование Service Locator внутри классов скрывает зависимости. Взглянув на публичный интерфейс класса, использующего Service Locator, разработчик видит пустой конструктор и может ошибочно предположить, что класс легок в использовании. Однако при попытке его инстанцировать во время выполнения возникнет ошибка NullReferenceException или исключение разрешения сервиса, так как внутри методов скрыты вызовы к глобальному контейнеру. Это превращает ошибки конфигурации из ошибок времени компиляции (или времени старта) в ошибки времени выполнения, которые гораздо труднее отлаживать.9 Более того, Service Locator обходит механизмы управления областями видимости (Scope), что может привести к захвату короткоживущих зависимостей долгоживущими объектами (Captive Dependency), о чем будет подробно сказано в соответствующих разделах данного отчета.12

Сравнительный анализ подходов к разрешению зависимостей представлен в Таблице 1.

| **Характеристика** | **Service Locator (Unity Style)** | **Dependency Injection (.NET Enterprise)** | **Архитектурное влияние** |
| --- | --- | --- | --- |
| **Точка доступа** | GetComponent, FindObject, Singleton.Instance | Конструктор класса | Явное декларирование зависимостей против скрытых связей. |
| **Видимость связей** | Скрытая (нужно читать код методов) | Явная (видно в сигнатуре конструктора) | Упрощение рефакторинга и понимания кода в DI. |
| **Тестируемость** | Низкая (требует настройки глобального состояния) | Высокая (легкая подмена на Mocks) | DI способствует TDD и надежности системы. |
| **Управление временем жизни** | Ручное или привязанное к сцене | Автоматическое (Container-managed) | Снижение риска утечек ресурсов в высоконагруженных системах. |
| **Связность (Coupling)** | Жесткая (зависимость от конкретной реализации поиска) | Слабая (зависимость от интерфейсов) | Гибкость архитектуры и возможность замены реализаций. |

## 3. Анатомия DI-контейнера в.NET 8: Механика "Под капотом"

Понимание внутреннего устройства Microsoft.Extensions.DependencyInjection позволяет демистифицировать процесс создания объектов и осознать, почему некоторые операции (например, разрешение Scoped-сервиса в Singleton) запрещены на уровне архитектуры. В.NET 8 этот механизм был существенно оптимизирован для обеспечения максимальной производительности.

### 3.1 Двухфазная модель: Регистрация и Разрешение

Жизненный цикл DI-контейнера можно разделить на две четкие фазы, аналогичные фазам "Редактирование" и "Runtime" в Unity.

Первая фаза — **Регистрация (IServiceCollection)**. Это изменяемая коллекция (список) объектов ServiceDescriptor. В файле Program.cs разработчик заполняет этот список "чертежами" будущих объектов. Каждый дескриптор содержит три ключевых элемента: тип сервиса (интерфейс), тип реализации (конкретный класс) и время жизни (Lifetime). На этом этапе объекты еще не создаются; мы лишь описываем правила их создания.16 Это похоже на создание префабов в редакторе Unity — мы настраиваем компоненты, но они еще не существуют в памяти.

Вторая фаза — **Разрешение (IServiceProvider)**. После вызова метода builder.Build() коллекция "замораживается" и компилируется в IServiceProvider (часто реализуемый как ServiceProvider). Это движок времени выполнения, который отвечает за непосредственное создание объектов. Он анализирует конструкторы запрошенных типов, рекурсивно находит их зависимости, сверяется с правилами времени жизни и инстанцирует граф объектов.4 Важно отметить, что IServiceCollection больше не используется; добавить новые сервисы в уже собранный контейнер невозможно, что гарантирует целостность графа зависимостей во время работы приложения.

### 3.2 Внутренняя механика: Фабрика мест вызова (CallSiteFactory)

Когда приложение запрашивает сервис из IServiceProvider, контейнер не использует простую рефлексию (Activator.CreateInstance) для каждого вызова, так как это было бы катастрофически медленно для высоконагруженного веб-сервера.19 Вместо этого.NET использует сложный механизм кэширования и компиляции, ядром которого является CallSiteFactory.4

Процесс разрешения выглядит следующим образом:

Когда запрашивается сервис (например, Controller A), CallSiteFactory начинает обход дерева зависимостей. Если Controller A зависит от Service B, а тот — от Repository C, фабрика создает цепочку объектов ServiceCallSite (ConstructorCallSite, ConstantCallSite и т.д.), которые описывают алгоритм создания каждого узла.22 Это можно сравнить с тем, как Unity компилирует шейдеры или готовит данные для рендеринга кадра — предварительный расчет шагов.

Затем происходит самое интересное — Компиляция деревьев выражений (Expression Trees). В стандартном режиме работы контейнер берет построенную цепочку ServiceCallSite и динамически генерирует IL-код (Intermediate Language), который представляет собой оптимизированную функцию создания объекта. Грубо говоря, контейнер пишет и компилирует на лету C#-код вида:

return new ControllerA(new ServiceB(new RepositoryC()));

Полученный делегат кэшируется. Все последующие запросы к этому сервису просто вызывают этот скомпилированный делегат, что работает на порядки быстрее рефлексии и приближается к скорости прямого вызова конструктора (new).24

В.NET 8 была введена оптимизация, позволяющая контейнеру выбирать между интерпретацией CallSite (для редко используемых путей или при первом запуске) и полноценной компиляцией (для "горячих" путей). Это позволяет ускорить "холодный старт" приложения, не тратя время на JIT-компиляцию редко используемых графов зависимостей.25

### 3.3 Оптимизации производительности в.NET 8

Платформа.NET 8 привнесла ряд существенных улучшений в работу DI-контейнера, направленных на минимизацию блокировок и улучшение работы с памятью.

**ConcurrentDictionary и отсутствие блокировок:** Внутреннее кэширование разрешенных сервисов (особенно Синглтонов и Scoped-сервисов в рамках запроса) теперь использует оптимизированные структуры данных, минимизирующие конкуренцию потоков (Thread Contention). В более ранних версиях интенсивное обращение к синглтонам из тысяч потоков могло вызывать накладные расходы на синхронизацию; в.NET 8 использование ConcurrentDictionary с lock-free чтением практически устраняет эту проблему.26

**Dynamic PGO (Profile-Guided Optimization):** Включенная по умолчанию в.NET 8 динамическая профильная оптимизация позволяет JIT-компилятору "наблюдать" за работой скомпилированных DI-делегатов в реальном времени. Если JIT видит, что определенная ветка логики (например, выбор реализации в зависимости от ключа) выполняется чаще всего, он может перекомпилировать код на лету, встроив (inline) вызовы конструкторов и убрав лишние проверки типов. Это обеспечивает прирост производительности до 15-20% в сценариях с интенсивным созданием объектов.24

**Совместимость с Native AOT:** Важным вектором развития.NET 8 является поддержка Native AOT (Ahead-of-Time compilation) — компиляции в нативный код без JIT. Поскольку классический DI полагается на Reflection.Emit для генерации кода во время выполнения (что невозможно в AOT),.NET 8 внедряет поддержку Source Generators. Это позволяет генерировать код разрешения зависимостей на этапе компиляции проекта, полностью устраняя накладные расходы на старт приложения и делая DI пригодным для использования в средах с ограниченными ресурсами, таких как AWS Lambda или мобильные устройства.28

## 4. Жизненные циклы (Lifetimes): Временное измерение архитектуры

В Unity жизненный цикл объекта обычно имеет пространственную привязку: объект существует, пока загружена Сцена, или пока он не уничтожен явно. В ASP.NET Core концепция "Сцены" заменяется концепцией "Запроса" (Request), и время жизни объектов становится строго регламентированным временным интервалом. Непонимание этих интервалов является главной причиной утечек памяти и багов многопоточности у разработчиков, переходящих с Unity.

### 4.1 Transient: Эфемерные работники

**Определение:** Новый экземпляр сервиса создается *каждый раз*, когда он запрашивается из контейнера.30

**Механика:** Когда компонент запрашивает Transient зависимость, контейнер просто вызывает конструктор и отдает объект. Он не сохраняет ссылку на него (за исключением случаев с IDisposable, о чем ниже). Это полный аналог вызова Instantiate() в Unity или оператора new.

**Применение:** Легковесные сервисы без состояния (stateless), конвертеры данных, валидаторы, стратегии алгоритмов. Поскольку они не хранят состояние, их безопасно использовать в многопоточной среде — у каждого потока будет своя копия.

**Влияние на память (Gen0):** Объекты Transient обычно аллоцируются в **Поколении 0** (Generation 0) управляемой кучи. Поскольку они живут очень недолго (часто только в рамках одного метода или одной цепочки вызовов), Server GC очищает их чрезвычайно эффективно. В.NET аллокация в Gen0 стоит очень дешево (просто инкремент указателя), а сборка мусора этого поколения происходит без ощутимых пауз. Поэтому создание тысяч Transient-объектов в секунду является нормальным паттерном для ASP.NET Core, в отличие от Unity, где мы стараемся избегать аллокаций в Update.31

**Риск Disposable:** Если Transient-сервис реализует интерфейс IDisposable, контейнер *обязан* сохранить ссылку на него, чтобы вызвать Dispose в конце области видимости. Это может привести к тому, что короткоживущие объекты будут накапливаться в памяти до конца HTTP-запроса, создавая ненужное давление на GC. Это явление известно как "Захват Disposable Transient".13

### 4.2 Scoped: Контекст запроса

**Определение:** Один экземпляр создается на каждый **клиентский запрос** (или Scope). Все компоненты, участвующие в обработке этого запроса, получают один и тот же экземпляр.13

**Механика:** Когда HTTP-запрос поступает на сервер, middleware ASP.NET Core создает "Область видимости" (IServiceScope). В этой области есть свой кэш сервисов.

1. Контроллер запрашивает DbContext. Контейнер проверяет кэш текущего Scope.
2. Кэш пуст -> Создается новый DbContext. Он сохраняется в кэше Scope и возвращается контроллеру.
3. Сервис бизнес-логики (внедренный в тот же контроллер) тоже запрашивает DbContext.
4. Контейнер проверяет кэш, находит уже созданный DbContext и возвращает его.
5. Запрос завершен -> Scope уничтожается -> Вызывается Dispose для DbContext.

**Аналогия с Unity:** Это можно сравнить с "Синглтоном на уровне Сцены" или "Контекстом Уровня". Представьте объект LevelScoreManager, который создается при загрузке уровня, доступен всем врагам и игроку на этом уровне, и уничтожается при выгрузке уровня. В ASP.NET "Уровень" — это HTTP-запрос, который "загружается" и "выгружается" за миллисекунды.

**Паттерн Unit of Work:** Scoped — это фундамент транзакционной целостности. Благодаря тому, что DbContext является Scoped, все изменения данных, сделанные разными сервисами в рамках одного запроса, накапливаются в одном контексте и могут быть сохранены одной транзакцией (SaveChanges). Это гарантирует атомарность операций бизнес-логики.34

**Иллюзия состояния:** Unity-разработчики часто ищут, где в памяти хранится объект "Игрок". В REST API сервер **Stateless**. Нет постоянного объекта игрока в памяти. Scoped-сервис загружает данные игрока из БД, обрабатывает их и "забывает". Состояние живет в Базе Данных, а Scope — это временное окно обработки.

### 4.3 Singleton: Глобальное состояние

**Определение:** Единственный экземпляр создается при первом запросе и живет до остановки приложения.13

**Механика:** Экземпляр кэшируется в корневом контейнере. Любой запрос от любого пользователя из любой точки мира получает ссылку на один и тот же объект в памяти.NET. Это аналог DontDestroyOnLoad или static класса в Unity.

**Архитектурные риски:**

1. Потокобезопасность (Критическое отличие):  
   В Unity метод Update выполняется в одном главном потоке. Вам редко нужны блокировки (lock) для игровой логики.  
   В ASP.NET Core Singleton атакуется тысячами потоков одновременно. Если Singleton имеет изменяемое состояние (например, List<string> logs), и два запроса попытаются добавить запись одновременно, произойдет коллизия, потеря данных или исключение.33 Unity-разработчики должны заново учиться писать потокобезопасный код, используя ConcurrentDictionary, Interlocked или lock.13
2. Давление на память (LOH):  
   Синглы живут долго, поэтому GC быстро продвигает их в Поколение 2 (Gen2). Если Singleton хранит ссылки на большие структуры данных, они могут попасть в Large Object Heap (LOH). Сборка мусора в Gen2 и LOH — это дорогая операция, которая может вызывать паузы в работе сервера. Неаккуратное использование памяти в синглтонах может привести к фрагментации кучи, которую сложно диагностировать.31

## 5. Управление памятью: Server GC против Unity GC

Фундаментальное различие между Unity и ASP.NET Core кроется в работе Сборщика Мусора (Garbage Collector).

### 5.1 Архитектура сборщиков

**Unity (Boehm / Incremental):** Исторически Unity использует консервативный сборщик мусора (в старых версиях Mono) или инкрементальный сборщик в более новых версиях. Он работает в одном процессе с игрой и вынужден останавливать главный поток для очистки памяти. Именно поэтому в Unity существует золотое правило: "Не создавать объекты в Update", так как это провоцирует спайки лагов (GC Spikes).39

**ASP.NET Core (Server GC):** Серверный GC в.NET — это "зверь" совершенно другой породы. Он создан для максимальной пропускной способности (Throughput).

* **Выделенные потоки:** Server GC создает отдельную кучу (Heap) и отдельный поток сборки мусора для *каждого логического ядра процессора*. На 16-ядерном сервере у вас будет 16 параллельных сборщиков мусора.
* **Поколения (Generations):** Память делится на Gen0, Gen1 и Gen2.
  + **Gen0:** Место рождения всех новых объектов (Transient, Scoped). Очистка Gen0 происходит молниеносно и часто даже не требует приостановки рабочих потоков (в режиме Background GC).
  + **Gen1:** Буфер для объектов, переживших одну сборку.
  + **Gen2:** Дом для долгожителей (Singleton) и объектов, переживших много сборок. Полная сборка Gen2 дорогая, но происходит редко.31

**Вывод для разработчика:** Правило "No Allocations in Update" из Unity **не работает** в ASP.NET Core. Создание короткоживущих объектов (DTO, ViewModels) в контроллерах — это норма. Server GC "проглатывает" такие аллокации с невероятной скоростью. Оптимизировать нужно не количество new, а избегать попадания временных объектов в Gen2 (например, через случайное сохранение их в статических полях).42

### 5.2 Проблема фрагментации LOH

Особое внимание следует уделить Large Object Heap (LOH). Объекты размером более 85,000 байт (например, большие массивы байтов при загрузке файлов или буферы изображений) попадают сразу в LOH. В отличие от обычной кучи, LOH (до недавних версий.NET) не уплотнялась (не дефрагментировалась) по умолчанию, так как копирование больших кусков памяти слишком дорого.

Если Singleton-сервис будет постоянно создавать и выбрасывать большие массивы разного размера, LOH превратится в "швейцарский сыр" — много дырок, но нет места для нового большого куска. Это приведет к OutOfMemoryException, даже если свободной памяти много.31

Решение: Использовать ArrayPool<T> (пулинг массивов) для работы с большими буферами, вместо того чтобы полагаться на обычный new.38

## 6. Многопоточность и безопасность Синглтонов

Как уже упоминалось, потокобезопасность — это ахиллесова пята Unity-разработчика при переходе на бэкенд.

В Unity вы можете написать:

C#

public class GameManager : MonoBehaviour {  
 public static GameManager Instance;  
 public int Score;  
 public void AddScore() { Score++; } // Безопасно, так как только 1 поток  
}

В ASP.NET Core аналогичный код в Singleton-сервисе:

C#

public class ScoreService {  
 public int Score;  
 public void AddScore() { Score++; } // КАТАСТРОФА!  
}

Операция Score++ не является атомарной (это чтение, сложение, запись). Если два запроса вызовут AddScore одновременно, они могут считать одно и то же старое значение, увеличить его и записать. В итоге вместо +2 мы получим +1. Это состояние гонки (Race Condition).

**Механизмы защиты:**

1. **Immutability (Неизменяемость):** Лучший способ избежать проблем — не иметь изменяемого состояния в Синглтонах. Если данные не меняются, их можно читать параллельно без блокировок.
2. **Interlocked:** Для простых счетчиков используйте Interlocked.Increment(ref \_score). Это атомарная операция на уровне процессора, работающая быстрее любых блокировок.
3. **Locks:** Для сложной логики используйте lock(\_syncRoot) {... }.
4. **Concurrent Collections:** Вместо List<T> и Dictionary<TKey, TValue> используйте ConcurrentBag<T>, ConcurrentQueue<T> и ConcurrentDictionary<TKey, TValue>. Они спроектированы для безопасного доступа из множества потоков.36

## 7. Анти-паттерны и архитектурные ловушки

### 7.1 Captive Dependencies (Захваченные зависимости)

Это наиболее коварная ошибка, которую контейнер.NET 8 пытается предотвратить, но разработчики часто игнорируют предупреждения.

Суть проблемы: Вы внедряете Scoped сервис (например, DbContext) в конструктор Singleton сервиса (например, BackgroundWorker).

Механика сбоя:

1. При старте приложения создается Singleton.
2. Контейнер видит зависимость от Scoped-сервиса. Чтобы создать Singleton, он *вынужден* создать экземпляр Scoped-сервиса прямо сейчас.
3. Singleton сохраняет ссылку на этот экземпляр в своем приватном поле.
4. Поскольку Singleton живет вечно, этот "Scoped" экземпляр тоже живет вечно. Он стал "пленником" (Captive).
5. Он никогда не будет уничтожен (Disposed).

**Последствия:**

* **Утечка памяти:** DbContext кэширует все сущности, которые через него проходят. Поскольку он не уничтожается, кэш растет бесконечно.12
* **Ошибки данных:** Этот DbContext будет содержать данные, актуальные на момент старта приложения. Он никогда не увидит изменений, сделанных другими запросами.
* **Падение:** DbContext не потокобезопасен. Когда Singleton попытается использовать его из разных потоков таймера, приложение упадет.

**Диагностика:** В.NET 8 при запуске в среде Development включена опция ValidateScopes. Контейнер при старте проверяет граф зависимостей и выбрасывает InvalidOperationException, если находит такую ситуацию. Unity-разработчики ни в коем случае не должны отключать эту проверку!.44

Решение — IServiceScopeFactory:

Если Синглтону нужен Scoped-сервис, он должен инжектировать фабрику областей видимости.

C#

public class Worker(IServiceScopeFactory scopeFactory) {  
 public void DoWork() {  
 using (var scope = scopeFactory.CreateScope()) { // Создаем Scope вручную  
 var db = scope.ServiceProvider.GetRequiredService<MyDbContext>();  
 // Работаем с БД...  
 } // Scope умирает, DbContext умирает. Все чисто.  
 }  
}

Это похоже на ручное создание "кадра" обработки.13

## 8. Инновации.NET 8 и практическое применение

Платформа.NET 8 принесла инструменты, делающие DI более гибким и похожим на инструменты, к которым привыкли разработчики игр, но без потери типобезопасности.

### 8.1 Keyed Services (Ключевые сервисы)

До.NET 8, если у вас было несколько реализаций одного интерфейса (например, INotificationService: SmsNotification, EmailNotification), выбрать нужную во время выполнения было сложно. Приходилось писать "костыли" в виде фабрик или использовать IEnumerable<INotificationService> и фильтровать их вручную. Это часто толкало Unity-разработчиков обратно к Service Locator.

В.NET 8 появилась нативная поддержка **Keyed Services**:

C#

// Регистрация с ключами  
builder.Services.AddKeyedTransient<INotificationService, SmsService>("sms");  
builder.Services.AddKeyedTransient<INotificationService, EmailService>("email");  
  
// Инъекция по ключу  
public class AlertSystem( INotificationService sender) {... }

Это дает гибкость выбора реализации (как теги в Unity), но сохраняет чистоту DI.4

### 8.2 Short-Circuit Routing (Короткое замыкание)

Хотя это фича маршрутизации, она тесно связана с DI..NET 8 позволяет помечать маршруты (например, robots.txt или Health Checks) как .ShortCircuit().

Такие маршруты обрабатываются до того, как управление передается в тяжелый конвейер Middleware и до того, как создаются тяжелые Scoped-сервисы (аутентификация, сессия). Это оптимизирует нагрузку на DI-контейнер и GC, так как для служебных запросов не создается лишних объектов.11

## 9. Заключение

Для Unity-разработчика, осваивающего третий день шестой недели, ключевой инсайт заключается не в синтаксисе services.AddScoped, а в понимании времени. В Unity мы мыслим пространством (Сценой); в бэкенде мы мыслим временем (Запросом). Dependency Injection в ASP.NET Core — это не просто способ создания классов, это хронометр, который управляет жизнью и смертью данных.

Отказ от GetComponent и статических менеджеров в пользу конструкторной инъекции и IServiceProvider поначалу кажется ограничением свободы. Но именно это ограничение (Constraint) позволяет строить системы, которые могут обрабатывать миллионы пользователей, не падая под весом собственного состояния. Мы заменяем ручное управление хаосом на автоматизированный порядок контейнера.

Практические лабораторные работы этого дня (анализ GUID в разных Lifetime) призваны перевести эту теорию в опыт, наглядно показав, как Transient-сервисы мелькают и исчезают, как Scoped-сервисы создают контекст транзакции, и как Singleton-сервисы стоят незыблемыми, но опасными монументами в потоке данных.

#### Источники

1. План обучения ASP.NET Core: Неделя 6
2. What's the difference between the Dependency Injection and Service Locator patterns?, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/1557781/whats-the-difference-between-the-dependency-injection-and-service-locator-patte>
3. So Singletons are bad, then what? - Software Engineering Stack Exchange, дата последнего обращения: декабря 5, 2025, <https://softwareengineering.stackexchange.com/questions/40373/so-singletons-are-bad-then-what>
4. Dependency injection - .NET - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection>
5. Dependency Injection in .NET Core: A Comprehensive Guide for Modern Applications!!! | by Bhargava Koya - Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@bhargavkoya56/dependency-injection-in-net-core-a-comprehensive-guide-for-modern-applications-347342913d24>
6. Does GetComponent<>() impact performance - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/50205533/does-getcomponent-impact-performance>
7. Are there best practices for "Singletons" (persistent GameObjects) in Unity? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/59726190/are-there-best-practices-for-singletons-persistent-gameobjects-in-unity>
8. Singleton vs Dependency Injection vs Service Locator vs Scriptable Objects : r/Unity3D, дата последнего обращения: декабря 5, 2025, <https://www.reddit.com/r/Unity3D/comments/13ooktu/singleton_vs_dependency_injection_vs_service/>
9. Is ServiceLocator an anti-pattern? | by iamprovidence - Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@iamprovidence/is-servicelocator-antipattern-7c54d6c1a3a1>
10. What's the difference between using dependency injection with a container and using a service locator? - Software Engineering Stack Exchange, дата последнего обращения: декабря 5, 2025, <https://softwareengineering.stackexchange.com/questions/390755/whats-the-difference-between-using-dependency-injection-with-a-container-and-us>
11. Service Lifetimes in ASP.NET Core | endjin - Azure Data Analytics Consultancy UK, дата последнего обращения: декабря 5, 2025, <https://endjin.com/blog/2022/09/service-lifetimes-in-aspnet-core>
12. c# - .Net 5 Console Background service exception with DI of Entity Framework Core, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/70049676/net-5-console-background-service-exception-with-di-of-entity-framework-core>
13. Dependency injection guidelines - .NET | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-guidelines>
14. Service Locator is an Anti-Pattern - ploeh blog, дата последнего обращения: декабря 5, 2025, <https://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/>
15. Is ServiceLocator an anti-pattern? [closed] - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/22795459/is-servicelocator-an-anti-pattern>
16. Tutorial: Use dependency injection in .NET - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-usage>
17. Microsoft.Extensions.DependencyInjection Namespace, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection?view=net-10.0-pp>
18. Dependency Injection in .NET Core 8: Concepts and Best Practices | by Pedro Torres, дата последнего обращения: декабря 5, 2025, <https://medium.com/@pedrolisboa_10855/dependency-injection-in-net-core-8-concepts-and-best-practices-0990211fc424>
19. c# - How costly is .NET reflection? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/25458/how-costly-is-net-reflection>
20. How can I Avoid Overusing Reflection in .NET | by Engr. Md. Hasan Monsur | ASP DOTNET, дата последнего обращения: декабря 5, 2025, <https://medium.com/asp-dotnet/how-can-i-avoid-overusing-reflection-in-net-1414d66288fd>
21. ASP.NET Core Dependency Injection: What is the IServiceProvider and how is it Built?, дата последнего обращения: декабря 5, 2025, <https://www.stevejgordon.co.uk/aspnet-core-dependency-injection-what-is-the-iserviceprovider-and-how-is-it-built>
22. Unobserved Task Exception(s) around DependencyInjection.ServiceLookup.get\_ServiceType() · Issue #37337 · dotnet/runtime - GitHub, дата последнего обращения: декабря 5, 2025, <https://github.com/dotnet/runtime/issues/37337>
23. Constantly getting "System.InvalidOperationException: No service for type 'Microsoft.AspNetCore.Components.WebAssembly.Authentication.BaseAddressAuthorizationMessageHandler' has been registered." exception in .NET 6 Blazor WASM hosted app., дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/answers/questions/1290884/constantly-getting-system-invalidoperationexceptio>
24. Performance Improvements in .NET 8 - Microsoft Dev Blogs, дата последнего обращения: декабря 5, 2025, <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/>
25. Performance Improvements in .NET 10 - Microsoft Dev Blogs, дата последнего обращения: декабря 5, 2025, <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-10/>
26. Performance Improvements in .NET 8 | by Rico Mariani - Medium, дата последнего обращения: декабря 5, 2025, <https://ricomariani.medium.com/performance-improvements-in-net-8-f673e805e09e>
27. What's new in .NET 8 runtime - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8/runtime>
28. Native AOT deployment overview - .NET | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/>
29. 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>
30. 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>
31. Large object heap (LOH) on Windows - .NET | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/large-object-heap>
32. Fundamentals of garbage collection - .NET | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals>
33. Do I need to lock singleton in ASP.NET Core? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/45527228/do-i-need-to-lock-singleton-in-asp-net-core>
34. Service Locator is not an Anti-Pattern - Jimmy Bogard, дата последнего обращения: декабря 5, 2025, <https://www.jimmybogard.com/service-locator-is-not-an-anti-pattern/>
35. Mastering Dependency Injection in .NET 8: A Comprehensive Guide | by Ashen Dunusinghe, дата последнего обращения: декабря 5, 2025, <https://medium.com/@avdunusinghe/mastering-dependency-injection-in-net-8-a-comprehensive-guide-bde6f783a54a>
36. c# - Singleton and Thread Safety? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/42470041/singleton-and-thread-safety>
37. Dependency injection in ASP.NET Core | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-10.0>
38. Extensive use of LOH causes significant performance issue - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/27381887/extensive-use-of-loh-causes-significant-performance-issue>
39. Let's figure out here once and for all how the garbage collector and memory work in native C# vs Unity. : r/Unity3D - Reddit, дата последнего обращения: декабря 5, 2025, <https://www.reddit.com/r/Unity3D/comments/1of15ld/lets_figure_out_here_once_and_for_all_how_the/>
40. Garbage collection best practices - Unity - Manual, дата последнего обращения: декабря 5, 2025, <https://docs.unity3d.com/2022.2/Documentation/Manual/performance-garbage-collection-best-practices.html>
41. Automatic Resource Management in .NET 7 Core - CODE Magazine, дата последнего обращения: декабря 5, 2025, <https://www.codemag.com/Article/2401081/Automatic-Resource-Management-in-.NET-7-Core>
42. 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>
43. Large Object Heap Fragmentation - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/686950/large-object-heap-fragmentation>
44. Validate that your DI Container can be created in ASP.NET Core - Steven Giesel, дата последнего обращения: декабря 5, 2025, <https://steven-giesel.com/blogPost/d67df2e6-b8f7-4fd8-92ba-426e268b5216/validate-that-your-di-container-can-be-created-in-aspnet-core>
45. Enable DI container scope validation by default for Development environment in all ASP.NET Core 2.0 templates · Issue #640 - GitHub, дата последнего обращения: декабря 5, 2025, <https://github.com/dotnet/templating/issues/640>