# Архитектурный синтез и построение микро-фреймворка на платформе.NET 8: Теоретический базис пятого дня

## 1. Введение: От игрового цикла к конвейеру обработки

Данный отчет представляет собой всесторонний теоретический анализ архитектурных концепций, составляющих основу пятого дня шестой недели программы профессиональной переподготовки Unity-разработчиков на платформу.NET Enterprise Backend. Документ служит фундаментальным справочным материалом для выполнения финального интеграционного задания недели — создания собственного веб-фреймворка.

В контексте перехода от разработки игр (GameDev) к корпоративной серверной разработке (Enterprise Backend), пятый день является точкой сингулярности, где разрозненные знания о Dependency Injection, Middleware и асинхронном программировании объединяются в единую ментальную модель. Если в среде Unity разработчик оперирует концепцией непрерывного игрового цикла (Game Loop), где состояние мира мутирует кадр за кадром в детерминированном порядке 1, то в ASP.NET Core архитектура инвертируется. Приложение переходит в состояние ожидания (Idle), реактивно откликаясь на внешние раздражители (HTTP-запросы) через сложную систему делегатов и событий.

Основной фокус данного документа сосредоточен на глубоком анализе подсистемы маршрутизации (Routing), механизмах "короткого замыкания" (Short-Circuiting) в.NET 8, внутренней механике Minimal APIs и стратегии построения модульных систем без использования тяжеловесных контроллеров. Анализ проводится с позиции старшего архитектора, обосновывающего выбор тех или иных паттернов для построения высоконагруженных систем.

### 1.1. Сравнительная характеристика архитектурных парадигм

Для формирования правильной ментальной модели необходимо четко разграничить подходы к управлению потоком выполнения. В Unity MonoBehaviour часто выступает как контейнер и логики, и данных, живущий в памяти долгое время. В ASP.NET Core компоненты обработки (Middleware, Endpoints) — это stateless-сервисы, которые создаются и уничтожаются (или переиспользуются) с высокой частотой, требуя строгой дисциплины управления памятью и зависимостями.1

Ниже приведена таблица, иллюстрирующая ключевые различия, критичные для понимания материала пятого дня.

| **Архитектурный аспект** | **Unity (Game Loop)** | **ASP.NET Core (Request Pipeline)** | **Техническое следствие для Дня 5** |
| --- | --- | --- | --- |
| **Инициация кода** | Полинговая система: Update() вызывается движком каждый кадр. | Событийная система: Код выполняется только при поступлении HTTP-пакета. | Необходимость понимания "холодного старта" и инициализации маршрутов при запуске приложения.2 |
| **Маршрутизация** | Императивная: SceneManager.LoadScene("Level1") или активация GameObject. | Декларативная: Сопоставление URL-шаблона /api/level/1 с делегатом кода. | Изучение разделения фаз сопоставления (UseRouting) и выполнения (UseEndpoints).3 |
| **Передача данных** | Прямые ссылки, статические поля, GetComponent. | HttpContext как единый контейнер данных, передаваемый по цепочке. | Понимание того, как UseRouting обогащает HttpContext метаданными для последующих слоев.4 |
| **Оптимизация** | Culling, LOD (Level of Detail), Object Pooling. | Short-Circuiting, Caching, Middleware Ordering. | Использование .ShortCircuit() для мгновенного возврата ответа, минуя тяжелую логику.5 |

## 2. Глубокое погружение в маршрутизацию: Анатомия и Механика

Маршрутизация в ASP.NET Core — это не просто механизм выбора метода для вызова. Это сложная подсистема, интегрированная в конвейер Middleware, которая позволяет принимать решения о судьбе запроса задолго до того, как будет выполнен бизнес-код. Понимание внутренней работы EndpointRoutingMiddleware и EndpointMiddleware является ключом к построению безопасных и эффективных API.

### 2.1. Двухфазная модель: Разделение ответственности

До появления Endpoint Routing (в.NET Core 2.x) маршрутизация была "черным ящиком", скрытым внутри MVC. Современный подход разделяет процесс на две дискретные фазы, что дает разработчику беспрецедентный контроль.4

#### 2.1.1. Фаза 1: Сопоставление (The Matching Phase) — UseRouting

Метод app.UseRouting() регистрирует в конвейере EndpointRoutingMiddleware.3 Этот компонент не выполняет бизнес-логику. Его задача — "смотреть на карту и выбирать дорогу".

Внутренняя механика процесса выглядит следующим образом:

1. **Построение графа маршрутов:** При старте приложения DfaMatcher строит оптимизированный граф (Deterministic Finite Automaton) из всех зарегистрированных маршрутов (MapGet, MapPost, контроллеры).6
2. **Анализ запроса:** При поступлении запроса middleware анализирует URL path, HTTP-метод и заголовки.
3. **Выбор Endpoint:** На основе графа выбирается наиболее подходящий Endpoint. Объект Endpoint содержит два критических компонента:
   * RequestDelegate: Делегат, который *будет* выполнен.
   * EndpointMetadataCollection: Коллекция метаданных (атрибуты, маркерные интерфейсы).3
4. **Мутация Контекста:** Самое важное действие — middleware сохраняет найденный Endpoint внутри HttpContext (через фичу IEndpointFeature). Сам код endpoint'а **не запускается**.8

*Аналогия с Unity:* Это эквивалент фазы Physics.Raycast. Мы выпускаем луч и получаем информацию о том, *в какой* коллайдер мы попали (RaycastHit), но еще не вызываем метод OnHit(). Мы просто "пометили" цель.

#### 2.1.2. Фаза 2: Выполнение (The Execution Phase) — UseEndpoints

Метод app.UseEndpoints(...) (или неявный вызов в конце Program.cs в.NET 6+) регистрирует EndpointMiddleware.3 Это терминальный компонент.

Его логика проста, но критична:

1. Он извлекает Endpoint из HttpContext, который был помещен туда ранее UseRouting.
2. Если endpoint найден, он вызывает его RequestDelegate.
3. Если endpoint не найден (и запрос дошел до этой точки), возвращается 404.

*Аналогия с Unity:* Это вызов метода SendMessage("OnHit") на объекте, который мы нашли ранее.

### 2.2. "Middleware Gap": Пространство стратегического маневра

Разделение на UseRouting и UseEndpoints создает уникальную архитектурную зону, которую можно назвать "Middleware Gap" (Промежуток Middleware).3

Любой Middleware, помещенный в этот промежуток, обладает "сверхспособностью": он уже знает, *какой* код должен выполниться (благодаря UseRouting), но выполнение еще не началось. Это идеальное место для реализации сквозной функциональности (Cross-Cutting Concerns), зависящей от конкретного маршрута.

Пример архитектурного применения:

Представьте Middleware авторизации. Если поместить его до UseRouting, он не будет знать, требует ли целевой метод контроллера атрибута [Authorize], так как метод еще не выбран. Помещая его между вызовами, мы позволяем ему прочитать метаданные выбранного endpoint'а и принять решение: пропустить запрос дальше к UseEndpoints или вернуть 401 Unauthorized.3

C#

// Псевдокод логики Middleware в "Gap"  
public async Task Invoke(HttpContext context, RequestDelegate next)  
{  
 // 1. Узнаем, куда летит запрос (спасибо UseRouting!)  
 var endpoint = context.GetEndpoint();   
   
 // 2. Проверяем метаданные  
 var authAttr = endpoint?.Metadata.GetMetadata<AuthorizeAttribute>();  
   
 if (authAttr!= null &&!UserIsAuthenticated(context))  
 {  
 // 3. Блокируем выполнение, не вызывая next  
 context.Response.StatusCode = 401;  
 return;   
 }  
   
 // 4. Пропускаем к UseEndpoints  
 await next(context);   
}

### 2.3. Работа с метаданными (EndpointMetadata)

Для Unity-разработчика метаданные можно сравнить с компонентами MonoBehaviour или атрибутами ``, но действующими в рантайме. HttpContext.GetEndpoint() возвращает объект, который является хранилищем всей статической информации о маршруте.9

Это позволяет создавать мощные, декларативные системы. Например, можно создать атрибут ``, пометить им метод Minimal API, и написать middleware, который будет динамически считывать это число и ограничивать трафик, не зная ничего о бизнес-логике самого метода.10

## 3. Внутреннее устройство Minimal APIs: От Лямбды к Делегату

Minimal APIs, введенные в.NET 6, часто воспринимаются как "синтаксический сахар", но под капотом они используют сложнейшие механизмы метапрограммирования, обеспечивающие производительность, сравнимую или превосходящую классические контроллеры.

### 3.1. Проблема типизации

В ASP.NET Core конвейер ожидает делегат типа RequestDelegate (Func<HttpContext, Task>). Однако в Minimal API мы пишем лямбды с произвольными аргументами:

C#

app.MapPost("/save", (SaveData data, ILogger log) => {... });

Как типизированные аргументы SaveData и ILogger попадают в функцию, которая принимает только HttpContext?

### 3.2. Роль RequestDelegateFactory

Здесь вступает в игру RequestDelegateFactory.12 В.NET 8 этот процесс часто оптимизируется на этапе компиляции с помощью **Request Delegate Generator (RDG)** для поддержки Native AOT, но логика остается прежней.

При запуске приложения фабрика анализирует сигнатуру лямбды и генерирует "код-обертку" (wrapper). Этот процесс включает:

1. **Интроспекцию параметров:** Анализируются типы аргументов.
   * Если тип известен DI-контейнеру (как ILogger), генерируется код serviceProvider.GetService<ILogger>().
   * Если тип является сложным объектом (POCO), предполагается чтение из тела запроса (JSON). Генерируется вызов FromJsonAsync.
   * Если тип примитивный (int, string), он ищется в Route Values или Query String.14
2. **Генерацию конвейера привязки:** Создается оптимизированный код, который извлекает данные из HttpContext, валидирует их и передает в вашу лямбду.

### 3.3. Деконструкция MapPost

Рассмотрим детально, что происходит при вызове app.MapPost("/api", (UserDto user) =>...).13 Сгенерированный код выполняет следующую последовательность:

1. **Проверка Content-Type:** Убеждается, что пришел application/json.
2. **Десериализация:** Асинхронно читает Body потока запроса и преобразует его в UserDto с помощью System.Text.Json.
3. **Валидация (опционально):** Если включена валидация, проверяет модель.
4. **Вызов хендлера:** Передает десериализованный объект в пользовательский код.
5. **Сериализация ответа:** Получает результат (например, Results.Ok(user)), выполняет его метод ExecuteAsync, который пишет JSON обратно в поток ответа.17

Это объясняет, почему в Minimal API не нужны явные атрибуты `` — фреймворк выводит источник данных на основе типа параметра. Для Unity-разработчика это аналогично тому, как JsonUtility.FromJson автоматически мапит поля, но здесь это происходит на уровне всего сетевого стека.

## 4. Оптимизация производительности: Short-Circuit Routing (.NET 8)

Одной из ключевых инноваций.NET 8 стало введение механизма "короткого замыкания" маршрутов (Short-Circuit Routing).5 Эта функция позволяет существенно сэкономить ресурсы сервера для определенных типов запросов.

### 4.1. Архитектурная проблема "Длинного Пути"

В классической схеме *каждый* запрос проходит через *все* зарегистрированные middleware. Даже запрос на получение фавиконки (favicon.ico) или проверку жизни сервиса (/health) будет проходить через аутентификацию, авторизацию, CORS, и сложные механизмы логирования. Это создает избыточную нагрузку на CPU и память (аллокации объектов контекста безопасности), что в высоконагруженных системах (High Load) неприемлемо.

### 4.2. Решение: .ShortCircuit()

Метод расширения ShortCircuit() (или MapShortCircuit для группы маршрутов) изменяет поведение EndpointRoutingMiddleware.3

Когда маршрутизатор находит endpoint, помеченный как "short-circuit", он **не передает управление** следующему компоненту в цепи (через next()). Вместо этого он немедленно вызывает делегат этого endpoint'а и завершает обработку запроса.3

Техническая реализация (концептуально):

C#

// Внутри EndpointRoutingMiddleware  
var endpoint = FindEndpoint(context);  
if (endpoint!= null)  
{  
 context.SetEndpoint(endpoint);  
 if (endpoint.Metadata.Has<ShortCircuitMetadata>())  
 {  
 // Мгновенное выполнение, обход всего остального конвейера  
 await endpoint.RequestDelegate(context);   
 return;   
 }  
}  
await \_next(context); // Стандартный путь

### 4.3. Стратегическое применение и риски

Использование данного механизма оправдано для:

* **Static Assets:** robots.txt, favicon.ico.19
* **Health Checks:** Эндпоинты для Kubernetes/Docker, которые вызываются каждую секунду.
* **Ping/Pong:** Простейшие API.

**Риски:** Поскольку запрос не проходит через последующие middleware, для short-circuit маршрутов **не будут работать**:

* CORS (если middleware CORS стоит после Routing).
* Авторизация (нельзя защитить такой маршрут стандартным [Authorize], так как UseAuthorization не выполнится).3
* Глобальное логирование запросов (если оно настроено через middleware в конце цепи).

## 5. Dependency Injection и Управление Состоянием

Переход к DI — это отказ от статических менеджеров (Singleton в стиле Unity) в пользу управляемых контейнером сервисов. В рамках пятого дня критически важно закрепить понимание жизненных циклов, так как ошибки здесь ведут к трудноуловимым багам.

### 5.1. Физика жизненных циклов (Lifetimes)

| **Lifetime** | **Аналогия с Unity** | **Поведение в ASP.NET Core** | **Потокобезопасность** |
| --- | --- | --- | --- |
| **Transient** | Instantiate() каждый раз. | Создается при *каждой* инъекции. Легковесный, без состояния. | Не требуется (объект локален). |
| **Scoped** | Объект, живущий в рамках одной сцены (запроса). | Создается один раз на HTTP-запрос. Общий для всех компонентов, обрабатывающих этот запрос. | Не требуется (в рамках одного запроса поток обычно один). |
| **Singleton** | DontDestroyOnLoad. | Создается один раз на все время жизни приложения. Общий для тысяч одновременных запросов. | **Критически важна.** Требуется lock или Concurrent коллекции.1 |

### 5.2. Captive Dependencies (Захваченные зависимости)

Это наиболее частая ошибка при переходе с Unity. Если вы внедрите **Scoped** сервис (например, DbContext для работы с БД) в конструктор **Singleton** сервиса, произойдет "захват". Singleton живет вечно, следовательно, он будет удерживать ссылку на Scoped сервис вечно.

**Последствия:**

1. **Утечка памяти:** DbContext накапливает кэш сущностей и никогда не очищается.
2. **Ошибки конкурентного доступа:** С одним и тем же экземпляром DbContext начнут работать разные потоки от разных запросов, что вызовет исключение, так как DbContext не потокобезопасен.

**Решение:** Внедрять IServiceScopeFactory в Singleton и создавать область видимости (scope) вручную для каждой операции:

C#

using (var scope = \_scopeFactory.CreateScope())  
{  
 var db = scope.ServiceProvider.GetRequiredService<MyDbContext>();  
 // Работа с БД  
}

### 5.3. Keyed Services (.NET 8)

В.NET 8 появилась нативная поддержка **Keyed Services** (Ключевых сервисов), что позволяет регистрировать несколько реализаций одного интерфейса и выбирать нужную по ключу.2 Это аналог паттерна "Стратегия", часто реализуемого в Unity через Enum и фабрики.

Пример регистрации:

C#

builder.Services.AddKeyedSingleton<ICache, RedisCache>("redis");  
builder.Services.AddKeyedSingleton<ICache, MemoryCache>("memory");

Пример использования в Minimal API:

C#

app.MapGet("/data", ( ICache cache) => {... });

Это упрощает архитектуру, убирая необходимость в написании кастомных резолверов зависимостей.

## 6. Лаборатория синтеза: Создание Микро-Фреймворка

Финальная задача дня — не просто написать код, а спроектировать архитектуру. Студенту предлагается создать приложение "без контроллеров" (Frameworkless), используя только примитивы маршрутизации.

### 6.1. Архитектура "Onion Pipeline"

Мы выстраиваем Middleware в порядке "Луковицы", где каждый слой имеет свою ответственность:

1. **Error Handling:** Внешний слой для перехвата всех исключений (try-catch).
2. **Profiling:** Замер времени выполнения (Stopwatch).
3. **Routing (UseRouting):** Определение цели.
4. **Authorization (Custom):** Слой безопасности в "Middleware Gap". Он проверяет метаданные найденного endpoint'а.
5. **Execution (UseEndpoints):** Выполнение бизнес-логики.

### 6.2. Реализация паттерна MapEndpoint

Для структурирования Minimal API, чтобы Program.cs не превратился в "God Object" (аналог огромного скрипта GameManager), применяется паттерн, похожий на библиотеку Carter или REPR (Request-Endpoint-Response).20

Создается интерфейс IEndpoint:

C#

public interface IEndpoint  
{  
 void Map(IEndpointRouteBuilder app);  
}

И механизм авто-регистрации через Reflection:

C#

// Сканирование сборки и вызов Map() для каждого endpoint'а  
var endpoints = Assembly.GetExecutingAssembly()  
 .GetTypes()  
 .Where(t => typeof(IEndpoint).IsAssignableFrom(t) &&!t.IsInterface &&!t.IsAbstract)  
 .Select(Activator.CreateInstance)  
 .Cast<IEndpoint>();  
  
foreach (var endpoint in endpoints)  
{  
 endpoint.Map(app);  
}

Этот подход учит разработчика создавать расширяемые системы, где добавление новой функциональности (нового файла класса) не требует изменения существующего кода инициализации (Open/Closed Principle).

## 7. Заключение

Пятый день шестой недели завершает трансформацию разработчика из "пользователя фреймворка" в "инженера платформы". Глубокое понимание разницы между UseRouting и UseEndpoints дает власть над потоком управления. Знание работы RequestDelegateFactory снимает "магию" с Minimal APIs. А владение инструментами оптимизации вроде Short-Circuiting и Keyed Services позволяет строить высокопроизводительные системы уровня Enterprise.

Переход от Unity завершен не на уровне синтаксиса (C# везде одинаков), а на уровне архитектурного мышления: от цикла кадров к конвейеру запросов, от монолитных скриптов к композиции сервисов, от ручного управления ссылками к строгому DI.

## 8. Глоссарий терминов

| **Термин** | **Определение в контексте ASP.NET Core** |
| --- | --- |
| **Middleware** | Компонент конвейера обработки запросов. Аналог фильтра или звена цепи. |
| **Endpoint** | Объект, представляющий конечную точку обработки (URL + Метод + Делегаты + Метаданные). |
| **Short-Circuit** | Прерывание выполнения конвейера middleware для немедленного возврата ответа. |
| **Captive Dependency** | Архитектурная ошибка внедрения зависимости с коротким временем жизни в зависимость с длинным временем жизни. |
| **RequestDelegate** | Функция Func<HttpContext, Task>, представляющая собой исполняемый блок кода в конвейере. |
| **Metadata** | Дополнительные данные (атрибуты), прикрепленные к маршруту, используемые middleware для принятия решений. |

#### Источники

1. План обучения ASP.NET Core: Неделя 6
2. I still don't understand why we change where to put endpoint registration from UseRouting to UseEndpoint. · dotnet aspnetcore · Discussion #28157 - GitHub, дата последнего обращения: декабря 5, 2025, <https://github.com/dotnet/aspnetcore/discussions/28157>
3. Routing in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-10.0>
4. What are the differences between app.UseRouting() and app.UseEndPoints()?, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/57846127/what-are-the-differences-between-app-userouting-and-app-useendpoints>
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. aspnetcore/src/Http/Routing/test/UnitTests/EndpointRoutingMiddlewareTest.cs at main - GitHub, дата последнего обращения: декабря 5, 2025, <https://github.com/dotnet/aspnetcore/blob/main/src/Http/Routing/test/UnitTests/EndpointRoutingMiddlewareTest.cs>
7. RequestDelegateResult Class (Microsoft.AspNetCore.Http), дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.requestdelegateresult?view=aspnetcore-9.0>
8. How does the "UseRouting" method work in ASP.NET Core? - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/answers/questions/1350132/how-does-the-userouting-method-work-in-asp-net-cor>
9. EndpointHttpContextExtensions.GetEndpoint(HttpContext) Method (Microsoft.AspNetCore.Http), дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.endpointhttpcontextextensions.getendpoint?view=aspnetcore-10.0>
10. Implementing fine-grained access control with ASP.NET Core custom endpoint metadata, дата последнего обращения: декабря 5, 2025, <https://medium.com/@asimmon/implementing-fine-grained-access-control-with-asp-net-core-custom-endpoint-metadata-3191b6d052b7>
11. Use Attributes & Middleware in ASP.NET Core for Logging, Telemetry, and Anything Else, дата последнего обращения: декабря 5, 2025, <https://michaelscodingspot.com/attributes-and-middleware-in-asp-net-core/>
12. ASP.NET Core Request Delegate Generator (RDG) for Native AOT | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/aot/request-delegate-generator/rdg?view=aspnetcore-10.0>
13. How is Delegate being cast to RequestDelegate in ASP.NET Core? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/73426685/how-is-delegate-being-cast-to-requestdelegate-in-asp-net-core>
14. Model Binding in ASP.NET Core | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-10.0>
15. aspnetcore/src/Http/Http.Extensions/src/RequestDelegateFactory.cs at main · dotnet ... - GitHub, дата последнего обращения: декабря 5, 2025, <https://github.com/dotnet/aspnetcore/blob/main/src/Http/Http.Extensions/src/RequestDelegateFactory.cs>
16. Tutorial: Create a minimal API with ASP.NET Core | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/tutorials/min-web-api?view=aspnetcore-10.0>
17. Create responses in Minimal API applications | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/responses?view=aspnetcore-10.0>
18. Introduction to ShortCircuit and MapShortCircuit in .NET 8 - DEV Community, дата последнего обращения: декабря 5, 2025, <https://dev.to/moh_moh701/introduction-to-shortcircuit-and-mapshortcircuit-in-net-8-12ml>
19. Route Short Circuit middleware in ASP.NET Core - Amit Prakash, дата последнего обращения: декабря 5, 2025, <https://www.amitprakash.me/blog/short-circuit-middleware>
20. How to structure your Minimal API in .NET? - Treblle, дата последнего обращения: декабря 5, 2025, <https://treblle.com/blog/how-to-structure-your-minimal-api-in-net>