# Стратегический анализ архитектуры Middleware в ASP.NET Core: Теоретический базис первого дня

## 1. Введение: Фундаментальный сдвиг парадигмы

Настоящий документ представляет собой исчерпывающее теоретическое руководство, разработанное специально для первого дня шестой недели учебного плана «Переход от Unity к Enterprise.NET Backend». Данный этап обучения, обозначенный в стратегическом отчете как «Анатомия Конвейера» (Middleware Mechanics), является критической точкой бифуркации в ментальной модели инженера.1

Для разработчика, чей профессиональный опыт сформирован в среде Unity, переход к ASP.NET Core требует деконструкции привычных паттернов управления жизненным циклом приложения. В игровой индустрии доминирует детерминированный, циклический процесс — Game Loop, где состояние мира обновляется дискретными квантами времени (кадрами), а объекты (MonoBehaviours) часто обладают персистентным состоянием и временем жизни, сопоставимым с длительностью игровой сессии.2

В противоположность этому, архитектура ASP.NET Core базируется на событийной, линейной модели обработки запросов (Request-Response Pipeline). Здесь нет «кадров» и нет глобального состояния, сохраняющегося между итерациями по умолчанию. Вместо этого система оперирует тысячами изолированных, короткоживущих контекстов (HttpContext), которые пронизывают слоистую структуру компонентов, называемых Middleware (промежуточное ПО).3

Понимание этой архитектуры требует глубокого погружения не только в синтаксис C#, но и в принципы работы веб-сервера Kestrel, модель асинхронности Task-based Asynchronous Pattern (TAP) и паттерны проектирования, лежащие в основе конвейера. Цель данного документа — предоставить максимально детализированную теоретическую базу, необходимую для выполнения лабораторных работ первого дня, включая создание «Инспектора Конвейера» и реализацию кастомных компонентов обработки запросов.1 Мы проанализируем внутреннее устройство RequestDelegate, механизм «матрешки» (Russian Doll), потокобезопасность и новейшие оптимизации, внедренные в.NET 8, такие как Short-Circuit Routing.4

## 2. Сравнительная архитектура: Game Loop против Request Pipeline

Центральным конфликтом при переходе с Unity на Backend является различие в инициации и потоке исполнения кода. Понимание этого различия — ключ к предотвращению критических ошибок, связанных с управлением памятью и многопоточностью.

### 2.1. Анатомия кадра в Unity (Contextual Baseline)

В Unity точкой входа для логики является событийная модель движка. Движок вызывает методы Update(), FixedUpdate(), LateUpdate() для всех активных объектов в сцене.

* **Цикличность:** Код выполняется в бесконечном цикле while(gameIsRunning).
* **Состояние (Stateful):** Переменные класса (private int health) сохраняют свои значения между вызовами Update.
* **Однопоточность:** Основная логика выполняется в главном потоке (Main Thread). Доступ к API движка (например, Transform) потоконебезопасен и ограничен главным потоком.5

### 2.2. Анатомия запроса в ASP.NET Core

В ASP.NET Core приложение большую часть времени может находиться в состоянии ожидания (Idle). Исполнение кода инициируется внешним событием — поступлением HTTP-пакета на сокет веб-сервера.

* **Линейность:** Для каждого запроса создается отдельная цепочка выполнения. Запрос проходит от первого middleware к последнему и возвращается обратно.
* **Отсутствие состояния (Stateless):** Объект контроллера или сервиса (в зависимости от Lifetime) часто создается заново для каждого запроса. Поля класса сбрасываются. Сохранение данных между запросами требует внешних хранилищ (БД, Redis).
* **Многопоточность:** Сервер обрабатывает тысячи запросов одновременно, используя пул потоков (ThreadPool). Один и тот же код middleware может исполняться параллельно в разных потоках для разных пользователей.1

### 2.3. Таблица архитектурных расхождений

Ниже приведен детальный сравнительный анализ, иллюстрирующий, как концепции Unity трансформируются в концепции Backend-разработки.

| **Характеристика** | **Unity (Game Dev)** | **ASP.NET Core (Backend)** | **Архитектурное следствие для Middleware** |
| --- | --- | --- | --- |
| **Инициатор исполнения** | Таймер движка (vsync / target frame rate). | Входящий I/O запрос (HTTP request). | Middleware не работает "постоянно", оно реагирует на событие. |
| **Модель потока** | **Sequence (Последовательность):** Объект А завершает Update, затем начинает Объект Б. | **Nesting (Вложенность):** Middleware А вызывает Middleware Б и *ждет* его завершения. | Использование await next() создает стек вызовов, напоминающий рекурсию. |
| **Время жизни данных** | **Scene / Application:** Данные живут пока загружена сцена. | **Request:** Данные живут миллисекунды (Scope). | Нельзя использовать static поля для хранения данных пользователя (Race Condition).6 |
| **Обработка ошибок** | **Локальная:** Ошибка в одном скрипте часто не ломает весь кадр (try-catch внутри движка). | **Глобальная:** Неперехваченное исключение прерывает запрос и возвращает 500 Error. | Необходимость глобального Middleware для перехвата исключений в начале конвейера.7 |
| **Оптимизация** | **Per-Frame:** Минимизация аллокаций в Update (GC Pressure). | **Per-Request:** Минимизация блокировок потоков (Async/Await) и аллокаций. | Блокирующий код (.Result) убивает производительность сервера, в отличие от фриза кадра в Unity. |

## 3. Концептуальная модель Middleware: «Матрешка» (Russian Doll)

В основе обработки запросов в ASP.NET Core лежит паттерн, который в технической литературе часто называют **«Russian Doll» (Матрешка)** или **Pipeline (Конвейер)**. В отличие от линейного списка событий в Unity (UnityEvent), middleware вкладываются друг в друга.8

### 3.1. Визуализация потока управления

Представьте запрос как физический объект, проходящий через серию слоев.

1. **Вход (Inbound Flow):** Запрос поступает из сети. Он попадает в первое middleware. Этот компонент может выполнить логику *до* передачи управления дальше (например, начать замер времени или проверить IP-адрес).
2. **Передача управления:** Middleware вызывает делегат next(). Это действие передает запрос во *внутренний* слой (следующее middleware).
3. **Погружение:** Процесс повторяется, пока не будет достигнут терминальный компонент (Endpoint), который генерирует ответ (например, JSON с данными).
4. **Выход (Outbound Flow):** После того как внутренний компонент завершил работу (вернул Task), управление возвращается к предыдущему middleware. Теперь оно может выполнить логику *после* (например, остановить таймер и записать лог, или перехватить исключение, возникшее внутри).10

### 3.2. Техническая реализация: Делегаты

На низком уровне эта структура реализуется через цепочку делегатов. Ключевым типом данных является RequestDelegate.

**Определение:**

C#

public delegate Task RequestDelegate(HttpContext context);

Это функция, которая принимает контекст запроса (HttpContext) и возвращает асинхронную задачу (Task). Каждое middleware, по сути, является фабрикой или оберткой вокруг такого делегата.11

При построении приложения (app.Build()) фреймворк берет все зарегистрированные компоненты и "сшивает" их. Если у нас есть Middleware A, B и C, то:

* Middleware C (внутреннее) оборачивается в Middleware B.
* Middleware B (содержащее C) оборачивается в Middleware A.
* Middleware A становится точкой входа в приложение (Server.RequestDelegate).12

Это объясняет, почему порядок регистрации в Program.cs имеет критическое значение. Компонент, зарегистрированный первым, выполняется первым при входе запроса и последним при выходе ответа (Stack Unwinding).3

### 3.3. Аналогия с Unity: Coroutines и Wrappers

Для Unity-разработчика ближайшей аналогией может служить система корутин (Coroutines) или вложенных вызовов. Представьте, что у вас есть система обработки урона:

C#

// Аналогия в C# (Unity style)  
public class DamageSystem : MonoBehaviour {  
   
 // Middleware 1: Shield (Щит)  
 public IEnumerator ApplyDamage(int damage) {  
 Debug.Log("Shield Check Started"); // Inbound logic  
   
 int damageAfterShield = CalculateShield(damage);  
   
 // Передача управления следующему слою (аналог await next())  
 yield return StartCoroutine(ArmorProcessing(damageAfterShield));  
   
 Debug.Log("Shield Check Finished"); // Outbound logic  
 }  
  
 // Middleware 2: Armor (Броня)  
 public IEnumerator ArmorProcessing(int damage) {  
 Debug.Log("Armor Check Started");  
 int finalDamage = CalculateArmor(damage);  
   
 // Terminal Middleware: Health (Здоровье)  
 ApplyHealthReduction(finalDamage);  
   
 yield return null; // Возврат управления наверх  
 }  
}

В ASP.NET Core вместо yield return используется await next(context). Пока next не завершится, внешнее middleware "ждет" и держит свой контекст выполнения (локальные переменные) в памяти (в State Machine, генерируемой компилятором).1

## 4. HttpContext: Объектная модель состояния запроса

Если в Unity мы привыкли обращаться к глобальным статическим классам вроде Time, Input или синглтонам GameManager.Instance, то в ASP.NET Core вся информация о текущей операции инкапсулирована в объект HttpContext.

### 4.1. Структура HttpContext

HttpContext — это контейнер, который создается веб-сервером (Kestrel) в момент получения байтов из сокета и уничтожается сразу после отправки ответа. Он содержит:

* **Request:** Свойства входящего запроса (Headers, Body, QueryString, Path).6
* **Response:** Объект для формирования ответа (StatusCode, Headers, BodyStream).
* **User:** Информация об аутентифицированном пользователе (ClaimsPrincipal).
* **Items:** Словарь IDictionary<object, object> для передачи данных между middleware в рамках одного запроса.6
* **RequestServices:** Локальный DI-контейнер (Scope), из которого можно получать сервисы, живущие только в рамках этого запроса.

### 4.2. Потокобезопасность (Thread Safety) — Критическая зона

Важнейшее отличие от Unity: HttpContext НЕ является потокобезопасным.6

В Unity метод Update гарантированно выполняется в одном потоке. В ASP.NET Core, хотя каждый отдельный запрос обрабатывается последовательно, сервер обрабатывает тысячи запросов параллельно.

Типичная ошибка Unity-разработчика:

Попытка сохранить HttpContext в статическое поле или в Singleton-сервис.

C#

// ОПАСНО: Никогда так не делайте!  
public static class ContextHolder {  
 public static HttpContext Current; // Это поле будет перезаписано другим запросом!  
}

Если два запроса придут одновременно, второй перезапишет переменную Current, и первый запрос начнет отправлять данные не тому пользователю. Это называется "Cross-talk" и является катастрофической уязвимостью. В ASP.NET Core доступ к контексту должен осуществляться только через аргументы методов middleware или контроллеров.13

### 4.3. Коллекция Items

Для передачи данных между middleware (например, результат валидации API-ключа, который нужно использовать позже в контроллере) используется коллекция HttpContext.Items. Это аналог "Blackboard" паттерна в AI Unity или просто передачи параметров, но привязанный к жизненному циклу запроса. Данные в Items исчезают вместе с контекстом после завершения запроса.14

## 5. Механизмы регистрации: Use, Run, Map

В файле Program.cs происходит конфигурация конвейера. Различают три основных метода расширения IApplicationBuilder, каждый из которых имеет семантическое значение.15

### 5.1. app.Use() — Звено цепи

Метод Use добавляет middleware, которое предполагает продолжение цепочки. Оно принимает два параметра: context и next.

* **Назначение:** Выполнение работы до и после следующего компонента.
* **Особенность:** Обязано вызвать await next() (или next.Invoke()), чтобы передать управление дальше. Если оно этого не сделает, цепочка прервется (Short-circuiting), и следующие компоненты не выполнятся.12

C#

app.Use(async (context, next) => {  
 // Входная логика  
 await next(); // Передача эстафеты  
 // Выходная логика  
});

### 5.2. app.Run() — Тупик (Terminal Middleware)

Метод Run добавляет терминальное middleware. Оно принимает только context.

* **Назначение:** Окончательная обработка запроса.
* **Особенность:** У него нет ссылки на next. Все, что добавлено в конвейер *после* app.Run, никогда не будет выполнено.3

**Пример:**

C#

app.Run(async context => {  
 await context.Response.WriteAsync("Hello World");  
});  
// Любой код middleware ниже этой строки будет проигнорирован.

### 5.3. app.Map() — Ветвление

Метод Map позволяет создать ответвление в конвейере на основе пути URL. Это условная логика "If".

* **Назначение:** Запуск специфического набора middleware для определенного префикса пути.
* **Аналогия Unity:** Переключение State Machine. Если состояние "Game", выполняем один Update, если "Menu" — другой.17

C#

app.Map("/api", apiApp => {  
 // Этот middleware выполнится только для запросов, начинающихся с /api  
 apiApp.UseAuthorization();   
 apiApp.Run(...);  
});

## 6. Жизненный цикл Middleware: Ловушки Dependency Injection

В Unity мы привыкли к Instantiate и Destroy. В ASP.NET Core middleware имеют свои нюансы создания, которые тесно связаны с DI (Dependency Injection).1

### 6.1. Convention-based Middleware (Стандартные)

Определяются как обычный C# класс.

* **Конструктор:** Принимает RequestDelegate next.
* **Метод:** InvokeAsync(HttpContext context).
* **Жизненный цикл:** Создается **один раз** при старте приложения (Singleton).

Проблема Captive Dependency:

Так как класс middleware создается один раз, вы не можете внедрять Scoped-сервисы (например, DbContext) через конструктор. Если вы это сделаете, вы "захватите" сервис, предназначенный для одного запроса, и заставите его жить вечно. Это приведет к утечкам памяти и ошибкам конкурентного доступа к БД.4

**Решение:** Внедрять Scoped-зависимости в метод InvokeAsync, а не в конструктор. ASP.NET Core умеет разрешать зависимости метода (Method Injection) в момент вызова.18

### 6.2. Factory-based Middleware (IMiddleware)

Реализуют интерфейс IMiddleware.

* **Метод:** InvokeAsync(HttpContext context, RequestDelegate next).
* **Жизненный цикл:** Регистрируются в DI-контейнере (обычно как Transient или Scoped). Создаются **заново для каждого запроса**.18

**Преимущество:** Позволяют безопасно внедрять Scoped-сервисы прямо в конструктор, что делает код чище и понятнее для тех, кто привык к Constructor Injection. Это более безопасный подход для новичков, переходящих с Unity, так как поведение ближе к ожидаемому "созданию объекта для обработки".19

## 7. Новые возможности.NET 8: Short-Circuiting

.NET 8 привнес важную оптимизацию в работу конвейера, называемую "Short-Circuiting" (Короткое замыкание).20

### 7.1. Проблема производительности конвейера

Даже если middleware просто передает управление дальше (await next()), это стоит ресурсов (аллокация Task, переключение контекста). Для некоторых запросов, таких как robots.txt или favicon.ico, прохождение через тяжелые middleware (аутентификация, логирование, сессии) избыточно.21

### 7.2. Решение:.ShortCircuit()

Теперь можно пометить определенные эндпоинты как "Short Circuit". Это говорит системе маршрутизации (Routing Middleware): "Если ты нашла совпадение для этого маршрута, выполни его логику немедленно и закончи обработку. Не вызывай остальные middleware в цепочке".4

C#

app.MapGet("/health", () => "OK").ShortCircuit();

Это аналогично оптимизации **Culling** в Unity: если объект не виден камерой, мы не тратим ресурсы на его отрисовку. Здесь, если запрос тривиален, мы не тратим ресурсы на запуск всего конвейера безопасности и логирования.3

## 8. Обработка исключений: Глобальный щит

В Unity необработанное исключение в Update обычно логируется в консоль, и скрипт продолжает работу (или отключается). Приложение не падает целиком. В Backend, если исключение не поймать, поток аварийно завершится, и клиент получит разрыв соединения или пустой ответ.

Поэтому первым (внешним) слоем "матрешки" всегда должно быть Exception Handling Middleware.

Оно оборачивает вызов next() в блок try-catch.

* Если любой компонент внутри (контроллер, БД, логика) выбросит исключение, оно "всплывет" наверх по стеку вызовов.
* Exception Middleware поймает его, залогирует и сформирует корректный HTTP-ответ (например, JSON с кодом 500 и описанием ошибки, если это среда разработки).22

Это реализация паттерна **Resilience**. Система самовосстанавливается после сбоя в обработке конкретного запроса, оставаясь доступной для других пользователей.7

## 9. Заключение: От игрового цикла к потоку данных

Первый день шестой недели — это не просто изучение новых классов. Это смена ментальной прошивки.

1. Мы отказываемся от **Game Loop** в пользу **Pipeline**.
2. Мы меняем **Global State** на **Request Context**.
3. Мы переходим от **Sequence** (последовательного вызова Update) к **Recursion/Nesting** (вложенности делегатов).

Понимание того, что каждый запрос — это отдельная жизнь приложения, рождающаяся и умирающая за миллисекунды, является фундаментом для построения масштабируемых систем. Инструменты, такие как Use, Run, Map и .ShortCircuit(), дают архитектору полный контроль над этим жизненным циклом, позволяя оптимизировать систему так же тонко, как Unity-разработчик оптимизирует рендеринг кадра.

### Приложение А: Терминологический словарь (Unity ->.NET Backend)

| **Термин Unity** | **Аналог в ASP.NET Core** | **Комментарий** |
| --- | --- | --- |
| Update() | InvokeAsync() | Основной метод выполнения логики компонента. |
| GameObject | Middleware | Компонент, выполняющий часть работы. |
| Time.deltaTime | Stopwatch | Время замеряется явно для мониторинга, а не для физики. |
| DontDestroyOnLoad | Singleton Service | Объект, живущий все время работы приложения. |
| Instantiate | Transient Service | Объект, создаваемый по требованию (DI). |
| Culling | Short-Circuiting | Пропуск ненужной обработки для оптимизации. |
| Coroutine | async/await | Асинхронная модель выполнения без блокировки потока. |

Этот документ служит теоретическим фундаментом для практических заданий Недели 6 и должен быть использован как справочный материал при написании кода "Инспектора Конвейера".

#### Источники

1. План обучения ASP.NET Core: Неделя 6
2. Advanced programming and code architecture | Unity, дата последнего обращения: декабря 5, 2025, <https://unity.com/how-to/advanced-programming-and-code-architecture>
3. ASP.NET Core Middleware | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-10.0>
4. 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>
5. Functional Unity Architecture: A Developer's Guide | by Bruno Mikoski | Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@bada/functional-unity-architecture-a-developers-guide-359e5111c5c2>
6. Use HttpContext in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/use-http-context?view=aspnetcore-10.0>
7. Mastering Advanced ASP.NET Core Middleware: A Complete Guide with Real-World Examples | by Chauhan Shubham - Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@chauhanshubham19765/mastering-advanced-asp-net-core-middleware-a-complete-guide-with-real-world-examples-f3bb2ae8bb02>
8. Middleware - Wolverine, дата последнего обращения: декабря 5, 2025, <https://wolverinefx.net/guide/handlers/middleware>
9. How we did authorization in FubuMVC, and what I'd do differently today, дата последнего обращения: декабря 5, 2025, <https://jeremydmiller.com/2016/04/19/how-we-did-authorization-in-fubumvc-and-what-id-do-differently-today/>
10. Understanding Middleware in ASP.NET Core - Poorna Soysa, дата последнего обращения: декабря 5, 2025, <https://poornasoysa.tech/understanding-middleware-in-aspnet-core/>
11. Deep Dive: How is the ASP.NET Core Middleware Pipeline Built? - Steve Gordon, дата последнего обращения: декабря 5, 2025, <https://www.stevejgordon.co.uk/how-is-the-asp-net-core-middleware-pipeline-built>
12. 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>
13. 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>
14. HttpContext Class and its Thread Safety - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/7569578/httpcontext-class-and-its-thread-safety>
15. What are the difference using app.Run and app.UseEndpoints in ASP.NET Core?, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/58432973/what-are-the-difference-using-app-run-and-app-useendpoints-in-asp-net-core>
16. OWIN app.use vs app.run vs app.map - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/35559763/owin-app-use-vs-app-run-vs-app-map>
17. Building powerful pipelines with custom middleware in ASP.NET Core - Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@denmaklucky/building-powerful-pipelines-with-custom-middleware-in-asp-net-core-b4ff8e626f49>
18. 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>
19. Explain the difference between app.Run and app.Use in ASP.NET Core. - MindStick, дата последнего обращения: декабря 5, 2025, <https://www.mindstick.com/forum/157450/explain-the-difference-between-app-run-and-app-use-in-asp-dot-net-core>
20. Route Short Circuit middleware in ASP.NET Core - Amit Prakash, дата последнего обращения: декабря 5, 2025, <https://www.amitprakash.me/blog/short-circuit-middleware>
21. 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>
22. 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>
23. ASP.NET Core Error Handling: Master Middleware, Logging & Exception Strategies (Part-20 of 40) - C# Corner, дата последнего обращения: декабря 5, 2025, <https://www.c-sharpcorner.com/article/asp-net-core-error-handling-master-middleware-logging-exception-strategies/>