# Глубокий анализ архитектуры ASP.NET Core для Unity-разработчиков: Теория Второго Дня Седьмой Недели

## Введение: Эволюция Парадигмы от Игрового Цикла к Распределенным Вычислениям

Переход от разработки интерактивных приложений реального времени (Unity/GameDev) к созданию корпоративных веб-сервисов (ASP.NET Core Backend) представляет собой фундаментальный когнитивный сдвиг. Это не просто смена языка или фреймворка; это изменение самой природы взаимодействия программного обеспечения со временем, состоянием и пользователем. В контексте учебного плана седьмой недели, посвященной проектированию REST API для сервиса календаря 1, второй день обучения является критическим этапом, на котором закладывается архитектурный фундамент понимания того, как именно сервер обрабатывает входящие сигналы.

В экосистеме Unity разработчик оперирует в рамках детерминированного **Игрового Цикла (Game Loop)**. Это бесконечный цикл while(true), который управляет обновлением физики, логики и рендеринга, как правило, синхронизированный с частотой обновления экрана. Объекты MonoBehaviour в этой среде обладают длительным жизненным циклом (Long-Lived Objects): они создаются при загрузке сцены и часто живут до ее выгрузки или завершения сессии.2 Состояние мира (позиции врагов, инвентарь, здоровье) мутирует в оперативной памяти и доступно глобально или через прямые ссылки. Потоковая модель здесь преимущественно однопоточная (Main Thread), где асинхронность эмулируется через Корутины (Coroutines), которые, по сути, являются итераторами, возвращающими управление главному циклу.4

В противоположность этому, backend-разработка на платформе ASP.NET Core (.NET 9) оперирует в парадигме **Request-Response (Запрос-Ответ)**. Здесь отсутствует понятие непрерывного кадра. Единицей работы является изолированный HTTP-запрос, который рождается в момент поступления сетевого пакета и умирает после отправки ответа. Архитектура системы является *Stateless* (без сохранения состояния): сервер не должен полагаться на то, что память процесса сохранит данные между двумя последовательными запросами одного пользователя, так как в облачной среде эти запросы могут быть обработаны разными экземплярами сервиса.1

Цель данного отчета — предоставить исчерпывающе детальный теоретический фундамент второго дня обучения, покрывающий анатомию контроллеров, механизмы маршрутизации, управление зависимостями (Dependency Injection) и асинхронную модель выполнения, с постоянным проведением параллелей и контрастов с привычными паттернами Unity-разработки.

## Часть I: Архитектура Конвейера Обработки (Request Pipeline) и Middleware

Прежде чем управление будет передано прикладному коду контроллера, HTTP-запрос проходит через сложную, строго упорядоченную систему компонентов, называемую конвейером обработки (Request Pipeline). Понимание этого механизма критически важно, так как именно здесь определяются правила игры для входящего трафика еще до того, как он достигнет бизнес-логики.

### 1.1. Middleware как Эволюция Событийной Модели

В Unity порядок выполнения скриптов определяется фазами кадра: сначала FixedUpdate для физики, затем Update для логики, затем LateUpdate для камеры и, наконец, рендеринг.3 Это жестко заданная последовательность событий. В ASP.NET Core аналогом этой последовательности выступает цепочка Middleware (промежуточного ПО).

Middleware — это программные модули, выстроенные в цепочку типа "Chain of Responsibility" (Цепочка обязанностей) или "Russian Doll" (Матрешка). Каждый компонент получает HttpContext (контекст запроса) и делегат RequestDelegate next, указывающий на следующий компонент в цепи.5 Компонент имеет три варианта действий:

1. **Обработать запрос и передать дальше:** Выполнить логику (например, начать замер времени), вызвать await next(context), и после возвращения управления выполнить завершающую логику (остановить таймер, залогировать результат).
2. **Модифицировать запрос:** Изменить заголовки, считать тело запроса или подменить пользователя.
3. **Прервать цепь (Short-circuit):** Не вызывать next, а сразу записать ответ в context.Response. Это происходит, например, если пользователь не авторизован, ресурс найден в кэше или сработал лимит запросов (Rate Limiting).6

В отличие от событий Unity, которые часто независимы друг от друга (ошибка в одном Update может не остановить другие), Middleware строго зависимы. Если Middleware аутентификации отклоняет запрос, Middleware маршрутизации даже не узнает о его существовании. Это создает мощный механизм защиты, называемый "Защитным периметром".

### 1.2. Порядок Регистрации и Его Критичность

В.NET 9 конфигурация конвейера происходит в файле Program.cs. Порядок вызова методов app.Use... определяет порядок исполнения Middleware. Это фундаментальное отличие от Unity, где порядок выполнения скриптов MonoBehaviour часто недетерминирован или настраивается через "Script Execution Order" в настройках проекта.

Типичная конфигурация конвейера для REST API выглядит следующим образом:

| **Компонент Middleware** | **Описание и Аналогия с Unity** | **Порядок и Значение** |
| --- | --- | --- |
| **ExceptionHandler** | Глобальная ловушка ошибок. Аналог try-catch вокруг всего игрового цикла. | **Первый в цепи.** Гарантирует, что любое исключение ниже по стеку будет перехвачено и преобразовано в корректный JSON-ответ (RFC 7807), а не приведет к падению сервера.8 |
| **HSTS / HttpsRedirection** | Принуждение к безопасности. Аналог проверки лицензии или целостности клиента. | Должен быть в начале, чтобы небезопасные запросы (HTTP) отклонялись до обработки логики. |
| **Routing** | Определение адресата. Аналог Raycasting'а для определения, в какой объект попал клик мыши. | Определяет, какой Endpoint (Контроллер + Метод) соответствует URL. Важно: само действие еще не выполняется, только выбирается.9 |
| **Cors** | Cross-Origin Resource Sharing. Аналог сетевых политик доступа. | Должен стоять после Routing, так как политики могут зависеть от выбранного маршрута, но до Auth. |
| **Authentication** | "Кто ты?". Проверка удостоверения личности (Token, Cookie). | Проверяет валидность токена. Устанавливает User в контексте (HttpContext.User). |
| **Authorization** | "Можешь ли ты это сделать?". Проверка прав доступа. | Проверяет, имеет ли аутентифицированный пользователь доступ к выбранному Endpoint'у. Требует, чтобы Routing уже отработал. |
| **MapControllers (Endpoints)** | Финальная точка. Передача управления методу контроллера. | Последний этап. Если запрос дошел сюда, значит он прошел все проверки безопасности. |

Ошибка в порядке (например, UseAuthorization перед UseRouting) приведет к тому, что система авторизации не сможет понять, к какому ресурсу запрашивается доступ, так как маршрут еще не выбран. Это подчеркивает процедурную природу инициализации backend-приложения в отличие от декларативной природы сцены Unity.

### 1.3. Endpoint Routing: Разделение Решений и Исполнения

Современный ASP.NET Core использует концепцию *Endpoint Routing*. Это означает, что процесс принятия решения "какой код запустить" (UseRouting) отделен от самого запуска кода (UseEndpoints). Это позволяет middleware, находящимся *между* этими двумя этапами (например, Authorization), получать доступ к метаданным выбранного endpoint'а.

Например, контроллер может быть помечен атрибутом ``.

1. **UseRouting:** Анализирует URL /api/users/delete/5 и понимает, что это метод DeleteUser в UsersController. Он прикрепляет объект Endpoint с метаданными (включая атрибут роли) к контексту запроса.
2. **UseAuthorization:** Смотрит в контекст, видит выбранный Endpoint, читает требование "Roles = Admin", проверяет текущего пользователя. Если прав нет — возвращает 403 Forbidden и прерывает цепь.
3. **UseEndpoints:** Если цепь не прервана, запускает метод контроллера.

В Unity нет прямого аналога такому разделению фаз. Ближайшая аналогия — это система событий UI, где сначала происходит Raycast (определение цели), а затем, возможно, фильтрация события (например, если UI перекрыт другим окном), и только потом вызов OnClick.10

## Часть II: Анатомия и Жизненный Цикл Контроллера в.NET 9

Контроллер в архитектуре REST API — это точка входа в бизнес-логику приложения. Однако его природа кардинально отличается от компонентов в игровых движках.

### 2.1. Контроллер vs MonoBehaviour: Концептуальная Пропасть

В Unity класс, наследуемый от MonoBehaviour, является компонентом, живущим на объекте сцены. Он имеет состояние (поля класса), которое сохраняется между кадрами.

В ASP.NET Core контроллер — это Transient (в терминологии времени жизни) объект. Он создается исключительно для обработки одного конкретного запроса и уничтожается немедленно после отправки ответа.11

Это означает, что хранить состояние в полях контроллера бессмысленно.

Неправильно (Unity-style):

C#

public class CounterController : ControllerBase {  
 private int \_count = 0; // Всегда будет сбрасываться в 0  
 [HttpPost("increment")]  
 public IActionResult Increment() {  
 \_count++;   
 return Ok(\_count);  
 }  
}

При каждом запросе Increment будет создаваться *новый* экземпляр CounterController, и \_count будет инициализироваться нулем. Для сохранения состояния необходимо использовать внешние хранилища (База данных, Distributed Cache) или Singleton-сервисы, о чем будет сказано в разделе про DI.

### 2.2. Наследование: Controller vs ControllerBase

Платформа.NET предоставляет два базовых класса:

1. **Controller:** Наследуется от ControllerBase и добавляет поддержку View() (Razor Views). Это необходимо для MVC-приложений, рендерящих HTML-страницы на сервере. Это "тяжелый" класс, тянущий за собой механизмы View Engine.12
2. **ControllerBase:** Облегченный класс, предоставляющий только инструменты для работы с HTTP API (доступ к Request, Response, User, методы-хелперы для статус-кодов типа Ok(), BadRequest()).

Для задачи построения REST API (Calendar Service) **единственно верным выбором** является наследование от ControllerBase.13 Это минимизирует накладные расходы и соответствует принципу единственной ответственности (SRP). Использование полного Controller в API проекте является признаком недостаточной квалификации или использования устаревших шаблонов.

### 2.3. Атрибут [ApiController] и Его Магия

Декорирование класса атрибутом [ApiController] активирует набор конвенций, упрощающих разработку API (.NET Core 2.1+):

* **Attribute Routing Requirement:** Запрещает использование традиционной маршрутизации через таблицу маршрутов (convention-based routing), принуждая к явному определению маршрутов атрибутами. Это делает API более предсказуемым.
* **Automatic HTTP 400 Responses:** Автоматически проверяет ModelState.IsValid. Если клиент прислал JSON, не соответствующий DTO (например, пропустил обязательное поле), контроллер *даже не начнет выполнять метод*, а сразу вернет стандартизированный ответ 400 Bad Request с перечислением ошибок валидации.14 В Unity аналогом была бы автоматическая проверка входных данных в инспекторе до запуска игры.
* **Binding Source Inference:** Автоматически понимает, что сложные объекты нужно искать в теле запроса (), а простые типы — в строке запроса или пути (`[FromQuery]`, ).

### 2.4. Механизм Инстанцирования: От IControllerFactory до Активатора

Процесс создания контроллера скрыт от глаз, но его понимание важно для отладки DI.

Когда маршрутизатор выбирает контроллер:

1. Вызывается IControllerFactory.
2. Она использует IControllerActivator для создания экземпляра.15
3. Активатор анализирует конструктор контроллера и запрашивает все необходимые зависимости у DI-контейнера (IServiceProvider).
4. Если какая-либо зависимость не может быть разрешена (не зарегистрирована в Program.cs), будет выброшено исключение InvalidOperationException с детальным описанием ошибки графа зависимостей.

В.NET 9 стандартом становится использование **Primary Constructors** (Первичных конструкторов), введенных в C# 12. Это позволяет объявлять зависимости прямо в сигнатуре класса, делая код лаконичным и декларативным.17

*Пример (.NET 9 Style):*

C#

[ApiController]  
")]  
public class EventsController(ICalendarService service, ILogger<EventsController> logger) : ControllerBase  
{  
 // service и logger доступны как поля класса  
 [HttpGet("{id:guid}")]  
 public async Task<IActionResult> Get(Guid id) => Ok(await service.GetEventAsync(id));  
}

## Часть III: Система Маршрутизации (Attribute Routing)

Маршрутизация — это "нервная система" API, отвечающая за доставку сигналов (запросов) к правильным органам (методам). В отличие от RPC-подходов, часто используемых в сетевом коде игр (где есть один сокет и switch-case по OpCode), REST API требует уникальной адресации для каждого ресурса.

### 3.1. Шаблоны и Токены

Атрибутная маршрутизация использует строковые шаблоны. Токен [controller] в атрибуте ")] автоматически заменяется на имя класса без суффикса "Controller". Это удобно, но требует дисциплины в именовании классов. Для версии API рекомендуется жестко прописывать префикс: ")].1

### 3.2. Ограничения Маршрутов (Route Constraints)

Ограничения позволяют дифференцировать маршруты не только по структуре, но и по типу данных. Это предотвращает попадание некорректных данных в контроллер и позволяет перегружать URL-адреса.

В Unity мы часто проверяем теги объектов при коллизии (if (other.CompareTag("Player"))). В ASP.NET Core эту роль выполняют Constraints.

Они записываются через двоеточие внутри параметра маршрута: {parameter:constraint}.

**Таблица ключевых ограничений (.NET 9):**

| **Ограничение** | **Описание и Пример** | **Применение в Calendar Service** |
| --- | --- | --- |
| guid | Соответствует формату GUID (32 цифры, дефисы). {id:guid} | Идентификация событий: GET /events/{id:guid}. Исключает конфликты с текстовыми командами. |
| int / long | Целые числа. {id:int} | Пагинация или идентификация в legacy-системах. |
| minlength(n) | Строка минимум n символов. {q:minlength(3)} | Поиск: запрет слишком коротких поисковых запросов. |
| datetime | Формат даты. {date:datetime} | Фильтрация по дням: GET /events/{date:datetime}.19 |
| alpha | Только буквы. {status:alpha} | Фильтрация по статусу, если статус передается текстом. |
| range(min, max) | Число в диапазоне. {month:range(1,12)} | Валидация номера месяца прямо в URL. |

Ограничения повышают производительность, так как отсев неподходящих запросов происходит на ранней стадии (Routing Middleware), не затрачивая ресурсы на создание контроллера и привязку модели.

### 3.3. Разрешение Неоднозначности (Ambiguity Resolution)

Если входящий URL соответствует нескольким маршрутам, система должна выбрать лучший. Алгоритм выбора в.NET Core работает на основе приоритетов (Precedence):

1. **Литералы** (статический текст) имеют высший приоритет. Маршрут events/today выиграет у events/{id}, если пришел запрос /events/today.
2. **Ограничения** повышают приоритет. {id:guid} специфичнее, чем просто {id}.
3. **Параметры** имеют низший приоритет.

Если система не может выбрать победителя (например, два идентичных шаблона), выбрасывается AmbiguousMatchException.20 Это частая ошибка при копировании методов контроллера ("Copy-Paste Programming").

## Часть IV: Внедрение Зависимостей (Dependency Injection)

Внедрение зависимостей (DI) в ASP.NET Core — это не просто паттерн, это кровеносная система приложения. Для разработчика Unity, привыкшего к паттерну Singleton (статический доступ через GameManager.Instance) или GetComponent, модель DI может показаться избыточно сложной, но она необходима для тестируемости и масштабируемости.21

### 4.1. Inversion of Control (IoC) Контейнер

Вместо того чтобы класс сам создавал свои зависимости (new DatabaseService()), он запрашивает их через конструктор. Контейнер DI берет на себя ответственность за создание этих объектов и управление их жизненным циклом. Это разрывает жесткую связность (Coupling).

### 4.2. Жизненные Циклы Сервисов (Service Lifetimes)

Понимание трех типов времени жизни сервисов является фундаментальным навыком backend-инженера. Ошибка здесь может привести к утечкам памяти или состоянию гонки (Race Conditions).23

| **Тип Жизни (Lifetime)** | **Механизм Работы** | **Аналогия с Unity** | **Применение в Calendar Service** |
| --- | --- | --- | --- |
| **Transient** (Временный) | Создается **новый экземпляр** каждый раз, когда запрашивается. Даже в рамках одного запроса, если сервис внедрен в два места, это будут два разных объекта. | Instantiate(prefab). Объект создается, используется и забывается. Эффект частиц, звук выстрела. | Легковесные конвертеры, валидаторы без состояния. |
| **Scoped** (В рамках запроса) | Создается **один раз на HTTP-запрос**. Все компоненты, участвующие в обработке этого запроса, делят один и тот же экземпляр. Уничтожается в конце запроса. | Сцена. Объект существует, пока загружена сцена, и доступен всем в этой сцене. При смене сцены (запроса) создается заново. | DbContext (База данных), сервисы бизнес-логики (ICalendarService), Unit of Work. Основной тип для веба. |
| **Singleton** (Одиночка) | Создается **один раз** при первом обращении и живет до остановки приложения. Один экземпляр обслуживает миллионы запросов от тысяч пользователей. | DontDestroyOnLoad, статический класс GameManager. Глобальное состояние. | Кэш (IMemoryCache), конфигурация, сервисы-одиночки. **Опасно:** требует потокобезопасности! |

### 4.3. Опасность Captive Dependency (Захваченная Зависимость)

Это архитектурная ловушка, специфичная для DI контейнеров. Она возникает, когда сервис с *длительным* временем жизни (например, Singleton) зависит от сервиса с *коротким* временем жизни (например, Scoped).26

Пример катастрофы:

Если вы внедрите Scoped DbContext в Singleton CacheService, то DbContext будет создан один раз и "захвачен" синглтоном навсегда.

1. Он не будет уничтожаться (Disposed), что приведет к утечке памяти.
2. Он будет использовать одно и то же подключение к БД для всех запросов, что недопустимо для Entity Framework (он не потокобезопасен).
3. Данные в нем устареют.

В.NET Core при запуске в режиме Development работает проверка ValidateScopes, которая выбросит исключение при обнаружении такой ситуации. В Unity таких проверок нет, и подобные ошибки часто приводят к трудноуловимым багам.

## Часть V: Асинхронность и Многопоточность (Async/Await)

Backend-разработка требует высочайшей эффективности использования ресурсов. В Unity асинхронность часто используется для удобства (подождать 2 секунды), в ASP.NET — для выживания под нагрузкой.

### 5.1. SynchronizationContext: Миф об "Одном Потоке"

В Unity await (или yield) возвращает выполнение в **Main Thread** (основной поток). Это гарантирует безопасность доступа к API движка (transform.position). Это обеспечивается кастомным UnitySynchronizationContext.4

В ASP.NET Core отсутствует SynchronizationContext.28

Это радикальное отличие. Когда код выполняет await database.GetDataAsync():

1. Текущий поток освобождается и возвращается в **ThreadPool** (Пул потоков). Он может немедленно начать обрабатывать *другой* запрос.
2. Когда база данных возвращает данные, продолжение метода (код после await) захватывается *любым* свободным потоком из пула.
3. Это может быть тот же поток, а может быть совершенно другой.

**Следствия:**

* Нельзя использовать ThreadStatic или полагаться на ManagedThreadId.
* Нельзя использовать блокирующие вызовы .Result или .Wait(). В старых версиях ASP.NET это приводило к Deadlock. В ASP.NET Core это приводит к **Thread Starvation** (истощению пула). Если все потоки заблокированы ожиданием, сервер перестает отвечать на новые запросы.30

### 5.2. CancellationToken: Уважение к Ресурсам

В веб-среде клиент может отменить запрос (закрыть вкладку, нажать "Stop"). Если сервер продолжает выполнять тяжелую работу (запрос к БД, расчеты), он тратит ресурсы впустую.

Unity-разработчики редко сталкиваются с этим, так как игровые процессы обычно должны завершаться логически.

В ASP.NET Core методы контроллера должны принимать CancellationToken и передавать его во все асинхронные методы.31

C#

[HttpGet("{id}")]  
public async Task<IActionResult> GetEvent(Guid id, CancellationToken ct)  
{  
 // Если клиент отключился, запрос к БД будет отменен  
 var evt = await \_service.GetByIdAsync(id, ct);  
 return Ok(evt);  
}

### 5.3. Сравнение с Coroutines

Корутины Unity — это механизм кооперативной многозадачности на одном потоке. async/await в.NET — это механизм работы с пулом потоков и асинхронным вводом-выводом (I/O).31 Использование Task.Run внутри контроллера (для CPU-bound задач) следует делать с осторожностью, чтобы не исчерпать пул потоков, обслуживающий входящие HTTP-запросы.

## Часть VI: Minimal API vs Controllers в Эру.NET 9

В.NET 9 активно продвигается подход **Minimal API** — сверхлегкий синтаксис для создания эндпоинтов без классов контроллеров. Однако учебный план 1 предписывает использование классических контроллеров. Почему?

### 6.1. Производительность и Архитектура

Minimal API обеспечивают чуть более высокую производительность (меньше аллокаций памяти, быстрее старт приложения) и идеально подходят для микросервисов с малой кодовой базой.13 Они позволяют писать код в функциональном стиле:

app.MapGet("/events", async (ICalendarService svc) => await svc.GetAll());

### 6.2. Педагогическая Ценность Контроллеров

Для перехода с Unity использование контроллеров оправдано следующими факторами:

1. **Структурность:** Контроллеры (классы) обеспечивают жесткую структуру кода, аналогичную организации скриптов в проекте Unity. Minimal API может превратить Program.cs в "God Object", если не применять продвинутые техники структурирования.
2. **Явность:** Контроллеры используют явное внедрение зависимостей через конструктор и явные атрибуты маршрутизации, что облегчает понимание потока данных для новичка.
3. **Инструментарий:** Поддержка фильтров действий (Action Filters), валидации моделей и структурирования ответов в MVC более развита "из коробки".33

## Заключение

Второй день седьмой недели обучения знаменует собой переход от написания кода, управляющего кадром, к написанию кода, управляющего потоком данных. Ключевые выводы для реализации сервиса календаря:

1. **Контроллер — это диспетчер, а не исполнитель.** Он живет миллисекунды. Логика должна быть в Сервисах.
2. **Scoped — ваш лучший друг.** Для обработки запросов используйте Scoped-зависимости. Избегайте глобального состояния (Singleton) для изменяемых данных.
3. **Асинхронность — это закон.** Любая операция ввода-вывода (БД, Файл, Сеть) должна быть асинхронной (async/await) и поддерживать отмену (CancellationToken).
4. **Маршруты — это контракт.** Используйте ограничения маршрутов ({id:guid}) для строгой типизации URL.

Усвоение этих принципов позволит создать масштабируемую, надежную систему, соответствующую промышленным стандартам Enterprise-разработки.

#### Источники

1. План обучения REST API для бэкенда неделя 7
2. What is MonoBehaviour in Unity 3D? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/41630986/what-is-monobehaviour-in-unity-3d>
3. Event function execution order - Unity - Manual, дата последнего обращения: декабря 5, 2025, <https://docs.unity3d.com/6000.2/Documentation/Manual/execution-order.html>
4. Coroutines OR Async/await? : r/unity - Reddit, дата последнего обращения: декабря 5, 2025, <https://www.reddit.com/r/unity/comments/zr39d4/coroutines_or_asyncawait/>
5. 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>
6. Understanding Middleware and Filters in .NET Core - FullStack Labs, дата последнего обращения: декабря 5, 2025, <https://www.fullstack.com/labs/resources/blog/how-to-use-middleware-and-filters-in-a-net-core-pipeline>
7. A Complete Beginner's Guide to ASP.NET Core .NET 8 Middleware | by Ravi Patel | Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@ravipatel.it/a-complete-beginners-guide-to-asp-net-core-net-8-middleware-1e35c0eab444>
8. 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>
9. ASP.NET Core MVC Request Life Cycle, дата последнего обращения: декабря 5, 2025, <https://www.c-sharpcorner.com/article/asp-net-core-mvc-request-life-cycle/>
10. Unity Events vs C# Actions : r/Unity3D - Reddit, дата последнего обращения: декабря 5, 2025, <https://www.reddit.com/r/Unity3D/comments/1jdtjuz/unity_events_vs_c_actions/>
11. ASP.NET MVC Controller Lifecycle - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/1763775/asp-net-mvc-controller-lifecycle>
12. Overview of ASP.NET Core MVC - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-10.0>
13. APIs overview | Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/apis?view=aspnetcore-10.0>
14. Routing to controller actions in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-10.0>
15. IControllerFactory vs IControllerActivator asp.net core - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/56478849/icontrollerfactory-vs-icontrolleractivator-asp-net-core>
16. Inside ASP.NET MVC: Instantiation of Controller, дата последнего обращения: декабря 5, 2025, <http://beletsky.net/2011/12/inside-aspnet-mvc-instantiation-of.html>
17. Declare C# primary constructors – classes, structs - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/primary-constructors>
18. Introducing Primary Constructor Usage in C# and .NET Core Applications - andrewhalil.com, дата последнего обращения: декабря 5, 2025, <https://andrewhalil.com/2025/10/09/introducing-primary-constructor-usage-in-c-and-net-core-applications/>
19. ASP.NET Core - Route Constraints - C# Corner, дата последнего обращения: декабря 5, 2025, <https://www.c-sharpcorner.com/blogs/asp-net-core-route-constraints>
20. asp.net core - Ambiguous routing need explanation as to why - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/78831871/ambiguous-routing-need-explanation-as-to-why>
21. Why Dependency Injection is a GAME CHANGER for Unity Games - YouTube, дата последнего обращения: декабря 5, 2025, <https://www.youtube.com/watch?v=65gsdxj6xmk>
22. MASTER ASP.NET Core DI: Singleton, Scoped, & Transient Explained in 15 Minutes (Avoid LifeTime BUGS) - YouTube, дата последнего обращения: декабря 5, 2025, <https://www.youtube.com/watch?v=6OgOYmPDRd4>
23. Dependency Injection in ASP.NET Core: Singleton vs Scoped vs Transient Explained!, дата последнего обращения: декабря 5, 2025, <https://www.youtube.com/watch?v=V-8HlozCTOU>
24. Can someone explain when to use Singleton, Scoped and Transient with some real life examples? : r/csharp - Reddit, дата последнего обращения: декабря 5, 2025, <https://www.reddit.com/r/csharp/comments/1acwtar/can_someone_explain_when_to_use_singleton_scoped/>
25. AddTransient, AddScoped and AddSingleton Services Differences - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/38138100/addtransient-addscoped-and-addsingleton-services-differences>
26. The ASP.NET Core Dependency Injection System | ABP.IO, дата последнего обращения: декабря 5, 2025, <https://abp.io/community/articles/the-asp.net-core-dependency-injection-system-3vbsdhq8>
27. Mastering Dependency Injection in ASP.NET Core – Complete Beginner to Advanced Guide, дата последнего обращения: декабря 5, 2025, <https://www.c-sharpcorner.com/article/mastering-dependency-injection-in-asp-net-core-complete-beginner-to-advanced-g/>
28. What replaces SynchronizationContext in ASP.NET Core? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/56059678/what-replaces-synchronizationcontext-in-asp-net-core>
29. Net 5 - Asp.Net still has no SynchronizationContext? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/65186102/net-5-asp-net-still-has-no-synchronizationcontext>
30. Async Mistakes that can kill an ASP.NET Core application - C# Corner, дата последнего обращения: декабря 5, 2025, <https://www.c-sharpcorner.com/article/common-mistakes-developers-make-async-core-how-to-avoid-them/>
31. Asynchronous coroutines with C# and IAsyncEnumerable - DEV Community, дата последнего обращения: декабря 5, 2025, <https://dev.to/noseratio/asynchronous-coroutines-with-c-8-0-and-iasyncenumerable-2e04>
32. ASP.NET Core MVC Controller vs. minimal API vs. FastEndpoints — What's the Best for Performance ? | by Denis Makarenko | Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@denmaklucky/asp-net-core-mvc-controller-vs-minimal-api-vs-fastendpoints-whats-the-best-for-performance-cfee8a1809d7>
33. Minimal API Performance Benchmark - DEV Community, дата последнего обращения: декабря 5, 2025, <https://dev.to/stevsharp/minimal-api-performance-benchmark-14ln>