# Архитектурная анатомия асинхронности в.NET: Глубокий анализ машины состояний и среды выполнения

## Отчет по материалам Недели 3, Дня 2 учебного плана «Transition from Unity to Backend»

Автор: Старший технический архитектор.NET / Performance Engineer

Дата: 3 декабря 2025 года

Контекст: Глубокое погружение в теоретические основы асинхронного программирования (Async/Await internals), предназначенное для инженеров, совершающих переход от разработки на Unity к Enterprise Backend системам.

## 1. Введение: Онтологический сдвиг парадигмы выполнения

Переход от разработки клиентских приложений на Unity к созданию масштабируемых бэкенд-систем на.NET Core требует фундаментального пересмотра ментальной модели выполнения программного кода. В среде Unity разработчик оперирует в детерминированном мире «Игрового Цикла» (Game Loop), где выполнение кода строго синхронизировано с частотой обновления кадров. Весь код, от физики до пользовательского интерфейса, выполняется в одном главном потоке (Main Thread), а асинхронность часто эмулируется через корутины (IEnumerator), которые технически являются механизмом кооперативной многозадачности, выполняемой в том же потоке.1

В мире Enterprise Backend, и в частности в ASP.NET Core, среда выполнения стохастична. Сервер обрабатывает тысячи запросов одновременно, используя пул потоков (ThreadPool). Ключевые слова async и await, введенные в C# 5.0, являются не просто синтаксическим сахаром, но мощной абстракцией, скрывающей сложнейшую машину состояний, управляющую разрывом и восстановлением потока выполнения. Для старшего инженера критически важно понимать, что происходит «под капотом» этой абстракции, поскольку именно там кроются причины дедлоков (deadlocks), утечек памяти (memory leaks) и проблем с производительностью под высокой нагрузкой (thread starvation).

Темы второго дня третьей недели обучения фокусируются на демистификации этой «магии» компилятора. Данный отчет представляет собой исчерпывающий теоретический анализ внутреннего устройства async/await, механизма машины состояний, управления контекстом синхронизации и специфики обработки исключений в асинхронном коде.2

## 2. Демистификация Машины Состояний (IAsyncStateMachine)

Когда разработчик применяет модификатор async к методу, он дает директиву компилятору Roslyn полностью трансформировать тело метода. Это процесс, известный в компиляторостроении как «Lowering» (понижение уровня абстракции). Исходный метод превращается в метод-заглушку (Stub) и вспомогательную структуру данных, реализующую интерфейс System.Runtime.CompilerServices.IAsyncStateMachine.3

### 2.1. Трансформация: От Метода к Структуре

Рассмотрим канонический пример асинхронного метода:

C#

public async Task<int> CalculateAsync(int baseValue)  
{  
 int factor = 10;  
 await Task.Delay(100); // Первая точка приостановки  
 return baseValue \* factor;  
}

Для CLR (Common Language Runtime) этого метода в таком виде не существует. Компилятор генерирует следующий архитектурный паттерн:

1. **Метод-стаб (Stub Method):** Сохраняет исходную сигнатуру. Его задача — инициализировать машину состояний, запустить её (метод Start) и вернуть клиенту объект Task, который будет служить «обещанием» (Promise) будущего результата.
2. **Структура Машины Состояний:** Сгенерированный тип (обычно с нечитаемым именем вроде <CalculateAsync>d\_\_0), который инкапсулирует всю логику метода.

#### Почему struct, а не class?

В релизных сборках (Release mode) машина состояний генерируется как значимый тип (struct). Это критически важное оптимизационное решение команды.NET. Если асинхронный метод завершается синхронно (например, await вызывается для задачи, которая уже завершена, или используется ValueTask), машина состояний никогда не покидает стек. Это обеспечивает так называемый «Fast Path» — выполнение без аллокаций в куче (Zero Allocation).

Только в случае, если методу действительно нужно приостановить выполнение (await незавершенной задачи), структура подвергается «упаковке» (Boxing) в кучу, чтобы пережить возврат из метода-стаба.6

Примечание: В режиме Debug машина часто генерируется как класс для поддержки функции Edit & Continue.8

### 2.2. Анатомия полей Машины Состояний

Сгенерированная структура содержит набор полей, которые можно классифицировать на три категории: служебные, локальные переменные (поднятые) и временные.9

| **Категория поля** | **Тип** | **Назначение** | **Влияние на архитектуру** |
| --- | --- | --- | --- |
| **Служебное** | int <>1\_\_state | Хранит текущее состояние выполнения.  -1: Created/Running (Работает или не запущен).  -2: Done (Завершен успешно или с ошибкой).  0, 1,...: Индекс оператора await, на котором приостановлено выполнение. | Позволяет методу MoveNext перепрыгнуть (через goto) к нужной точке кода при возобновлении. |
| **Служебное** | AsyncTaskMethodBuilder<T> | Строитель метода. Отвечает за создание возвращаемого Task, привязку машины состояний к продолжению и установку результата/исключения. | Ключевой элемент интеграции с TPL (Task Parallel Library). |
| **Поднятое (Hoisted)** | int baseValue | Аргумент исходного метода. | Аргументы должны быть сохранены, так как стек исходного вызова будет уничтожен при первой приостановке. |
| **Поднятое (Hoisted)** | int <factor>5\_\_1 | Локальная переменная factor из тела метода. | **Критический момент:** Локальные переменные превращаются в поля структуры. Это увеличивает размер машины состояний и удерживает ссылки на объекты дольше, чем в синхронном коде. |
| **Временное** | TaskAwaiter <>u\_\_1 | Хранилище для авайтера (объекта ожидания). | Используется для получения результата (GetResult) после возобновления выполнения. |

### 2.3. Механика метода MoveNext(): Сердце асинхронности

Вся логика исходного метода переносится в метод void MoveNext(), который реализует интерфейс IAsyncStateMachine. Этот метод представляет собой один большой конечный автомат, управляемый полем state.4

#### Алгоритмический разбор выполнения MoveNext:

1. Вход и маршрутизация:  
   В начале метода находится оператор switch (или цепочка if) по полю state. Если state == -1, выполнение начинается с начала метода. Если state == 0, происходит переход (goto) на метку, расположенную сразу после первого оператора await.
2. Синхронное выполнение:  
   Код выполняется синхронно до тех пор, пока не встретится выражение await task.
3. Запрос авайтера:  
   Для ожидаемой задачи вызывается метод GetAwaiter(). Это возвращает объект TaskAwaiter.
4. Проверка завершенности (IsCompleted):  
   Машина состояний проверяет свойство awaiter.IsCompleted. Это развилка «Fast Path» vs «Slow Path».
   * **Fast Path (Синхронное завершение):** Если задача уже завершена (например, Task.CompletedTask или результат закэширован), метод **не приостанавливается**. Код продолжает выполняться синхронно. Это объясняет, почему частые вызовы await для завершенных задач дешевы.
   * **Slow Path (Асинхронное ожидание):**
     1. Текущее значение state устанавливается в 0 (индекс текущего await).
     2. Авайтер сохраняется в поле <>u\_\_1.
     3. Вызывается метод builder.AwaitUnsafeOnCompleted(ref awaiter, ref this). Этот метод регистрирует машину состояний как продолжение (continuation) для задачи.
     4. Происходит возврат (return) из метода MoveNext.
     5. Поток, который выполнял этот код, освобождается и возвращается в пул (или продолжает выполнять вызывающий код).
5. Возобновление (Continuation):  
   Когда ожидаемая операция (например, I/O запрос к базе данных) завершается, она инициирует вызов делегата, указывающего на MoveNext.
   * Метод MoveNext вызывается снова (возможно, на другом потоке).
   * state равен 0, поэтому выполнение прыгает на метку после await.
   * Вызывается awaiter.GetResult(). Если операция завершилась ошибкой, именно здесь будет выброшено исключение.
   * Поле авайтера обнуляется для снижения давления на GC.
6. Завершение:  
   Когда логика метода доходит до конца, вызывается builder.SetResult(result). Это переводит Task, который был возвращен клиенту в самом начале, в состояние RanToCompletion.13

### 2.4. «Поднятие» переменных (Variable Hoisting) и влияние на память

Процесс превращения локальных переменных в поля структуры называется «поднятием» (Hoisting). В контексте управления памятью это имеет серьезные последствия, которые часто игнорируются разработчиками, пришедшими из Unity.6

Проблема: В синхронном методе время жизни локальной переменной ограничено блоком кода. Как только выполнение покидает блок, переменная удаляется со стека. В асинхронном методе переменная живет столько же, сколько живет объект машины состояний (то есть Task).

Пример: Если вы объявили временный буфер byte buffer = new byte[1024 \* 1024]; в начале метода, и используете его, а затем делаете долгий await Task.Delay(10000), этот мегабайт памяти будет удерживаться в куче (так как машина упакована) все 10 секунд, даже если буфер больше не нужен.

**Рекомендация:** В высоконагруженном коде следует явно присваивать null локальным переменным ссылочного типа перед длительными await, если они больше не нужны, или ограничивать их область видимости фигурными скобками { }, чтобы компилятор мог оптимизировать поля машины состояний (переиспользовать слоты полей).15

## 3. Контекст синхронизации (SynchronizationContext)

Понимание SynchronizationContext является ключевым для предотвращения дедлоков и понимания того, почему код ведет себя по-разному в Unity, WPF и ASP.NET Core.

### 3.1. Философия Контекста: Где выполняется код?

SynchronizationContext — это абстракция, которая отвечает на вопрос: «Где должен быть выполнен этот фрагмент кода?». await использует эту абстракцию для планирования продолжения (кода после await).17

Механизм следующий:

1. Перед приостановкой (await), инфраструктура захватывает текущий контекст: var context = SynchronizationContext.Current;.
2. Когда задача завершается, если контекст был захвачен, продолжение планируется через метод context.Post(delegate, state).
3. Если контекста не было (null), продолжение планируется на планировщике задач по умолчанию (TaskScheduler.Default), что обычно означает ThreadPool.

### 3.2. Типология Контекстов

Различные среды выполнения предоставляют разные реализации этого класса, что диктует поведение асинхронного кода:

#### A. UnitySynchronizationContext (Unity Game Loop)

В среде Unity существует кастомный контекст, который жестко привязывает выполнение к главному потоку (Main Thread).19

* **Механизм:** Метод Post ставит делегат в очередь, которая разбирается движком Unity в определенной фазе кадра.
* **Следствие:** Любой код после await в Unity (по умолчанию) возвращается в главный поток. Это безопасно для работы с GameObject, но может вызвать падение FPS, если продолжение содержит тяжелые вычисления.

#### B. AspNetSynchronizationContext (Legacy ASP.NET /.NET Framework)

В старом ASP.NET (WebForms, MVC 4/5) контекст гарантировал, что в любой момент времени запрос обрабатывается только одним потоком.21

* **Механизм:** Контекст привязан к текущему HttpContext. При завершении асинхронной операции контекст восстанавливает HttpContext.Current и личность пользователя (Principal).
* **Следствие:** Это основной источник дедлоков в старых системах.

#### C. ASP.NET Core (Отсутствие Контекста)

В ASP.NET Core было принято архитектурное решение **полностью отказаться** от SynchronizationContext.23

* **Причина:** Производительность. Восстановление контекста требует накладных расходов. Кроме того, это устраняет класс дедлоков, связанных с блокировкой потоков.
* **Следствие:** SynchronizationContext.Current в методах контроллера ASP.NET Core равен null. Код после await может выполняться на **любом** потоке из пула. Это означает, что вы не можете полагаться на то, что код «до» и «после» await выполняется одним и тем же потоком. Также HttpContext не является потокобезопасным, и доступ к нему из параллельных веток кода запрещен.

### 3.3. Анатомия Дедлока (The Deadlock)

Понимание механизма дедлока критично, так как многие Unity-разработчики переносят привычки (использование .Result) в бэкенд. Дедлок возникает при сочетании трех факторов 21:

1. Использование блокирующего ожидания (Task.Wait() или Task.Result).
2. Наличие однопоточного контекста синхронизации (UI или Legacy ASP.NET).
3. Ожидание задачи, которая пытается вернуться в этот же контекст.

**Сценарий:**

1. Главный поток (UI) вызывает MethodAsync().Result. Главный поток блокируется и ждет.
2. MethodAsync запускает операцию и делает await.
3. Операция завершается. Машина состояний видит захваченный UI-контекст и вызывает context.Post, чтобы выполнить продолжение на главном потоке.
4. **Тупик:** Главный поток не может обработать сообщение из очереди Post, так как он синхронно ждет завершения MethodAsync. А MethodAsync не может завершиться, пока главный поток не выполнит его продолжение.

Почему этого нет в ASP.NET Core?

Так как контекста нет (null), продолжение планируется в ThreadPool. Поток пула свободен, он выполняет код, задача завершается, и блокировка .Result снимается. Однако, использовать блокировки в Core все равно считается антипаттерном («Sync over Async»), так как это приводит к истощению пула потоков (Thread Starvation).28

### 3.4. ConfigureAwait(false): Глубинная механика

Метод ConfigureAwait(false) — это инструкция машине состояний: «Не захватывай контекст синхронизации».

* **Внутренняя реализация:** Билдер вызывает AwaitUnsafeOnCompleted с флагом, запрещающим маршалинг обратно в контекст. Продолжение всегда уходит в ThreadPool.30
* **В Unity:** Использование ConfigureAwait(false) опасно, если вы работаете с API Unity (Transform, Rigidbody) после await, так как код выполнится в фоновом потоке, и Unity выбросит исключение.
* **В библиотеках:** Обязательно к использованию, чтобы не навязывать контекст вызывающего кода библиотечному методу.

## 4. Обработка исключений: Async Void и AggregateException

Модель обработки ошибок в асинхронном коде существенно отличается от синхронной и имеет свои «минные поля».

### 4.1. Async Task vs. Async Void: Архитектурная пропасть

Различие между async Task и async void касается не только возвращаемого значения, но и механизма распространения исключений.32

#### Async Task (Безопасный подход)

Когда исключение происходит внутри метода async Task:

1. Машина состояний перехватывает его внутри MoveNext.
2. Вызывается builder.SetException(ex).
3. Исключение **сохраняется** внутри объекта Task. Задача переходит в статус Faulted.
4. Исключение всплывает только тогда, когда вызывающий код делает await task.

#### Async Void (Катастрофический подход)

Методы async void существуют только для совместимости с event handlers (например, Button\_Click). У них нет объекта Task, куда можно «положить» ошибку.34

**Механизм краша:**

1. Исключение перехватывается в MoveNext.
2. Так как Task нет, билдер не может сохранить ошибку.
3. Исключение выбрасывается (re-throw) **напрямую в SynchronizationContext**, который был активен при запуске.
4. Если контекста нет (Console, ASP.NET Core), исключение выбрасывается в ThreadPool.
5. Это вызывает событие AppDomain.UnhandledException, что неминуемо приводит к завершению процесса (Process Termination). Даже try-catch вокруг вызова async void метода не поможет, так как метод возвращает управление мгновенно, а ошибка происходит асинхронно позже.35

**Правило:** Никогда не используйте async void в бэкенде. Даже для фоновых задач («Fire and Forget») используйте Task.Run и регистрируйте продолжение для логирования ошибок.

### 4.2. Агрегирование исключений (AggregateException)

При использовании Task.WhenAll вы ожидаете несколько задач. Если несколько из них упадут с ошибкой, результирующая задача будет содержать AggregateException, внутри которого находится список всех исключений (InnerExceptions).38

Однако оператор await спроектирован так, чтобы эмулировать поведение синхронного кода, где может быть выброшено только одно исключение. Поэтому await taskWhenAll **распаковывает** агрегат и выбрасывает только **первое** исключение из списка. Остальные ошибки «проглатываются» для блока catch.40

**Паттерн правильной обработки:**

C#

Task all = Task.WhenAll(tasks);  
try  
{  
 await all;  
}  
catch  
{  
 // Здесь мы поймали только первое исключение.  
 // Чтобы увидеть все, нужно обратиться к самой задаче:  
 foreach (var ex in all.Exception.InnerExceptions)  
 {  
 logger.LogError(ex);  
 }  
}

### 4.3. ExceptionDispatchInfo

Для сохранения стека вызовов при передаче исключения через границы потоков используется класс System.Runtime.ExceptionServices.ExceptionDispatchInfo. Асинхронная машина состояний использует его внутренне. Это позволяет при await видеть полный стек-трейс, как если бы вызов был синхронным, а не обрезанный стек, начинающийся с метода MoveNext.42

## 5. Внутреннее устройство и Производительность

Для Senior-разработчика важно понимать не только как работает код, но и сколько он стоит (Cost of Abstraction).

### 5.1. Allocations (Аллокации)

Асинхронность в C# не бесплатна. Каждая асинхронная операция (которая не завершилась синхронно) влечет за собой:

1. **Boxing машины состояний:** Структура переносится в кучу (Heap).6
2. **Task object:** Аллокация самого объекта задачи.
3. **Delegate allocation:** Создание делегата Action для передачи в OnCompleted.
4. **Hoisted Locals:** Все локальные переменные живут в куче внутри машины состояний.

### 5.2. Оптимизации.NET Core /.NET 5+

Microsoft проделала огромную работу по оптимизации этого механизма 9:

1. **ValueTask:** Позволяет избежать аллокации Task, если результат уже готов (I/O буферизирован) или при использовании пулинга.
2. **PoolingAsyncValueTaskMethodBuilder:** Введенный в новых версиях (.NET 6+), этот механизм позволяет **переиспользовать** объекты машин состояний (как Object Pooling в Unity). Это достигается атрибутом ``. Это радикально снижает давление на GC в высоконагруженных сценариях (например, при чтении из сети).
3. **Уменьшение размера машины:** Компилятор стал умнее в переиспользовании полей для временных переменных.

## 6. Рекомендации по Рефакторингу (Transition Guide)

При переходе от Unity-стиля к Enterprise коду, следуйте следующим паттернам:

| **Паттерн Unity** | **Паттерн.NET Backend** | **Почему?** |
| --- | --- | --- |
| Coroutine с yield return | async Task / async ValueTask | Типизация, обработка ошибок, возможность возврата значений. |
| void Update() | Бесконечный цикл в BackgroundService | Бэкенд не имеет кадровой структуры. |
| GameObject.SendMessage | Channel<T> или MediatR | Типизированная, асинхронная передача сообщений без рефлексии. |
| Коллбэки (Action<T> callback) | await результата | Избегание "Callback Hell", линейность кода. |
| try-catch внутри корутины | try-catch вокруг await | Исключения всплывают по стеку корректно. |

### Практическое задание: Декомпиляция

Для закрепления материала рекомендуется использовать инструмент **SharpLab.io** или **dotPeek**. Напишите простой метод с await, циклом try-catch и посмотрите, во что он превращается. Найдите метод MoveNext, поле <>1\_\_state и обратите внимание на то, как обрабатывается исключение внутри блока catch (вызов SetException). Это визуализирует абстракцию и демистифицирует процесс.

### Таблица Сводных Данных по Средам Выполнения

| **Характеристика** | **Unity** | **Legacy ASP.NET** | **ASP.NET Core** |
| --- | --- | --- | --- |
| **SynchronizationContext** | UnitySynchronizationContext | AspNetSynchronizationContext | null (None) |
| **Threading Model** | Single Threaded (Main) | One Thread per Request | ThreadPool (Agile) |
| **Async Void Safety** | Относительно безопасно (логи в консоль) | Опасно (может убить пул) | Фатально (Process Crash) |
| **ConfigureAwait(false)** | **Опасно** (если нужен доступ к API движка) | **Рекомендуется** (избегание дедлоков) | **Без разницы** (но рекомендуется для библиотек) |
| **Deadlock Risk** | Низкий (обычно зависание UI) | Высокий (блокировка Request Queue) | Низкий (нет контекста) |

Этот отчет завершает теоретический обзор второго дня. Глубокое понимание этих механик позволяет писать высокопроизводительный, надежный код и избегать распространенных ловушек при миграции с клиентской разработки на серверную.

#### Источники

1. Разработка бэкенда: план обучения C#, <https://drive.google.com/open?id=12UXauWxfywsyECXIJhYT_VLYCVRzly-AgMy8MLGdNis>
2. План обучения C# третьей недели, <https://drive.google.com/open?id=1b2lowEHeEu3G5cvI1pNLgzC_eEIr_IPseENNbu1AyFA>
3. Async/Await in C# is syntax sugar for StateMachine | by Ranjeet Singh - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@ranjeetdotme/how-async-await-works-in-c-eccf16bd3b90>
4. Understanding async/await State Machine in .NET - Just a blog, дата последнего обращения: декабря 3, 2025, <https://mykkon.work/async-state-machine/>
5. What is the purpose of IAsyncStateMachine.SetStateMachine? - Codemia, дата последнего обращения: декабря 3, 2025, <https://codemia.io/knowledge-hub/path/what_is_the_purpose_of_iasyncstatemachinesetstatemachine>
6. Asynchronous Programming - Async Performance: Understanding the Costs of Async and Await | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/archive/msdn-magazine/2011/october/asynchronous-programming-async-performance-understanding-the-costs-of-async-and-await>
7. How does async/await work under the hood (IL level) ? : r/csharp - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/csharp/comments/1hw3f35/how_does_asyncawait_work_under_the_hood_il_level/>
8. Why are async state machines classes (and not structs) in Roslyn? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/33871181/why-are-async-state-machines-classes-and-not-structs-in-roslyn>
9. How Async/Await Really Works in C# - .NET Blog, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/dotnet/how-async-await-really-works/>
10. Do all value types become subject to GC in when async function is suspended?, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/77239277/do-all-value-types-become-subject-to-gc-in-when-async-function-is-suspended>
11. Is that make sense atomic for MoveNext in async generated StateMachine - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/11623048/is-that-make-sense-atomic-for-movenext-in-async-generated-statemachine>
12. Asynchronous C#: Below the Surface - Aaron Bos, дата последнего обращения: декабря 3, 2025, <https://aaronbos.dev/posts/async-csharp-below-surface>
13. Avoid memory allocations and data copies - C# | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/performance/>
14. [C#] Using GC.KeepAlive in async methods | by Kevin Gosse - Medium, дата последнего обращения: декабря 3, 2025, <https://kevingosse.medium.com/c-using-gc-keepalive-in-async-methods-8d20fd79f0a0>
15. Are async methods' local variables Garbage Collected while the task is still running?, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/73433195/are-async-methods-local-variables-garbage-collected-while-the-task-is-still-run>
16. Dissecting Memory Management in .NET: Variable Lifetime, GC, JIT & Async - YouTube, дата последнего обращения: декабря 3, 2025, <https://www.youtube.com/watch?v=Ssu4o14Tohg>
17. Demystifying Synchronization Context in .NET | by José Sousa | Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@josesousa8/demystifying-synchronization-context-in-net-7ddf4473efcb>
18. ExecutionContext vs SynchronizationContext - .NET Blog, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/dotnet/executioncontext-vs-synchronizationcontext/>
19. Manual: Overview of .NET in Unity, дата последнего обращения: декабря 3, 2025, <https://docs.unity3d.com/2020.1/Documentation/Manual/overview-of-dot-net-in-unity.html>
20. Looking into Unity's async/await, дата последнего обращения: декабря 3, 2025, <https://gametorrahod.com/unity-and-async-await/>
21. async/await deadlocking when using a SynchronizationContext - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/34151179/async-await-deadlocking-when-using-a-synchronizationcontext>
22. ConfigureAwait(false) relevant in ASP.NET Core? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/42053135/configureawaitfalse-relevant-in-asp-net-core>
23. ConfigureAwait and SynchronizationContext in C# - GeeksforGeeks, дата последнего обращения: декабря 3, 2025, <https://www.geeksforgeeks.org/c-sharp/synchronizationcontext-and-configureawait-in-c-sharp/>
24. ASP.NET Core Blazor synchronization context - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/aspnet/core/blazor/components/synchronization-context?view=aspnetcore-10.0>
25. I didn't understand why people struggled with (.NET's) async - Productive Rage, дата последнего обращения: декабря 3, 2025, <https://www.productiverage.com/i-didnt-understand-why-people-struggled-with-nets-async>
26. C# Deadlocks in Depth – Part 2 | Michael's Coding Spot, дата последнего обращения: декабря 3, 2025, <https://michaelscodingspot.com/c-deadlocks-in-depth-part-2/>
27. C Async await deadlock problem gone in .NetCore? - Codemia, дата последнего обращения: декабря 3, 2025, <https://codemia.io/knowledge-hub/path/c_async_await_deadlock_problem_gone_in_netcore>
28. C# Async await deadlock problem gone in .NetCore? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/53265020/c-sharp-async-await-deadlock-problem-gone-in-netcore>
29. Not much difference between ASP.NET Core sync and async controller actions, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/48015096/not-much-difference-between-asp-net-core-sync-and-async-controller-actions>
30. Task.ConfigureAwait Method (System.Threading.Tasks) | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.configureawait?view=net-10.0>
31. Why I no longer use ConfigureAwait(false) - DEV Community, дата последнего обращения: декабря 3, 2025, <https://dev.to/noseratio/why-i-no-longer-use-configureawait-false-3pne>
32. Async/Await - Best Practices in Asynchronous Programming | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/archive/msdn-magazine/2013/march/async-await-best-practices-in-asynchronous-programming>
33. Async Task v/s Async void in C#: What's the Difference and Why It Matters? | by Pavan pitthdiya | Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@pavanpitthdiya/async-task-v-s-async-void-in-c-whats-the-difference-and-why-it-matters-19b53e7c10d8>
34. c# - async/await - when to return a Task vs void? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/12144077/async-await-when-to-return-a-task-vs-void>
35. AppDomain.UnhandledException Event (System) - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.appdomain.unhandledexception?view=net-10.0>
36. Async method exception crashing application - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/48344645/async-method-exception-crashing-application>
37. Sneaky async void Leads to ASP.NET Core Crash | Josh the Coder, дата последнего обращения: декабря 3, 2025, <https://joshthecoder.com/2023/12/01/sneaky-async-void-leads-to-aspnetcore-crash.html>
38. Why doesn't await on Task.WhenAll throw an AggregateException? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/12007781/why-doesnt-await-on-task-whenall-throw-an-aggregateexception>
39. Exception handling (Task Parallel Library) - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/exception-handling-task-parallel-library>
40. Advanced stuff about Task they don't tell you | by iamprovidence - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@iamprovidence/advanced-stuff-about-task-they-dont-tell-you-619ccb4784df>
41. AggregateException from Task.WhenAll only contains first exception when awaited, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/61370351/aggregateexception-from-task-whenall-only-contains-first-exception-when-awaited>
42. A try/catch wrapper for C# async functions up to lucky 7 argument. - GitHub Gist, дата последнего обращения: декабря 3, 2025, <https://gist.github.com/ghstahl/7022ee06c1f9a1753a11efb51882740c>
43. What's the point of passing ExceptionDispatchInfo around instead of just the Exception? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/45723327/whats-the-point-of-passing-exceptiondispatchinfo-around-instead-of-just-the-exc>
44. Demystifying async/await in C# .NET 8: Optimizing Performance and Responsiveness | by Robert Dennyson | Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@robertdennyson/demystifying-async-await-in-c-net-8-optimizing-performance-and-responsiveness-b04f5e32d0d2>
45. What happened to "Make async ValueTask/ValueTask methods ammortized allocation-free"? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/77696198/what-happened-to-make-async-valuetask-valuetask-methods-ammortized-allocation-f>