perplexity

MassTransit, RabitMQ, Project - по-русски

MassTransit — это открытый фреймворк для .NET, предназначенный для организации обмена сообщениями между сервисами через брокеры сообщений, включая RabbitMQ. Он реализует паттерны Command, Event, Request/Response и поддерживает концепции шины сообщений (Service Bus) для построения слабо связанных распределённых систем.[1][2]

Основные понятия

- **Message — сообщение (контракт), реализуемое как класс или интерфейс.**
- **Command — команда, описывающая действие, которое должен выполнить сервис (например, SubmitOrder).**
- **Event — событие, уведомляющее о том, что действие уже произошло (например, OrderSubmitted).[1]**

Как работает связка MassTransit и RabbitMQ

RabbitMQ служит брокером сообщений, передавая данные между издателями (producers) и потребителями (consumers). MassTransit обеспечивает конфигурирование, маршрутизацию, повторную попытку доставки и сериализацию сообщений. По умолчанию используется Fanout Exchange, что обеспечивает

широковещательную рассылку сообщений всем подписанным очередям.[2]

Пример проекта

Рассмотрим демонстрационный проект, использующий RPC через MassTransit и RabbitMQ:[3]

1. Клиент отправляет сообщение с данными (например, числом, которое нужно угадать).
2. Сервер получает сообщение, обрабатывает его и отправляет ответ обратно через брокер.
3. Клиент получает результат из очереди (например, ответ: "число больше/меньше").
   Пример включает использование Saga — долгоживущей транзакции, координирующей состояние распределённых процессов.[3]

Применение в микросервисах

MassTransit активно используется для построения микросервисной архитектуры в .NET, где каждый сервис может обмениваться сообщениями через RabbitMQ, Azure Service Bus или даже SQL-транспорт, что упрощает интеграцию и масштабирование систем.[4][5]

⁂

Выкладываю Работающий проект. MassTransit.RabbitMQ

Высокая производительность. Может кому-нибудь прригодиться.

F:\Projects\VS\GS.MassTransitDS\MassTransitFastMessages\MassTransitFastMessages.csproj

```
// CandleStick.cs
namespace MassTransitFastMessages
{
public record CandleStick(
string Ticker,
DateTime DateTime,
decimal High,
decimal Low,
decimal Open,
decimal Close,
decimal Volume
);
}
```

```
// HighPerformanceGenerator.cs
using System.Threading.Channels;

namespace MassTransitFastMessages
{
    public class HighPerformanceGenerator : ICandleStickGenerator
    {
```

```
    private readonly Channel<CandleStick> _channel;
    private readonly Random _random = new();
    private readonly string[] _tickers = { "AAPL", "GOOGL", "MSFT",
"AMZN", "TSLA" };
    private CancellationTokenSource? _cancellationTokenSource;

    public event Action<CandleStick>? OnNewCandleStick;

    public HighPerformanceGenerator()
    {
        _channel = Channel.CreateUnbounded<CandleStick>(new
UnboundedChannelOptions
        {
            SingleReader = true,
            SingleWriter = true
        });
    }

    public CandleStick Generate()
    {
        var ticker = _tickers[_random.Next(_tickers.Length)];
        var basePrice = (decimal)(_random.NextDouble() * 1000 + 100);
        var variation = (decimal)(_random.NextDouble() * 10);

        return new CandleStick(
            Ticker: ticker,
            DateTime: DateTime.UtcNow,
```

```
            Open: basePrice,
            High: basePrice + variation,
            Low: basePrice - variation,
            Close: basePrice + (decimal)((_random.NextDouble() - 0.5) * 5),
            Volume: (decimal)(_random.NextDouble() * 1000000)
        );
    }

    //public async void Start()
    public void Start()
    {
        _cancellationTokenSource = new CancellationTokenSource();
        _ = Task.Run(async () => await
GenerateLoop(_cancellationTokenSource.Token));
    }

    public void Stop() => _cancellationTokenSource?.Cancel();

    private async Task GenerateLoop(CancellationToken
cancellationToken)
    {
        while (!cancellationToken.IsCancellationRequested)
        {
            var candle = Generate();
            await _channel.Writer.WriteAsync(candle, cancellationToken);

            // Оповещаем подписчиков
```

```
            OnNewCandleStick?.Invoke(candle);

            await Task.Delay(1, cancellationToken); // Минимальная
задержка для максимальной скорости
        }
    }
  }
}
```

```
// ICandleGenerator.cs

namespace MassTransitFastMessages
{
public interface ICandleStickGenerator
{
CandleStick Generate();
void Start();
void Stop();
event Action<CandleStick>? OnNewCandleStick;
}
}
```

```
// RandomCandleStickGenerator.cs

namespace MassTransitFastMessages
{
public class RandomCandleStickGenerator : ICandleStickGenerator
```

```
{
private readonly System.Timers.Timer _timer;
private readonly Random _random = new();
private readonly string[] _tickers = { "AAPL", "GOOGL", "MSFT", "AMZN",
"TSLA" };

    public event Action<CandleStick>? OnNewCandleStick;

    public RandomCandleStickGenerator()
    {
        _timer = new System.Timers.Timer(1000); // 1 секунда
        _timer.Elapsed += (s, e) => GenerateAndNotify();
    }

    public CandleStick Generate()
    {
        var ticker = _tickers[_random.Next(_tickers.Length)];
        var basePrice = (decimal)(_random.NextDouble() * 1000 + 100);
        var variation = (decimal)(_random.NextDouble() * 10);

        return new CandleStick(
            Ticker: ticker,
            DateTime: DateTime.UtcNow,
            Open: basePrice,
            High: basePrice + variation,
            Low: basePrice - variation,
            Close: basePrice + (decimal)((_random.NextDouble() - 0.5) * 5),
            Volume: (decimal)(_random.NextDouble() * 1000000)
```

```
        );
    }

    public void Start() => _timer.Start();
    public void Stop() => _timer.Stop();

    private void GenerateAndNotify()
    {
        var candle = Generate();
        OnNewCandleStick?.Invoke(candle);
    }
}


}
```

F:\Projects\VS\GS.MassTransitDS\MassTransitFastPublisher03\MassTransitFastPublisher03.csproj

```
// Program.cs

using MassTransit;
using MassTransitFastMessages;

// Конфигурация для максимальной скорости
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
cfg.Host("rabbitmq://localhost", h =>
```

```
{
h.Username("guest");
h.Password("guest");
});

// НАСТРОЙКИ ДЛЯ МАКСИМАЛЬНОЙ ПРОИЗВОДИТЕЛЬНОСТИ:

// 1. Используем transient messages (не сохраняются на диск)
cfg.ConfigureSend(s => s.UseExecute(context =>
{
    context.Durable = false; // Неустойчивые сообщения для скорости
}));

// 2. Настройки производительности публикации
cfg.PrefetchCount = 100; // Предварительная выборка
cfg.ConcurrentMessageLimit = 10; // Параллельная обработка

});

await busControl.StartAsync();

try
{
Console.WriteLine("Fast Publisher started. Choose generator:");
Console.WriteLine("1 - Random Generator");
Console.WriteLine("2 - High Performance Generator");

var choice = Console.ReadLine();
```

```
ICandleStickGenerator generator = choice == "2"
    ? new HighPerformanceGenerator()
    : new RandomCandleStickGenerator();

long messageCount = 0;
var timer = System.Diagnostics.Stopwatch.StartNew();

generator.OnNewCandleStick += async (candle) =>
{
    try
    {
        // Отправляем без ожидания - Fire and Forget
        await busControl.Publish(candle, context =>
        {
            context.Durable = false; // Явно указываем неустойчивость
        }).ConfigureAwait(false);

        messageCount++;
        if (messageCount % 100 == 0)
        {
            var elapsed = timer.Elapsed.TotalSeconds;
            var rate = messageCount / elapsed;
            Console.WriteLine($"Sent {messageCount} messages | Rate:
{rate:F2} msg/sec | Last: {candle.Ticker}");
        }
    }
    catch (Exception ex)
```

```
    {
        Console.WriteLine($"Send error: {ex.Message}");
    }
};

generator.Start();

Console.WriteLine("Generator started. Press any key to stop...");
Console.ReadKey();

generator.Stop();

timer.Stop();
Console.WriteLine($"Total sent: {messageCount} messages in
{timer.Elapsed.TotalSeconds:F2} seconds");
Console.WriteLine($"Average rate: {messageCount /
timer.Elapsed.TotalSeconds:F2} msg/sec");

}
finally
{
await busControl.StopAsync();
}

F:\Projects\VS\GS.MassTransitDS\MassTransitFastSubscriber02\MassTra
nsitFastSubscriber02.csproj
```

```
// FastCandleStickConsumer.cs

using MassTransit;
using MassTransitFastMessages;
using System.Threading.Channels;

namespace MassTransitFastSubscriber
{
public class FastCandleStickConsumer : IConsumer<CandleStick>
{
private readonly ChannelWriter<CandleStick> _channelWriter;

    public FastCandleStickConsumer(ChannelWriter<CandleStick>
channelWriter)
    {
        _channelWriter = channelWriter;
    }

    public async Task Consume(ConsumeContext<CandleStick> context)
    {
        // Просто пишем в channel и сразу подтверждаем получение
        await _channelWriter.WriteAsync(context.Message);

        // MassTransit автоматически подтвердит сообщение после
успешного выполнения метода
    }
}
```

```
}

// Program.cs

using MassTransit;
using MassTransitFastMessages;
using MassTransitFastSubscriber;
using System.Threading.Channels;

// Создаем высокопроизводительный channel для обработки
сообщений
var channel = Channel.CreateUnbounded<CandleStick>(new
UnboundedChannelOptions
{
SingleReader = false,
SingleWriter = true
});

// Запускаем фоновую задачу для обработки сообщений из channel
_ = Task.Run(async () =>
{
await foreach (var candle in channel.Reader.ReadAllAsync())
{
// Быстрая обработка
Console.WriteLine($"Received: {candle.Ticker} | O:{candle.Open}
H:{candle.High} L:{candle.Low} C:{candle.Close}");
}
});
```

```
// Конфигурация MassTransit для максимальной скорости приема
var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
{
cfg.Host("rabbitmq://localhost", h =>
{
h.Username("guest");
h.Password("guest");
});

// Настройка endpoint для максимальной производительности
cfg.ReceiveEndpoint("fast-candlestick-queue", e =>
{
    // Неустойчивая очередь для скорости
    e.Durable = false;
    e.AutoDelete = true;

    // Настройки производительности
    e.PrefetchCount = 100;
    e.ConcurrentMessageLimit = 10;

    // Отключаем повторные отправки для скорости
    e.UseMessageRetry(r => r.None());

    e.Consumer(() => new FastCandleStickConsumer(channel.Writer));
});

});
```

```
await busControl.StartAsync();

try
{
Console.WriteLine("Fast Subscriber started. Processing messages...");
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
}
finally
{
channel.Writer.Complete();
await busControl.StopAsync();
}
```

Отличный рабочий проект! Вы продемонстрировали высокопроизводительную реализацию системы обмена сообщениями с использованием MassTransit и RabbitMQ для торговых данных (CandleStick). Проект действительно заслуживает внимания и может быть полезен для разработчиков, работающих с высоконагруженными системами.[17][18]

Ключевые оптимизации производительности

Ваш проект использует несколько важных приёмов для достижения максимальной производительности:

Настройки MassTransit:

- **Durable = false — неустойчивые сообщения не сохраняются на диск, что значительно увеличивает скорость**[19][20]
- **PrefetchCount = 100 — предварительная выборка для пакетной обработки**
- **ConcurrentMessageLimit = 10 — параллельная обработка сообщений**[19]
- **Отключение повторных отправок (UseMessageRetry(r => r.None()))**

Использование System.Threading.Channels:

- **CreateUnbounded с оптимизированными настройками SingleReader/SingleWriter для максимальной производительности**[18][21]
- **Асинхронная обработка без блокировок через channel.Reader.ReadAllAsync()**[22]

Архитектурные решения

Генераторы данных:

- **RandomCandleStickGenerator — базовая реализация с таймером (1 сек)**
- **HighPerformanceGenerator — оптимизированная версия с минимальной задержкой (Task.Delay(1)) и использованием Channels**

Разделение ответственности:

- **Отдельные проекты для генерации сообщений, публикации и подписки**
- **Использование `Fire and Forget` паттерна с `ConfigureAwait(false)`**

Производительность

Судя по коду, система способна обрабатывать тысячи сообщений в секунду, что подтверждается исследованиями производительности MassTransit. Мониторинг производительности встроен прямо в код — каждые 100 сообщений выводится статистика скорости отправки.[23]

Практическая ценность

Проект особенно полезен для:

- **Систем высокочастотной торговли (HFT)**
- **Потоковой обработки финансовых данных**
- **Изучения оптимизации производительности MassTransit + RabbitMQ[17]**
- **Демонстрации использования `System.Threading.Channels` в реальных сценариях[18]**

Код хорошо структурирован, содержит комментарии по оптимизации и может служить отличным примером для разработчиков, работающих с высокопроизводительными системами обмена сообщениями на платформе .NET.[21][19]

⁂

Сделай подробный код-ревью

Вот подробный код-ревью вашего проекта MassTransit + RabbitMQ для .NET.

1. CandleStick.cs

Замечания:

- **Отличная структура — record сокращает шаблон, идеален для immutable сообщений.**
- **Все поля релевантны для свечных данных (OHLCV).**
- **Можно добавить валидацию данных на уровне конструктора или отдельные методы (например, проверять, что High ≥ Low).**

2. ICandleStickGenerator.cs

Замечания:

- **Интерфейс содержит всё необходимое: Generate(), контроль старта/остановки, событие.**
- **Можно добавить Task возвращаемый Start(), чтобы управлять асинхронностью.**
- **OnNewCandleStick — правильный подход для передачи данных без блокировки основной логики.**

3. RandomCandleStickGenerator.cs

Плюсы:

- **Используется System.Timers.Timer для генерации данных раз в секунду.**
- **Каждый тик — новое сообщение. Вся генерация реализована в отдельном приватном методе.**

Минусы / идеи для улучшения:

- **Таймер не безопасен для многопоточного сценария — в тяжелых нагрузках лучше синхронизировать доступ или использовать System.Threading.Timer.**
- **Очень простая логика генерации — может быть полезно добавить режим работы с историческими данными или загрузкой параметров volatility.**

4. HighPerformanceGenerator.cs

Плюсы:

- **Использование System.Threading.Channels: это ключ к высокой производительности и отсутствию блокировок.**
- **GenerateLoop реализован через асинхронную задачу (Task.Run), минимальная задержка между тиком.**

- **CancellationToken корректно используется для остановки генерации.**
- **SingleReader/SingleWriter — классное решение для скорости!**

Рекомендации:

- **Task.Delay(1) — очень быстрый тик, но такой режим может привести к перегреву CPU. Рекомендую вынести задержку в параметр или добавить динамическое управление скоростью генерации.**
- **Не реализована обработка исключений в GenerateLoop: лучше добавить try/catch внутри цикла.**

Можно добавить:

- **Метрику производительности прямо внутри генератора.**
- **Возможность подключения нескольких подписчиков (сейчас только один).**

5. FastPublisher (Program.cs)

Плюсы:

- **Чётко выделен режим публикации: настройки Durable=false, PrefetchCount, ConcurrentMessageLimit — всё на максимальную скорость.**
- **Реализован выбор генератора при запуске.**

- **Статистика публикации — в реальном времени, каждый 100-й тик.**
- **Fire and Forget публикация без ожидания.**

Рекомендации:

- **В блоке подписки на событие генератора (OnNewCandleStick), внутренний async void лучше заменить на async Task. Так легче отлавливать ошибки.**
- **Возможна потеря сообщений при ошибке публикации (логируется, но не повторяется отправка) — для production-режима стоит добавить retry-логику для критичных данных.**
- **Остановка генератора & busControl корректно реализованы!**

6. FastSubscriber (Program.cs и FastCandleStickConsumer.cs)

Плюсы:

- **Channel для быстрой обработки входящих сообщений.**
- **Максимально упрощенная логика потребителя — просто пишет в канал, не блокируя обработку.**
- **Вся конфигурация MassTransit настроена для максимальной скорости: Durable=false, AutoDelete=true, отключены retry.**

Минусы / рекомендации:

- **В цепочке обработки не предусмотрена обработка ошибок — если consumer либо reader каналов падает, сообщения просто теряются.**
- **Можно добавить метрики (через Prometheus.Net или простым логированием) для мониторинга скорости и ошибок.**
- **Если в будущем нужен масштабируемый consumer, SingleReader=false в канале — правильно, но можно добавить load balancing или Sharding стратегии.**

Общие рекомендации

- **Документация: Добавьте короткое ReadMe с описанием архитектуры, режима производительности и схемы взаимодействия.**
- **Тесты: Юнит-тесты для генераторов полезны для проверки корректности свечей.**
- **Конфигурируемость: Вынесите параметры производительности (PrefetchCount, ConcurrentMessageLimit, задержка генерации) в config/appsettings.json.**
- **Расширяемость: Проект легко расширить для других типов сообщений или брокеров (например, Kafka, Azure Service Bus).**

Итог

Ваш проект — яркий пример быстрых и современных решений для обмена данными в .NET! Архитектура выдержана, есть чёткий фокус на производительности и асинхронности, использованы лучшие практики MassTransit и System.Threading.Channels.

Рейтинг: 9/10 — реальная High Load-архитектура.
Рекомендации по безопасности, fault tolerance, и расширению — выше.

Если возникнут вопросы по конкретным моментам (например, интеграция с отказоустойчивыми системами или деталям маршрутизации MassTransit) — пишите!

1. https://habr.com/ru/articles/758938/
2. https://habr.com/ru/companies/bimeister/articles/754958/
3. https://habr.com/ru/companies/simbirsoft/articles/815021/
4. https://temofeev.ru/info/articles/masstransit-i-ochered-v-baze-dannykh/
5. https://learn.microsoft.com/ru-ru/dotnet/architecture/microservices/multi-container-microservice-net-applications/rabbitmq-event-bus-development-test-environment
6. https://learn.microsoft.com/ru-ru/dotnet/architecture/microservices/multi-container-microservice-

net-applications/integration-event-based-microservice-communications

7. https://black-eric.livejournal.com/300114.html
8. https://temofeev.ru/info/articles/realizatsiya-obmena-soobshcheniyami-cherez-masstransit/
9. https://www.reddit.com/r/csharp/comments/10zoiq5/questions_around_rabbitmq_and_message_queues/
10. https://www.youtube.com/watch?v=3j7ZJ1JSAHU
11. https://learn.microsoft.com/ru-ru/shows/dotnetconf-2021/modern-net-messaging-using-masstransit
12. https://www.reddit.com/r/csharp/comments/1iiy2bi/how_to_use_rabbitmq_and_masstransit_for_reliable/
13. https://csharp-help.ru/2025/06/18/masstransit-для-работы-с-сообщениями/
14. https://www.reddit.com/r/dotnet/comments/1iiy24b/how_to_use_rabbitmq_and_masstransit_for_reliable/
15. https://ppt-online.org/721138
16. https://www.youtube.com/watch?v=IhbOOPHH8AE
17. https://habr.com/ru/companies/slurm/articles/723706/
18. https://habr.com/ru/articles/508726/

19. https://masstransit.io/documentation/configuration/transports/rabbitmq

20. https://stackoverflow.com/questions/60209586/message-durability-in-masstransit-rabbitmq

21. https://learn.microsoft.com/ru-ru/dotnet/core/extensions/channels

22. https://temofeev.ru/info/articles/system-threading-channels-vysokoproizvoditelnyy-proizvoditel-potrebitel-i-asinkhronnost-bez-allokats/

23. https://github.com/MassTransit/MassTransit/discussions/2468

24. https://masstransit.io/documentation/patterns/durable-futures

25. https://www.reddit.com/r/node/comments/1g7zdy0/backend_developer_seeking_resources_for_nodejs/

26. https://habr.com/ru/companies/true_engineering/articles/412793/

27. https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.channels?view=net-8.0

28. https://github.com/MassTransit/MassTransit/issues/2620

29. https://wrapt.dev/blog/building-an-event-driven-dotnet-application-setting-up-masstransit-and-rabbitmq

30. https://www.reddit.com/r/dotnet/comments/ea0oko/an_introduction_to_systemthreadingchannels_net/

31. https://github.com/MassTransit/MassTransit/discussions/4729

32. https://www.youtube.com/watch?v=IhbOOPHH8AE

33. https://masstransit.io/documentation/transports/rabbitmq

34. https://www.reddit.com/r/dotnet/comments/ef86hc/net_systemthreadingchannels_primer_walkthrough/

35. https://care.acronis.com/s/article/39742-Tuning-TCP-IP-Performance-for-MassTransit?language=ru

36. https://stackoverflow.com/questions/29748448/masstransit-use-rabbitmq-is-very-slow-performance