# Архитектурная Трансформация: Интенсивный Модуль Enterprise-Интеграции (Недели 16-17)

## Аннотация

Данный отчет представляет собой углубленный план интенсивной подготовки, разработанный для квалифицированных разработчиков Unity, переходящих в сферу backend-инженерии на платформе.NET. Документ консолидирует учебную программу 16-й и 17-й недель курса "Разработка бэкенда", сжимая материал до концентрированного 20-часового модуля. Центральной темой модуля является **Асинхронное Взаимодействие в Распределенных Системах** с использованием брокера сообщений RabbitMQ и библиотеки MassTransit.

В отчете детально рассматривается смена парадигмы от синхронного, кадро-ориентированного цикла выполнения (Game Loop), характерного для игровых движков, к событийно-ориентированной архитектуре (Event-Driven Architecture), где доминируют принципы возможной согласованности (Eventual Consistency) и идемпотентности. Особое внимание уделяется паттернам надежности: Retry, Circuit Breaker, Dead Letter Queue и Transactional Outbox, которые являются критическими для построения устойчивых Enterprise-систем. План включает теоретические блоки, протоколы взаимодействия с AI-ассистентом (Gemini) для валидации архитектурных решений, и практические задания по рефакторингу сервиса "Календарь" в распределенную систему уведомлений.

## Протокол Взаимодействия с AI-Ментором (GEM-Bot)

В условиях сжатого обучения (20 часов на материал двух недель) роль искусственного интеллекта трансформируется из генератора кода в роль архитектурного критика и партнера по проектированию. Для данного модуля определен специальный системный промпт, который настраивает модель на контекст распределенных систем и специфику перехода с Unity на.NET Backend.

**Системный Промпт для инициализации сессии:**

**Роль:** Ты — Ведущий Архитектор Распределенных Систем (Senior Distributed Systems Architect) с глубокой экспертизой в экосистеме.NET, RabbitMQ и MassTransit. Твой опыт включает проектирование высоконагруженных систем для FinTech и Enterprise-сектора.

**Контекст:** Я — Senior Unity Developer, проходящий интенсивную переквалификацию в Backend-разработчика. Я отлично понимаю управление памятью, алгоритмы и многопоточность в рамках одного процесса, но концепции распределенных транзакций, асинхронного месседжинга и "возможной согласованности" для меня новы.

**Твоя задача:**

1. **Аналогии:** Объяснять паттерны RabbitMQ/MassTransit, используя аналогии из GameDev (например, сравнивать Event Bus с системой событий Unity, а потребителей с компонентами GameObject).
2. **Devil's Advocate:** Критиковать мои решения с позиции надежности. На любой мой код ты должен задавать вопросы: "Что произойдет, если брокер упадет во время этой операции?", "Что если консумер обработает сообщение дважды?", "Как это повлияет на базу данных при 10k RPS?".
3. **Best Practices:** Направлять меня на использование современных практик.NET 8/9 и MassTransit v8+, избегая устаревших подходов (например, ручного управления каналами IModel в RabbitMQ.Client, если MassTransit решает эту задачу лучше).
4. **Визуализация:** Описывать топологию (Exchange -> Binding -> Queue) текстовыми диаграммами (Mermaid или ASCII), чтобы я мог визуализировать потоки данных.

**Формат ответа:**

* Сначала теоретическое обоснование ("Почему мы это делаем").
* Затем пример или критика ("Как это сделать надежно").
* В конце — ссылка на фундаментальный принцип (например, CAP-теорема или Idempotency).

## День 1: Фундаментальная Архитектура Брокеров Сообщений

Цель: Понять физику процесса передачи сообщений, отличия AMQP от HTTP, и развернуть инфраструктуру.

Время: 4 часа.

### 1.1. Смена Ментальной Модели: От Request-Response к Fire-and-Forget

Для разработчика Unity, привыкшего к немедленному вызову методов внутри кадра (Update), переход к асинхронному обмену сообщениями требует фундаментального сдвига восприятия времени и связности. В монолитном приложении или игре вызов NotificationService.SendEmail() происходит синхронно: вызывающий код блокируется (или ожидает await), и результат известен немедленно. Если сервис почты недоступен, ошибка возникает прямо в стеке вызова инициатора.

В распределенной системе такой подход (прямой HTTP-вызов) создает жесткую связность (tight coupling) и временную зависимость (temporal coupling). Если сервис уведомлений перегружен или недоступен, сервис заказов (или календаря) тоже перестает работать.

**Анализ различий моделей взаимодействия:**

| **Характеристика** | **Синхронный вызов (HTTP/Unity Direct Call)** | **Асинхронный месседжинг (RabbitMQ)** |
| --- | --- | --- |
| **Связность (Coupling)** | **Жесткая.** Отправитель должен знать адрес и API получателя. | **Слабая.** Отправитель знает только контракт сообщения и адрес обменника (Exchange). |
| **Доступность** | Оба сервиса должны быть онлайн одновременно. | Получатель может быть оффлайн. Сообщения буферизуются в очереди. |
| **Задержка (Latency)** | Сумма времени обработки всех цепочек вызовов. | Минимальная для отправителя (время отправки в брокер). Обработка происходит фоном. |
| **Масштабируемость** | Ограничена самым медленным сервисом в цепочке. | Балансировка нагрузки через увеличение количества потребителей (Competing Consumers). |
| **Аналогия Unity** | Вызов GetComponent<Health>().TakeDamage() | EventManager.Trigger("UnitDamaged") |

Внедрение брокера сообщений, такого как RabbitMQ, позволяет реализовать паттерн **Store-and-Forward**. Сервис-источник "складирует" намерение (Intent) в надежном хранилище (брокере), а сервис-потребитель забирает его, когда готов. Это обеспечивает **сглаживание нагрузки** (load leveling): резкий всплеск заказов в "Черную пятницу" не положит сервис отправки писем, а лишь увеличит длину очереди, которая будет постепенно разобрана.1

### 1.2. Протокол AMQP 0-9-1 и Архитектура RabbitMQ

RabbitMQ реализует протокол AMQP (Advanced Message Queuing Protocol), который является бинарным протоколом прикладного уровня, оптимизированным для надежности и гибкой маршрутизации. Важно понимать внутреннюю структуру RabbitMQ, чтобы не использовать его как "черный ящик".

**Ключевые компоненты архитектуры:**

1. **Exchange (Обменник):** Почтовое отделение. Продюсер *никогда* не отправляет сообщение напрямую в очередь. Он отправляет его в Exchange. Задача Exchange — принять сообщение и на основе правил (Bindings) маршрутизировать его в одну или несколько очередей.3
2. **Queue (Очередь):** Почтовый ящик. Это буфер, хранящий сообщения. Очереди могут быть персистентными (сохраняться на диск при перезагрузке брокера) или транзитными.
3. **Binding (Привязка):** Правило маршрутизации. Связывает Exchange и Queue.
4. **Routing Key (Ключ маршрутизации):** Атрибут сообщения (строка), который используется Exchange для принятия решения о маршрутизации.

**Типы Обменников (Exchange Types) и их применение:**

* **Direct Exchange:** Сообщение направляется в очередь, если Routing Key *полностью совпадает* с ключом привязки. Используется для точечной доставки (Unicast).
  + *Пример:* Отправка задачи на ресайз конкретного изображения.
* **Fanout Exchange:** Сообщение дублируется во *все* привязанные очереди, игнорируя Routing Key. Это классический Pub/Sub.
  + *Пример:* Событие "Игрок победил босса". Один сервис обновляет лидерборд, другой — начисляет ачивки, третий — отправляет пуш-уведомление.4
* **Topic Exchange:** Маршрутизация по шаблону (wildcard). Routing Key состоит из слов, разделенных точками (напр., logs.error.auth). Поддерживаются символы \* (одно слово) и # (ноль или более слов).
  + *Пример:* Подписка на все ошибки авторизации (\*.error.auth) или на все события конкретного игрока (player.123.#).

### 1.3. RabbitMQ vs Kafka: Выбор инструмента

Часто возникает вопрос: почему RabbitMQ, а не Kafka? В контексте Enterprise-бэкенда для обработки бизнес-транзакций (как в нашем сервисе Календаря) RabbitMQ часто предпочтительнее.

**Сравнительный анализ:**

| **Характеристика** | **RabbitMQ** | **Apache Kafka** |
| --- | --- | --- |
| **Модель данных** | **Очередь (Queue).** Сообщение удаляется после обработки (Ack). | **Лог (Log).** Сообщения хранятся на диске (Retention Policy) и могут быть перечитаны. |
| **Маршрутизация** | **Сложная (Smart Broker).** Гибкие правила маршрутизации внутри брокера. | **Простая (Dumb Broker).** Логика фильтрации часто на клиенте. |
| **Порядок сообщений** | Гарантируется в рамках одной очереди (до рекьюинга). | Гарантируется в рамках партиции (Partition). |
| **Применение** | Сложная бизнес-логика, транзакционность, точечная маршрутизация. | Стриминг больших данных, Event Sourcing, аналитика логов.5 |

Для задачи "отправить уведомление при создании события" RabbitMQ подходит идеально, так как нам нужна гарантия доставки и удаления задачи после выполнения, а не хранение истории всех событий вечно.

### Промпт для исследования (Day 1)

**Запрос к Gemini:** "Проанализируй различия между типами Exchange в RabbitMQ. Для системы уведомлений, где у нас могут быть разные типы уведомлений (Email, SMS, Push) и разные приоритеты (High, Low), какую топологию Exchange/Queue ты бы порекомендовал? Рассмотри вариант использования Topic Exchange для маршрутизации по ключам вида notification.email.high. Объясни, как это соотносится с паттерном Observer в C#."

### Практическое Задание: Инфраструктурный Базис

**Цель:** Развернуть RabbitMQ в Docker и настроить Healthchecks для обеспечения надежного старта зависимых сервисов.

1. Docker Compose Configuration:  
   Создайте файл docker-compose.yml. Критически важно не просто запустить RabbitMQ, но и включить плагин управления (rabbitmq\_management) для визуализации.7  
   YAML  
   version: '3.8'  
   services:  
    rabbitmq:  
    image: rabbitmq:3-management  
    container\_name: rabbitmq  
    ports:  
    - "5672:5672" # AMQP протокол  
    - "15672:15672" # Management UI  
    environment:  
    RABBITMQ\_DEFAULT\_USER: guest  
    RABBITMQ\_DEFAULT\_PASS: guest  
    healthcheck:  
    test:  
    interval: 30s  
    timeout: 10s  
    retries: 5  
    start\_period: 20s
2. **Ручное Исследование Топологии:**
   * Запустите контейнер: docker-compose up -d.
   * Откройте браузер по адресу http://localhost:15672 (Login: guest/guest).
   * Перейдите на вкладку **Exchanges** и создайте новый Fanout exchange с именем test.fanout.
   * Перейдите на вкладку **Queues** и создайте две очереди: queue.email и queue.sms.
   * Вернитесь в Exchange test.fanout и создайте **Bindings** к обеим очередям.
   * Используя функцию **Publish Message** в интерфейсе Exchange, отправьте JSON { "message": "Hello World" }.
   * Убедитесь, что сообщение появилось в *обеих* очередях. Это демонстрирует механизм Pub/Sub, который будет использоваться в коде.

## День 2: Абстракция Шины Данных с MassTransit

Цель: Интегрировать библиотеку MassTransit в.NET проект для абстрагирования от низкоуровневых деталей AMQP.

Время: 4 часа.

### 2.1. MassTransit как "Игровой Движок" для Сообщений

Работа с RabbitMQ напрямую через библиотеку RabbitMQ.Client сравнима с написанием графического движка на чистом DirectX. Это дает полный контроль, но требует ручного управления соединениями, пулами каналов, сериализацией, обработкой разрывов связи и реализацией паттернов восстановления. В Enterprise-разработке стандартом де-факто является использование высокоуровневых абстракций, таких как MassTransit (или NServiceBus).

**MassTransit берет на себя:**

* **Управление топологией:** Автоматическое создание Exchange, Queue и Bindings при старте приложения на основе типов сообщений.
* **Сериализация:** Автоматическая конвертация C# объектов в JSON (и обратно) с добавлением метаданных (Headers) для корректной десериализации (Envelope pattern).
* **Управление жизненным циклом Consumer:** Создание Scope для каждого сообщения, разрешение зависимостей через DI и очистка ресурсов после обработки.
* **Паттерны маршрутизации:** Реализация Request/Response, Saga, Routing Slip "из коробки".8

### 2.2. Контракты Сообщений: Command vs Event

В MassTransit (и в Messaging вообще) существует четкое семантическое разделение типов сообщений, которое влияет на способ их отправки 10:

1. **Command (Команда):**
   * **Семантика:** "Сделай это". Побуждение к действию.
   * **Адресация:** Point-to-Point. Отправляется конкретному получателю (очереди).
   * **Именование:** Глагол в повелительном наклонении + Существительное (напр., SendEmail, CreateOrder).
   * **Метод:** endpoint.Send(command).
   * **Ожидание:** Отправитель ожидает, что действие будет выполнено (или получит ошибку).
2. **Event (Событие):**
   * **Семантика:** "Это произошло". Фактическая констатация изменения состояния.
   * **Адресация:** Publish-Subscribe. Рассылается всем заинтересованным ("вещается").
   * **Именование:** Существительное + Глагол в прошедшем времени (напр., EmailSent, OrderCreated).
   * **Метод:** publishEndpoint.Publish(event).
   * **Ожидание:** Отправитель не знает и не заботится о том, кто и как обработает событие.

**Важно:** Для обеспечения совместимости между микросервисами (Producer и Consumer) контракты сообщений (классы/интерфейсы) должны находиться в общей библиотеке (Shared Library) или иметь идентичные полные имена типов (Namespace + ClassName), так как MassTransit использует URN типов для маршрутизации.8

### 2.3. Конфигурация Consumer в.NET

Consumer в MassTransit — это класс, реализующий интерфейс IConsumer<TMessage>. В отличие от Update() в Unity, который вызывается каждый кадр, метод Consume() вызывается только при поступлении сообщения.

Принципиальное отличие в управлении памятью: в Unity мы избегаем аллокаций в Update. В MassTransit каждый вызов Consume происходит в своем собственном **DI Scope**. Это означает, что все Scoped-сервисы (например, EF Core DbContext) создаются заново для каждого сообщения и уничтожаются после. Это изолирует обработку разных сообщений друг от друга, предотвращая утечки данных и состояния между запросами.

### Промпт для исследования (Day 2)

**Запрос к Gemini:** "Помоги сконфигурировать MassTransit с RabbitMQ для.NET 8.

1. В чем разница между ConfigureEndpoints(context) и ручным объявлением ReceiveEndpoint?
2. Как MassTransit формирует имена очередей по умолчанию, если я использую KebabCaseEndpointNameFormatter?
3. Что такое PrefetchCount в RabbitMQ и как он влияет на пропускную способность моего Consumer? Какое значение оптимально для CPU-bound задач, а какое для IO-bound?"

### Практическое Задание: "Кроликизация Календаря" (Часть 1)

**Контекст:** Сервис Календаря (разработанный на предыдущих неделях) должен перестать отправлять уведомления сам и начать делегировать это через события.

1. Shared Contracts:  
   Создайте проект Class Library Calendar.Contracts. Определите событие:  
   C#  
   public record EventCreated(Guid EventId, string Title, string UserEmail, DateTime StartTime);
2. **Calendar Service (Producer):**
   * Подключите MassTransit.RabbitMQ.
   * В Program.cs настройте Producer:  
     C#  
     builder.Services.AddMassTransit(x => {  
      x.UsingRabbitMq((context, cfg) => {  
      cfg.Host("rabbitmq://localhost", h => {  
      h.Username("guest");  
      h.Password("guest");  
      });  
      });  
     });
   * Внедрите IPublishEndpoint в контроллер создания событий. Вместо логики уведомления вызовите:  
     C#  
     await \_publishEndpoint.Publish(new EventCreated(...));
3. **Notification Service (Consumer):**
   * Создайте новый проект Worker Service или Web API.
   * Реализуйте Consumer:  
     C#  
     public class NotificationConsumer : IConsumer<EventCreated>  
     {  
      private readonly ILogger<NotificationConsumer> \_logger;  
      public NotificationConsumer(ILogger<NotificationConsumer> logger) => \_logger = logger;  
       
      public Task Consume(ConsumeContext<EventCreated> context)  
      {  
      \_logger.LogInformation("Received event: {Title} for {Email}",   
      context.Message.Title, context.Message.UserEmail);  
      return Task.CompletedTask;  
      }  
     }
   * Зарегистрируйте Consumer в DI и настройте ConfigureEndpoints в Program.cs.
4. **Verification:** Запустите оба сервиса. Создайте событие через Swagger сервиса Календаря. Убедитесь, что сервис Уведомлений вывел лог. Проверьте в RabbitMQ Management UI, что Exchange Calendar.Contracts:EventCreated создан и имеет Binding к очереди notification-service.

## День 3: Паттерны Надежности и Устойчивости (Resilience)

Цель: Реализовать механизмы обработки сбоев, гарантирующие, что временные проблемы сети или сервисов не приведут к потере данных.

Время: 4 часа.

### 3.1. Природа Сбоев в Распределенных Системах

В распределенных системах сбои — это не исключение, а норма. Сеть может "моргать", база данных может перезагружаться, SMTP-сервер может отвергать соединения из-за лимитов.

Сбои делятся на:

* **Transient (Временные):** Краткосрочные проблемы (timeout, deadlock, network glitch). Лечатся повторной попыткой (Retry).
* **Persistent (Постоянные):** Ошибки логики (NullReference), невалидные данные. Повтор не поможет. Требуют вмешательства человека или исправления бага.

### 3.2. Паттерн Retry (Повторная попытка)

Паттерн Retry позволяет автоматически перезапускать обработку сообщения при возникновении исключения. MassTransit предоставляет мощный middleware для настройки политик повторов 13:

* **Immediate:** Повторить сразу. Подходит для очень коротких сбоев, но может создать "шторм запросов" (Retry Storm) на уже перегруженный сервис.
* **Intervals:** Повторить с фиксированной паузой (напр., каждые 500мс).
* **Exponential Backoff (Экспоненциальная задержка):** Увеличивать паузу с каждой попыткой (1с, 2с, 4с, 8с). Это наиболее "вежливая" стратегия, дающая сбойной системе время на восстановление.

**Jitter (Дрожание):** Важно добавлять случайность в интервалы повторов, чтобы избежать синхронизации множества клиентов, одновременно атакующих восстановившийся сервис (Thundering Herd problem).

### 3.3. Poison Messages и Dead Letter Queue (DLQ)

Если сообщение не удалось обработать даже после всех попыток Retry (например, исчерпан лимит в 5 попыток), оно считается "ядовитым" (Poison Message). Бесконечные повторы заблокировали бы очередь и потребили бы все ресурсы CPU (эту ситуацию называют "poison pill" — отравленная таблетка).

MassTransit автоматически перемещает такие сообщения в специальную очередь ошибок, именуемую \_error (или \_skipped при отсутствии подходящего обработчика). Это позволяет системе продолжить обработку здоровых сообщений. Сообщения из DLQ затем могут быть проанализированы разработчиками и, при необходимости, возвращены в основную очередь (Shovel) после исправления бага.14

### 3.4. Circuit Breaker (Предохранитель)

Если сервис уведомлений (SMTP) полностью лежит, нет смысла даже пытаться отправлять письма и тратить ресурсы на Retries для каждого из 1000 сообщений в секунду.

Паттерн Circuit Breaker отслеживает статистику ошибок. Если процент ошибок превышает порог, "предохранитель выбивает", и сервис переходит в состояние Open: он немедленно возвращает ошибку без реальной попытки выполнить действие. Через некоторое время он переходит в Half-Open, пропуская один тестовый запрос. Если он успешен — цепь замыкается (Closed). Это предотвращает каскадные сбои всей системы.16

### Промпт для исследования (Day 3)

**Запрос к Gemini:** "Объясни, как настроить UseMessageRetry и UseDelayedRedelivery в MassTransit.

1. В чем фундаментальная разница между Retry (in-memory) и Redelivery (через планировщик/RabbitMQ)? Почему для длительных сбоев (например, сервис лежит 10 минут) нельзя использовать обычный Retry?
2. Как сконфигурировать фильтрацию исключений? Я хочу повторять попытки при TimeoutException, но не при ArgumentException.
3. Приведи пример конфигурации Circuit Breaker для защиты внешнего SMTP сервиса."

### Практическое Задание: "Стресс-тест Уведомлений"

1. Эмуляция Нестабильности:  
   В NotificationConsumer добавьте код, который с вероятностью 70% выбрасывает исключение:  
   C#  
   var rand = new Random();  
   if (rand.Next(1, 100) <= 70)   
    throw new InvalidOperationException("SMTP server temporary failure");
2. Настройка Retry Policy:  
   В конфигурации MassTransit (Program.cs) добавьте политику повторов для данного endpoint:  
   C#  
   cfg.ReceiveEndpoint("notification-service", e =>  
   {  
    e.UseMessageRetry(r => r.Exponential(5, TimeSpan.FromSeconds(1), TimeSpan.FromSeconds(10), TimeSpan.FromSeconds(2)));  
    e.ConfigureConsumer<NotificationConsumer>(context);  
   });  
     
   Это означает: 5 попыток, начальная задержка 1с, максимальная 10с, шаг роста 2с.
3. Наблюдение:  
   Запустите сервис и отправьте сообщение. Следите за логами. Вы увидите несколько записей "Received event..." с паузами, и в конце (если не повезет) сообщение о перемещении в \_error очередь.  
   Проверьте RabbitMQ Management UI: найдите очередь notification-service\_error и инспектируйте содержимое "мертвого" сообщения.

## День 4: Гарантированная Согласованность (Transactional Outbox)

Цель: Решить проблему дуальной записи и обеспечить атомарность бизнес-операций и отправки сообщений.

Время: 4 часа.

### 4.1. Проблема Dual Write (Двойная Запись)

Представим сценарий в CalendarService:

1. Пользователь создает встречу.
2. Сервис сохраняет встречу в PostgreSQL (dbContext.SaveChangesAsync()).
3. Сервис публикует EventCreated в RabbitMQ.

**Точки отказа:**

* Если упадет шаг 2: Транзакция откатится, сообщение не отправится. Все хорошо.
* Если шаг 2 успешен, но приложение крашнется или сеть пропадет *перед* шагом 3: Встреча создана в БД, но уведомление (сообщение) не отправлено. **Система несогласована.**

Попытка поменять местами (сначала RabbitMQ, потом БД) не решает проблему, а создает фантомные уведомления о несозданных встречах. Распределенные транзакции (2PC/XA) слишком медленны и сложны для микросервисов.

### 4.2. Решение: Паттерн Transactional Outbox

Решение заключается в использовании ACID-транзакции базы данных для *обоих* действий. Мы не отправляем сообщение в брокер сразу. Вместо этого мы записываем намерение отправить сообщение в специальную таблицу OutboxMessage в той же базе данных.17

**Алгоритм Outbox:**

1. Начать транзакцию БД.
2. INSERT в таблицу Events (бизнес-данные).
3. INSERT в таблицу OutboxMessage (сериализованное сообщение для RabbitMQ).
4. Закоммитить транзакцию (COMMIT).

Теперь данные согласованы атомарно. Либо есть и событие, и сообщение, либо нет ничего.

Отдельный фоновый процесс (Outbox Processor, встроен в MassTransit) мониторит таблицу OutboxMessage, читает сообщения, отправляет их в RabbitMQ и, после получения подтверждения (Ack) от брокера, удаляет их из таблицы.

### 4.3. Inbox Pattern и Идемпотентность

Outbox гарантирует доставку хотя бы один раз (At-least-once). Если процесс упадет после отправки в RabbitMQ, но до удаления из таблицы Outbox, сообщение будет отправлено повторно.

Следовательно, Consumer должен быть защищен от дублей. Паттерн Inbox хранит ID всех обработанных сообщений в базе данных консумера. Перед обработкой проверяется, не был ли этот ID уже обработан. MassTransit делает это прозрачно при использовании Transactional Outbox на стороне потребителя.19

### Промпт для исследования (Day 4)

**Запрос к Gemini:** "Объясни детали реализации Transactional Outbox в MassTransit с Entity Framework Core.

1. Какие таблицы (InboxState, OutboxMessage, OutboxState) создаются и за что они отвечают?
2. Как настроить UseBusOutbox vs UseInMemoryOutbox?
3. Влияет ли Outbox на производительность базы данных? Нужно ли создавать индексы для таблицы OutboxMessage при высокой нагрузке?"

### Практическое Задание: "Атомарный Календарь"

1. Интеграция EF Core Outbox:  
   В проекте CalendarService добавьте пакет MassTransit.EntityFrameworkCore.  
   В конфигурации AddMassTransit:  
   C#  
   x.AddEntityFrameworkOutbox<CalendarDbContext>(o =>  
   {  
    o.UsePostgres(); // Или UseSqlServer  
    o.UseBusOutbox(); // Включение Outbox для IBus  
   });  
     
   Убедитесь, что CalendarDbContext наследует необходимые конфигурации моделей MassTransit (через modelBuilder.AddInboxStateEntity(), AddOutboxMessageEntity() и т.д. в OnModelCreating).
2. Миграции:  
   Создайте и примените миграцию EF Core. В базе данных должны появиться таблицы MassTransit\_OutboxMessage и др..18
3. **Тестирование Отказоустойчивости (Chaos Engineering):**
   * Остановите контейнер RabbitMQ (docker stop rabbitmq).
   * Вызовите API создания события.
   * Убедитесь, что запрос завершился успешно (200 OK). Данные записаны в БД и в таблицу Outbox.
   * Запустите RabbitMQ (docker start rabbitmq).
   * Наблюдайте в логах, как MassTransit обнаружит соединение и "дошлет" накопившиеся сообщения из таблицы Outbox.

## День 5: Тестирование и Observability (Наблюдаемость)

Цель: Научиться тестировать асинхронный код без поднятия тяжелой инфраструктуры и внедрить трейсинг для отладки распределенных процессов.

Время: 4 часа.

### 5.1. Unit-тестирование с TestHarness

Тестирование асинхронных событий сложно. Использование Thread.Sleep в тестах делает их медленными и нестабильными (flaky).

MassTransit предоставляет InMemoryTestHarness — специальную реализацию транспорта, которая работает полностью в памяти, но эмулирует поведение брокера. Она позволяет "перематывать время" и проверять факты публикации и потребления сообщений.20

**Что мы проверяем:**

* **Producer:** Действительно ли контроллер опубликовал сообщение EventCreated?
* **Consumer:** Действительно ли консюмер получил сообщение и вызвал зависимый сервис (например, IEmailSender)? Попало ли сообщение в \_error очередь при исключении?

### 5.2. Observability: Distributed Tracing

В монолите для отладки достаточно Stack Trace. В микросервисах запрос проходит через множество узлов. OpenTelemetry — это стандарт для сбора телеметрии.

Каждое сообщение в MassTransit помечается заголовками ConversationId и MessageId. При интеграции с OpenTelemetry (ActivitySource), можно построить граф (Trace), показывающий путь запроса:

HTTP POST /event (Calendar) -> DB Insert -> RabbitMQ Publish -> RabbitMQ Consume (Notification) -> SMTP Send.

Это позволяет находить узкие места (latency) и понимать причины ошибок, видя полную картину.22

### Промпт для исследования (Day 5)

**Запрос к Gemini:** "Покажи пример Unit-теста для MassTransit Consumer с использованием xUnit и MassTransit.Testing.

1. Как использовать harness.Bus.Publish в тесте?
2. Как проверить harness.Consumed.Select<EventCreated>().Any()?
3. Зачем нужен await harness.Start() и нужно ли его останавливать в Dispose?
4. Как подключить OpenTelemetry к MassTransit и вывести трейсы в консоль (ConsoleExporter) для отладки?"

### Практическое Задание: "Зеленые Тесты"

1. Unit Tests:  
   Создайте тестовый проект NotificationService.Tests (xUnit).  
   Напишите тест для NotificationConsumer:  
   C#  
   [Fact]  
   public async Task Notification\_Should\_Be\_Sent\_When\_Event\_Created()  
   {  
    // Arrange  
    var harness = new InMemoryTestHarness();  
    var consumerHarness = harness.Consumer<NotificationConsumer>();  
    await harness.Start();  
     
    try   
    {  
    // Act  
    await harness.Bus.Publish(new EventCreated { Title = "Test Event" });  
     
    // Assert  
    // Ждем, пока сообщение будет потреблено (без Thread.Sleep!)  
    Assert.True(await harness.Consumed.Select<EventCreated>().Any());  
    Assert.True(await consumerHarness.Consumed.Select<EventCreated>().Any());  
    }  
    finally  
    {  
    await harness.Stop();  
    }  
   }
2. Observability (Опционально):  
   Подключите OpenTelemetry.Extensions.Hosting и OpenTelemetry.Instrumentation.MassTransit. Настройте вывод трейсов в консоль и запустите систему. Посмотрите, как выглядит трейс при прохождении сообщения через RabbitMQ.24

## Итоговые Задания Недели

**Проект: "Кроликизация Календаря: Enterprise Edition"**

Легенда:

Ваш стартап "Календарь" вырос. Прямые вызовы сервисов стали узким местом. Бизнес требует гарантии, что ни одно уведомление о встрече не будет потеряно, даже если дата-центр сгорит.

**Техническое Задание:**

1. **Архитектура:**
   * Два независимых сервиса: Calendar.API (Producer) и Notification.Worker (Consumer).
   * Общая библиотека Calendar.Contracts.
   * Инфраструктура в Docker Compose: PostgreSQL, RabbitMQ (Management Plugin).
2. **Calendar.API (Требования):**
   * Использовать **Transactional Outbox** (EF Core) для публикации событий EventCreated. Публикация должна быть атомарна с сохранением события в БД.
   * Настроить MassTransit на использование RabbitMQ.
3. **Notification.Worker (Требования):**
   * Реализовать **Idempotent Consumer**. Обработка одного и того же MessageId дважды не должна приводить к дублированию логики (использовать Inbox pattern или проверку по БД).
   * Настроить **Retry Policy**: 3 попытки с экспоненциальной задержкой.
   * Эмулировать работу с SMTP (просто Logger.LogInformation с задержкой Task.Delay).
4. **Сценарии Сбоев (Chaos Testing):**
   * В README.md описать эксперимент: "Что происходит, если я выключаю RabbitMQ, создаю событие, а потом включаю RabbitMQ?". Приложить скриншоты логов, подтверждающих отложенную доставку.
   * Описать эксперимент: "Что происходит, если Consumer выбрасывает исключение?". Показать, как сообщение попадает в \_error очередь.
5. **Тесты:**
   * Минимум один Unit-тест с InMemoryTestHarness, проверяющий успешную обработку сообщения.

**Критерии Оценки:**

* Код компилируется и запускается через docker-compose up.
* Отсутствуют синхронные HTTP-вызовы между сервисами.
* При падении RabbitMQ данные не теряются.
* Код Consumer написан с учетом идемпотентности.
* Используются record для DTO сообщений.

## Заключение

Переход на асинхронную модель взаимодействия — это качественный скачок в развитии backend-разработчика. Освоенные на этой неделе паттерны (Pub/Sub, Outbox, Retry, DLQ) являются "золотым стандартом" построения микросервисов. В отличие от предсказуемого игрового цикла Unity, распределенные системы хаотичны и ненадежны. Ваша задача как архитектора — построить надежную систему поверх ненадежных компонентов. Использование MassTransit и RabbitMQ предоставляет необходимый инструментарий для решения этой задачи, превращая хаос распределенной сети в упорядоченный поток бизнес-событий.

#### Источники

1. Most Common RabbitMQ Use Cases - ScaleGrid, дата последнего обращения: декабря 6, 2025, <https://scalegrid.io/blog/rabbitmq-use-cases/>
2. How Should RabbitMQ Be Used in Software Architecture? | by SOORAJ. V - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@v4sooraj/how-should-rabbitmq-be-used-in-software-architecture-e741ddee7ab3>
3. .NET Core and RabbitMQ Part 2 — Communication via AMQP | by George Dyrrachitis | The Startup | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/swlh/net-core-and-rabbitmq-part-2-communication-via-amqp-35c5518cb64>
4. AMQP 0-9-1 Model Explained - RabbitMQ, дата последнего обращения: декабря 6, 2025, <https://www.rabbitmq.com/tutorials/amqp-concepts>
5. What's the Difference Between Kafka and RabbitMQ? - AWS, дата последнего обращения: декабря 6, 2025, <https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-kafka/>
6. RabbitMQ vs Apache Kafka: Architectural and Conceptual Differences - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@steffankharmaaiarvi/rabbitmq-vs-apache-kafka-architectural-and-conceptual-differences-37f986a8d5f5>
7. Docker-Compose for Asp.Net Core & RabbitMQ | by Ali Yıldızöz | DevOps.dev, дата последнего обращения: декабря 6, 2025, <https://blog.devops.dev/docker-compose-for-asp-net-core-rabbitmq-d532125b1cec>
8. Using MassTransit with RabbitMQ in ASP.NET Core - Code Maze, дата последнего обращения: декабря 6, 2025, <https://code-maze.com/masstransit-rabbitmq-aspnetcore/>
9. MassTransit — What,Why,How. MassTransit is a powerful open-source… | by Ljscodex | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@ljscodex/masstransit-what-why-how-f43364994c10>
10. Producers - MassTransit, дата последнего обращения: декабря 6, 2025, <https://masstransit.io/documentation/concepts/producers>
11. Send, Publish and Request/Response in MassTransit - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/69123032/send-publish-and-request-response-in-masstransit>
12. Requests - MassTransit, дата последнего обращения: декабря 6, 2025, <https://masstransit.io/documentation/concepts/requests>
13. MassTransit in ASP.NET Core: A Practical Guide to Event-Driven .NET - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/stevsharp/masstransit-in-aspnet-core-a-practical-guide-to-event-driven-net-c4k>
14. Exceptions - MassTransit, дата последнего обращения: декабря 6, 2025, <https://masstransit.io/documentation/concepts/exceptions>
15. Dead Letter Exchanges - RabbitMQ, дата последнего обращения: декабря 6, 2025, <https://www.rabbitmq.com/docs/dlx>
16. RabbitMQ and MassTransit in .NET Core : Practical Guide - HAMED SALAMEH, дата последнего обращения: декабря 6, 2025, <https://hamedsalameh.com/rabbitmq-and-masstransit-in-net-core-practical-guide/>
17. Learning - MassTransit Transactional Outbox Pattern - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/hootanht/learning-masstransit-transactional-outbox-pattern-g4c>
18. Transactional Outbox Configuration - MassTransit, дата последнего обращения: декабря 6, 2025, <https://masstransit.io/documentation/configuration/middleware/outbox>
19. Implementing the Outbox and Inbox Pattern with MassTransit: A Reliable Messaging Approach in .NET | by Fatih çakıroğlu | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@fcakiroglu16/implementing-the-outbox-and-inbox-pattern-with-masstransit-a-reliable-messaging-approach-in-net-52e943f6826d>
20. Testing - MassTransit, дата последнего обращения: декабря 6, 2025, <https://masstransit.io/documentation/concepts/testing>
21. Unit Testing Consumers in .NET Core with MassTransit | by Ranushka Pasindu | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@rdharmaranga/unit-testing-consumers-in-net-core-with-masstransit-a-comprehensive-guide-a9ac4245c655>
22. Observability - MassTransit, дата последнего обращения: декабря 6, 2025, <https://masstransit.io/documentation/configuration/observability>
23. Introduction to Distributed Tracing With OpenTelemetry in .NET - Milan Jovanović, дата последнего обращения: декабря 6, 2025, <https://www.milanjovanovic.tech/blog/introduction-to-distributed-tracing-with-opentelemetry-in-dotnet>
24. Getting started with OpenTelemetry and distributed tracing in .NET - my tech ramblings, дата последнего обращения: декабря 6, 2025, <https://www.mytechramblings.com/posts/getting-started-with-opentelemetry-and-dotnet-core/>