Задание Модульбанк стажировка C# #4

**Цель задания:**

Освоить взаимодействие между микросервисами на основе событий: подключить RabbitMQ (можно Kafka, но везде ниже будем говорить RabbitMQ), реализовать шаблон Transactional Outbox и обеспечить надежную доставку доменных событий из сервиса «Счета» без потери и дублей.

**Крайний срок сдачи задания:**

9:00 – 19 августа 2025

**Описание бизнес задачи:**

Мы продолжаем развивать микросервис «Счета» (Account Service) розничного банка. На этом этапе сервис должен публиковать и (частично) потреблять доменные события, чтобы другие системы банка могли реагировать на изменения в счетах и транзакциях.

На данном этапе реализации сервиса мы хотим реализовать следующие пользовательские истории:

* Я, как менеджер банка Анна, открыла клиенту Ивану бесплатный текущий счёт, чтобы он мог хранить средства клиенту — система опубликовала событие AccountOpened, чтобы CRM получила данные о счёте.
* Я, как менеджер банка Анна, открыла клиенту Ивану срочный вклад «Надёжный‑6» под 3 % годовых, чтобы он смог накопить средства.
* Я, как кассир банка Алексей, пополнил текущий счёт клиента Ивана на 1 000 рублей наличными. — система опубликовала событие MoneyCredited для сервиса уведомлений.
* Я, как клиент банка Иван, перевёл 200 рублей со своего текущего счёта на вклад «Надёжный‑6», чтобы пополнить вклад.— система гарантированно опубликовала TransferCompleted, даже если RabbitMQ был временно недоступен.
* Я, как менеджер банка Анна, выдал клиенту Ивану выписку по его счетам.
* Я, как клиент банка Иван, запросил баланс своего текущего счёта.
* Я, как менеджер банка Анна, закрыла срочный вклад клиента Ивана «Надёжный‑6» и начислила проценты, чтобы завершить договор.
* Я, как система, каждый день начисляю проценты по вкладу, чтобы поддерживать актуальное состояние счета по вкладу.— по завершении операции публикуется InterestAccrued.
* Я, как сотрудник antifraud, временно блокирую клиента — сервис «Счета» получает событие ClientBlocked и запрещает новые расходные операции (в рамках задания — через RMQ‑заглушку).

**События и контракты (минимальный набор для задания):**

Публикуемые события:

* AccountOpened { eventId, occurredAt, accountId, ownerId, currency, type }
* MoneyCredited { eventId, occurredAt, accountId, amount, currency, operationId }
* MoneyDebited { eventId, occurredAt, accountId, amount, currency, operationId, reason }
* TransferCompleted { eventId, occurredAt, sourceAccountId, destinationAccountId, amount, currency, transferId }
* InterestAccrued { eventId, occurredAt, accountId, periodFrom, periodTo, amount }

Потребляемые события (через отдельный consumer внутри сервиса «Счета»):

* ClientBlocked { eventId, occurredAt, clientId } — отмечаем все его счета как “Frozen” (flag), запрещаем дебетовые операции.
* ClientUnblocked { eventId, occurredAt, clientId } — снимаем флаг “Frozen”.

Транспорт: RabbitMQ.

Формат сообщений: JSON (UTF‑8).

Версионирование: поле meta.version (строка), обратная совместимость обязательна.

Корреляция: в заголовках RMQ передаём X-Correlation-Id и X-Causation-Id (GUID).

**Техническая задача:**

1. Обновить файл README.md в корне решения с кратким описанием назначения сервиса и пошаговой инструкцией запуска (локально и в Docker), с учетом добавленной функциональности. Инструкции должно быть достаточно для запуска на windows машине с установленным докером.

2. Добавить RabbitMQ в docker‑compose. Объявить exchange account.events (тип topic). Настроить очереди: account.crm (routing key account.\*), account.notifications (money.\*), account.antifraud (client.\*), account.audit (например, #).

3. Реализовать Transactional Outbox в сервисе «Счета» для всех командных хендлеров, которые меняют состояние и должны породить событие.

Гарантии доставки: минимум «at‑least‑once» из Outbox; на стороне потребителя реализовать идемпотентность (Inbox): таблица inbox\_consumed (message\_id UUID PK, processed\_at, handler), проверять перед обработкой.

6. Описать и реализовать топологию маршрутизации:

* account.events/account.opened → routing key account.opened;
* money.credited/debited/transfer.completed → money.\*;
* antifraud.client.blocked/unblocked → client.\*.

7. Добавить структурированное логирование

* публикаций/потребления (eventId, type, correlationId, retry, latency).
* логи входящих/исходящих HTTP запросов.

8. Интеграционные тесты:

• OutboxPublishesAfterFailure — имитируем временную недоступность RMQ, убеждаемся, что событие публикуется после восстановления;

• ClientBlockedPreventsDebit — после получения события ClientBlocked попытка списания возвращает 409 Conflict, событие MoneyDebited не публикуется.

9. Обновить Swagger: добавить раздел «События» с документацией по контрактам (описание полей и примеры JSON).

10. Безопасность: все бизнес HTTP методы — только по валидному JWT‑токену. События не должны содержать персональные данные сверх необходимого.  
Исключения: /health/live и /health/ready — AllowAnonymous;

11. Качество кода: 0 ошибок Resharper; структура проекта — vertical slice (Features\<Feature>\<Operation>).

12. Наблюдаемость: добавить endpoint /health/live и /health/ready; health‑проверка для подключения RabbitMQ и отставания Outbox (например, WARN при >100 непубликованных сообщений).

13. Docker: сделать named volume для данных Postgres и отдельный для RabbitMQ (персистентность); порты API :80, RabbitMQ :5672, management :15672.

14. Карантин входящих сообщений (валидация envelope/версии)

* Таблица inbox\_dead\_letters (message\_id, received\_at, handler, payload, error).
* При ошибке envelope/версии → записать в карантин, залогировать WARN, ACK (или NACK без requeue при наличии DLX), бизнес-логику не выполнять.

15. Потребители внутри сервиса «Счета»

* Обязательный: AntifraudConsumer (очередь account.antifraud, ключ client.#) — обрабатывает ClientBlocked и ClientUnblocked (ставит/снимает Frozen).
* Опциональный (для тестов/аудита): AuditConsumer (очередь account.audit, ключ #) — пишет принятые события в таблицу audit\_events (можно использовать тот же inbox\_consumed).
* Рекомендации: отдельный канал RMQ на consumer; BasicQos(prefetch: 1); ACK после коммита БД.

17. Структурированные логи публикаций/потребления: eventId, type, correlationId, retry, latency

**Критерии приемки**:

1. После выполнения шагов из README сервис поднимается через docker‑compose; доступен Swagger на http://localhost/ и RabbitMQ UI на http://localhost:15672 (логин/пароль из README).
2. При создании счёта публикуется событие AccountOpened;   
   при пополнении — MoneyCredited;   
   при переводе — TransferCompleted;  
    при начислении процентов — InterestAccrued.
3. Если RabbitMQ недоступен во время перевода, запись остаётся в outbox и публикуется позже без потери данных.
4. На стороне consumer реализована идемпотентность: повторная доставка одного и того же messageId не ведёт к повторной бизнес‑обработке.
5. Тест OutboxPublishesAfterFailure проходит на «чистой» машине (dotnet test); тест TransferEmitsSingleEvent подтверждает ровно 50 событий при 50 переводах.
6. Все HTTP методы требуют валидного JWT; при отсутствии — 401 Unauthorized.
7. Resharper → Inspect → Code Issues In Solution — 0 ошибок/варнингов (допустимы подавления с пояснением).
8. Логи содержат correlationId и eventId для каждой публикации/обработки; присутствуют HEALTH‑пробы.
9. Структура проекта соответствует рекомендациям vertical slice; Swagger содержит описания контрактов событий и примеры.
10. Данные Postgres и RMQ сохраняются между перезапусками контейнеров.
11. ClientBlocked блокирует дебетовые операции; ClientUnblocked — снимает блокировку.
12. Сообщения без meta.version или с неподдерживаемой версией **не** меняют состояние и попадают в inbox\_dead\_letters (лог WARN).

**Приложение 1. Регламент обмена сообщениями**

· Формат: JSON, UTF‑8.

· Обязательные поля payload: eventId (GUID), occurredAt (UTC), meta { version, source, correlationId, causationId }.

· Именование событий: <Aggregate><Action> в PascalCase (например, TransferCompleted).

· Версионирование: meta.version = "v1", при несовместимых изменениях выпускать v2, поддерживать параллельно.

· Маршрутизация: topic exchange account.events; routing key по схеме <domain>.<event> (например, money.transfer.completed).

· Повторная доставка: допускается; потребители обязаны быть идемпотентны и фиксировать обработанные messageId.

· Секьюрность: запрещено публиковать лишние ПДн; вместо ФИО — идентификаторы.

**Приложение 2. Важные замечания по реализации**

· Outbox и изменение агрегата — только в одной БД‑транзакции.

· Dispatcher должен быть устойчив к падениям: безопасная повторная публикация, обработка дедупликации на стороне потребителя.

· Рекомендуется использовать Retry с экспоненциальной паузой и джиттером; при достижении MAX\_RETRIES — статус dead‑letter, алерт в логах.

· Для диспетчера допустимо использовать Hangfire RecurringJob (например, \*/10 \* \* \* \* \*), либо BackgroundService с таймером.

· Любые прямые вызовы IModel.BasicPublish из бизнес‑хендлеров запрещены

· Для упрощения, консьюмеров “заглушки” можно создавать как хостед сервисы в вашем же сервисе Accounts

**Приложение 3. Замечания по оформлению**

· Формат коммитов task4: <Название коммита>

**Приложение 4: Рекомендации по работе с сообщениями**

4.1 Рекомендуемая оболочка для событий  
  
{  
eventId: GUID

occurredAt: string (ISO-8601, UTC, оканчивается на "Z")  
  
payload:

meta: {

* version: string — в этом задании всегда "v1"
* source: string — идентификатор источника (например, "account-service")
* correlationId: GUID
* causationId: GUID  
    
  }

4.2 Заголовки RMQ (Message Properties/Headers):

* X-Correlation-Id и X-Causation-Id дублируют значения из meta.

4.3 Версионирование (meta.version):

* На публикации всегда ставим "v1".
* На потреблении валидируем envelope; поддерживаем только "v1".  
   Неподдержанная/отсутствующая версия → карантин (см. ниже), без применения бизнес-логики.

## 4.4 Транспорт и формат

RabbitMQ: topic exchange — account.events

Сообщения: JSON (UTF-8)  
4.5 пример JSON (envelope + payload)

AccountOpened (routing key = account.opened)

{

"eventId": "b5f3a7f6-2f4e-4b1a-9f3a-2b0c1e7c1a11",

"occurredAt": "2025-08-12T21:00:00Z",

"meta": {

"version": "v1",

"source": "account-service",

"correlationId": "11111111-1111-1111-1111-111111111111",

"causationId": "22222222-2222-2222-2222-222222222222"

},

"accountId": "9c3f3f5d-7c2e-4a1a-9f5a-1b3a7e9d2f11",

"ownerId": "2a7e9d2f-9f5a-4a1a-7c2e-9c3f3f5d1b3a",

"currency": "RUB",

"type": "Checking"

}

**Приложение 5: Хороший цикл статей по RMQ**

на мой взгляд, хотя и не завершенный.   
  
<https://habr.com/ru/articles/488654/>