Конструирование программного обеспечения

Домашняя работа № 4 Асинхронное межсервисное взаимодействие

Шопоголикам нужна ваша помощь!

С наступлением волшебного сезона распродаж, подарков и корпоративов миллионы людей уже стоят в онлайн-очередях за своими мечтами — кто за новым ноутбуком, кто за свитером с оленями. Сервера греются сильнее, чем глинтвейн в чашке, а нагрузка растёт быстрее, чем цены на доставку курьером “до 31 декабря”.

В это непростое для IT-вселенной время на плечи разработчиков и архитекторов ложится святой долг — уберечь интернет-магазины от падений и багов в самый ответственный момент. Ведь любой таймаут при оплате — это не просто ошибка, а трагедия под ёлкой.

Вся надежда на вас! Ваш чистый код — антидот от жабы, которая шепчет "отложи до января".

Техническое задание

Вы являетесь частью команды разработки интернет-магазина «Ozon», который ожидает значительный рост трафика в преддверии Нового года. Ваша задача — разработать два ключевых микросервиса, отвечающих за работу с оплатами (Payments Service) и за работу с заказами (Orders Service).

В Payments Service пользователю должен быть доступен функционал:

1. Создания счета (не более одного счета на каждого пользователя).
2. Пополнения счета.
3. Просмотра баланса счета.

В Orders Service пользователю должен быть доступен функционал:

1. Создания заказа (создание заказа должно асинхронно запускать процесс оплаты заказа).
2. Просмотр списка заказов.
3. Просмотр статуса отдельного заказа.

Предполагается, что вы разрабатываете свою систему в рамках готовой инфраструктуры компании и в каждом запросе вам приходит user_id пользователя, сделавшего запрос (при тестировании системы указывайте user_id самостоятельно). При реализации процессов, связанных с оплатой, требуется обеспечить гарантии доставки и обработки сообщений. При списании денег за заказ необходимо обеспечить семантику effectively exactly once — даже если событие дублируется, деньги должны списываться не более одного раза за заказ. Реализация может быть основана на использовании Transactional Inbox + Outbox и идемпотентной логики обработки сообщений. Стоит учесть, что при параллельных операциях над счетом не должно возникать коллизий и учет денег на балансе не должен «ломаться». Этого можно достичь атомарной инструкцией (Compare and Swap) над счетчиком баланса, либо отслеживать баланс через цепочку транзакций. Остатки на складе, в данном релизе, брать во внимание не нужно. Рассматривается исключительно заказ и успешность / неуспешность его оплаты.

Все сценарии (кроме сценария создания заказа) синхронные. Последовательность действий сценария создания заказа описана на схеме ниже (Пользовательские сценарии).

Архитектурный комитет предлагает следующее решение с четким разделением ответственности микросервисов (вы можете предложить свой вариант):

1. API Gateway – отвечает только за routing запросов.
2. Order Service – отвечает за создание заказа, просмотра списка заказов и просмотра статуса заказа.
3. Payments Service – отвечает за создание, пополнение и просмотра баланса счета.

Критерии оценки

Требования до 8 баллов

1. Реализация основных требований к функциональности – **2 балла**
2. Архитектурное проектирование – **5 баллов**
   a. Четкое разделение на сервисы (Order Service, Payments Service).
   b. Логичное использование очередей сообщений (например, RabbitMQ, Kafka). Доставка сообщений должна обеспечивать at-least-once, при этом семантика exactly-once достигается благодаря идемпотентной обработке.
   c. Применение паттернов:
      - Transactional Outbox в Order Service.
      - Transactional Inbox и Outbox в Payments Service.
      - Обеспечение семантики exactly once при списании денег у пользователя
   d. Транспорт между микросервисами:
      - Разрешено использовать любой брокер сообщений (Kafka, RabbitMQ и др.) с поддержкой at-least-once доставки.
      - Важно, чтобы каждый сервис мог сохранять/чекпоинтить offset (в случае Kafka) или отслеживать уникальность сообщений (в случае RabbitMQ), обеспечивая идемпотентность бизнес-операций.
3. Реализация коллекции Postman / Swagger, которая должна демонстрировать функциональность реализованных микросервисов, охватывая все API – **0,5 балл**
4. Корректность Dockerfile и docker-compose.yml – **0,5 балл**
   a. Все микросервисы должны быть упакованы в Docker-контейнеры.
   b. Вся система должна разворачиваться с помощью docker-compose.yml
   c. Работоспособность всей системы после запуска docker compose up

Требования до 10 баллов

Должны быть выполнены все предыдущие критерии. Можно выполнить одно из двух, а можно оба пункта

I. Реализация фронтенд-части приложения – **2 балл**
   - Фронтенд должен быть реализован как отдельный сервис, взаимодействующий с бэкенд-микросервисами через REST API или GraphQL.
   - Фронтенд может быть реализован в виде десктопного, мобильного или веб-приложения.
   - Допускается использование любых современных фреймворков.
   - Фронтенд должен быть упакован в Docker-контейнер и запускаться через Docker Compose вместе с остальными сервисами.
   - Фронтенд не является основной частью работы, можно ограничиться минимально-достаточным интерфейсом в виде веб-приложения, но можно написать десктопный и/или мобильный клиентов с архитектурой (например WPF приложение с применением MVVM).

II. Реализация реального отслеживания состояния заказа через WebSocket + уведомления (push) – 2 балл

   Технологии:
   - Бэкенд: WebSocket (Spring WebSocket, Socket.IO, Django Channels, FastAPI WebSockets и др.).
   - Фронтенд: Подключение через WebSocket API или библиотеку (Socket.IO-client).

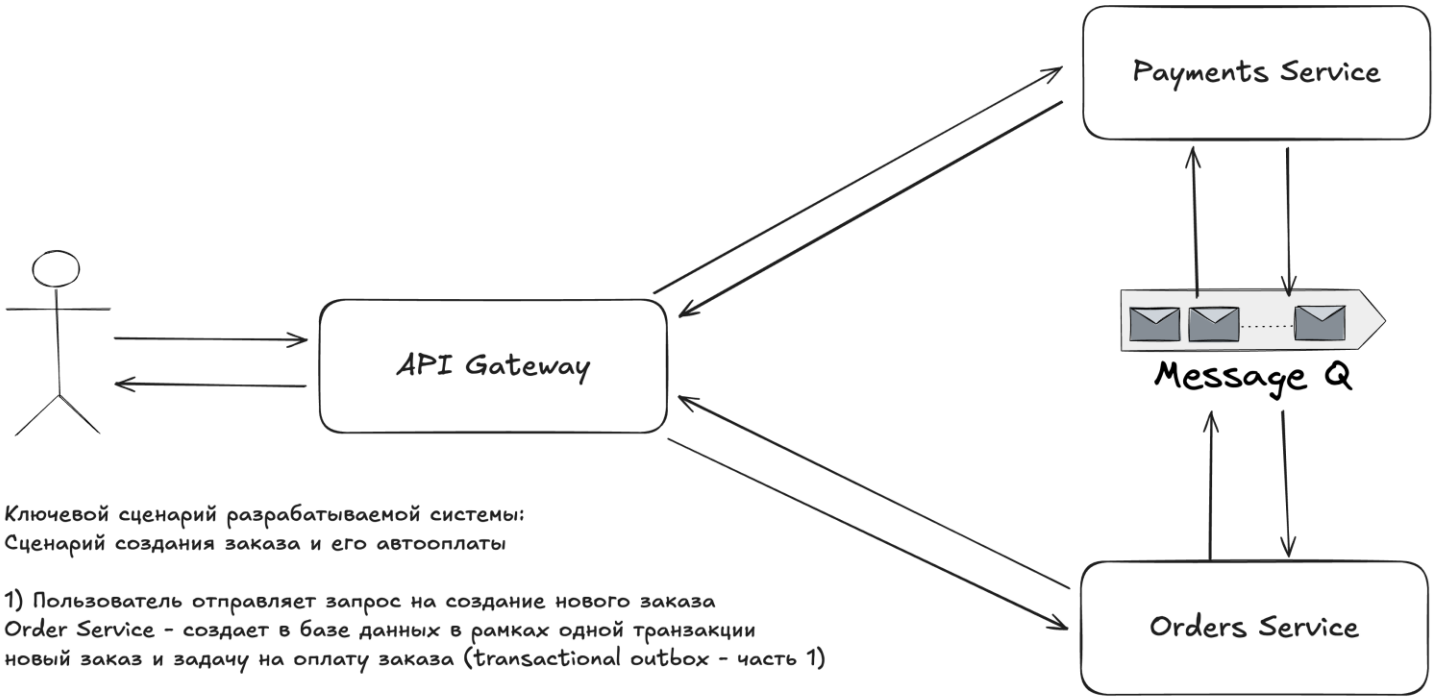
   Функционал:
   - Клиент подключается к WebSocket-серверу после создания заказа.
   - При изменении статуса заказа (например, "В обработке" → "Готов к выдаче") сервер отправляет уведомление.
   - На фронтенде отображается всплывающее push-уведомление (например, через toastify, notyf или браузерные Notification API).

   Два балла по данному пункту можно получить, если сделать еще несколько инстансов бэкенда с правильной доставкой push по WebSocket.

Штрафы

– 1 балл за каждый день просрочки дедлайна.

Пользовательские сценарии



Тут можно
1) создать счет по user_id
2) Пополнить этот счет на x денег
3) Посмотреть баланс счета

*Считаем что у одного пользователя всегда не более одного счета

Ключевой сценарий разрабатываемой системы:
Сценарий создания заказа и его автооплаты

1) Пользователь отправляет запрос на создание нового заказа
Order Service - создает в базе данных в рамках одной транзакции
новый заказ и задачу на оплату заказа (transactional outbox - часть 1)

2) Order Service асинхронно вычитывает задачу из базы данных
и отправляет ее в очередь (transactional outbox - часть 2)

3) Payments Service вычитывает задачу на оплату заказа и
сохраняет ее в БД (transactional inbox - часть 1)

4) Payments Service выполняет задачу (transactional inbox - часть 2)
и записывает в свою БД задачу на отправку события в очередь (transactional outbox - часть 1)
Если нет счета на такого пользователя - fail событие
Если у пользователя не хватает денег - fail событие
Если удачно списалось - success событие

5) Payments Service асинхронно вычитывает задачу на отправку события
о статусе оплаты из базы данных и отправляет ее в очередь (transactional outbox - часть 2)

5) Order Service ожидает событие об успешности/неуспешности оплаты
из очереди и обновляет статус соответствующего ей заказа
(можно не использовать паттер transactional inbox, потому что изменения идемпотентные)

Тут можно
1) создать заказ
2) посмотреть список заказов
3) посмотреть статус заказа

Заказ это:
Order - (id, user_id, amount, description, status)
status = {
  NEW - сразу после создания,
  FINISHED - если оплата прошла успешно,
  CANCELLED - если оплата не удалась
}