

КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Контрольная работа № 3 Асинхронное межсервисное взаимодействие.

Дедлайн сдачи: 15 июня, 23:59

Формат сдачи КР определяется семинаристом. Задание выполняется на языке программирования являющимся основным для семинарских занятий (иной язык только по согласованию с семинаристом)

Предисловие

Шопоголикам нужна ваша помощь!

С наступлением тёплых дней и приближением летнего сезона многие люди начинают планировать отдых, отпуск и покупки, связанные с предстоящими поездками. В условиях роста онлайн-шопинга, особенно в преддверии сезонных распродаж, нагрузка на интернет-магазины значительно возрастает. Это ставит перед разработчиками и архитекторами задачу обеспечить бесперебойную работу критически важных компонентов системы — в первую очередь, процесса создания заказа и его оплаты.

Вся надежда на вас! Ваш код — оружие против жабы.



Техническое задание

Вы являетесь частью команды разработки интернет-магазина, который ожидает значительный рост трафика в преддверии летних распродаж. Ваша задача — разработать два ключевых микросервиса, отвечающих за работу с платежами (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 у пользователя требуется обеспечить семантику *at most once* (реализация семантики *exactly once* будет дополнительным плюсом).

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.

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

1. Реализация основных требований к функциональности – **2 балла**
2. Архитектурное проектирование – **4 баллов**
 - a. Четкое разделение на сервисы (Order Service, Payments Service).
 - b. Логичное использование очередей сообщений.
 - c. Применение паттернов:
 - Transactional Outbox в Order Service.
 - Transactional Inbox и Outbox в Payments Service.
 - Обеспечение семантики exactly once при списании денег у пользователя
3. Реализация коллекции Postman / Swagger, которая должна демонстрировать функциональность реализованных микросервисов, охватывая все API – **0,5 балл**
4. Покрыто тестами более 65% кода – **0,5 балла**
5. Корректность Dockerfile и docker-compose.yml – **1 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 на 9 и 10 баллов (должны быть выполнены все предыдущие критерии)

6. Реализация фронтенд-части приложения – **2 балл**
 - a. Фронтенд должен быть реализован как отдельный сервис, взаимодействующий с бэкенд-микросервисами через REST API или GraphQL.
 - b. Фронтенд может быть реализован в виде десктопного, мобильного или веб-приложения.
 - c. Допускается использование любых современных фреймворков.
 - d. Фронтенд должен быть упакован в Docker-контейнер и запускаться через Docker Compose вместе с остальными сервисами.
 - e. Фронтенд не является основной частью работы, можно ограничиться минимально-достаточным интерфейсом в виде веб-приложения для получения 1 балла, но можно написать десктопный и/или мобильный клиентов с архитектурой (например WPF приложение с применением MVVM).
7. Реализация реального отслеживания состояния заказа через WebSocket + уведомления (push) – **2 балл**

Технологии:

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

Функционал:

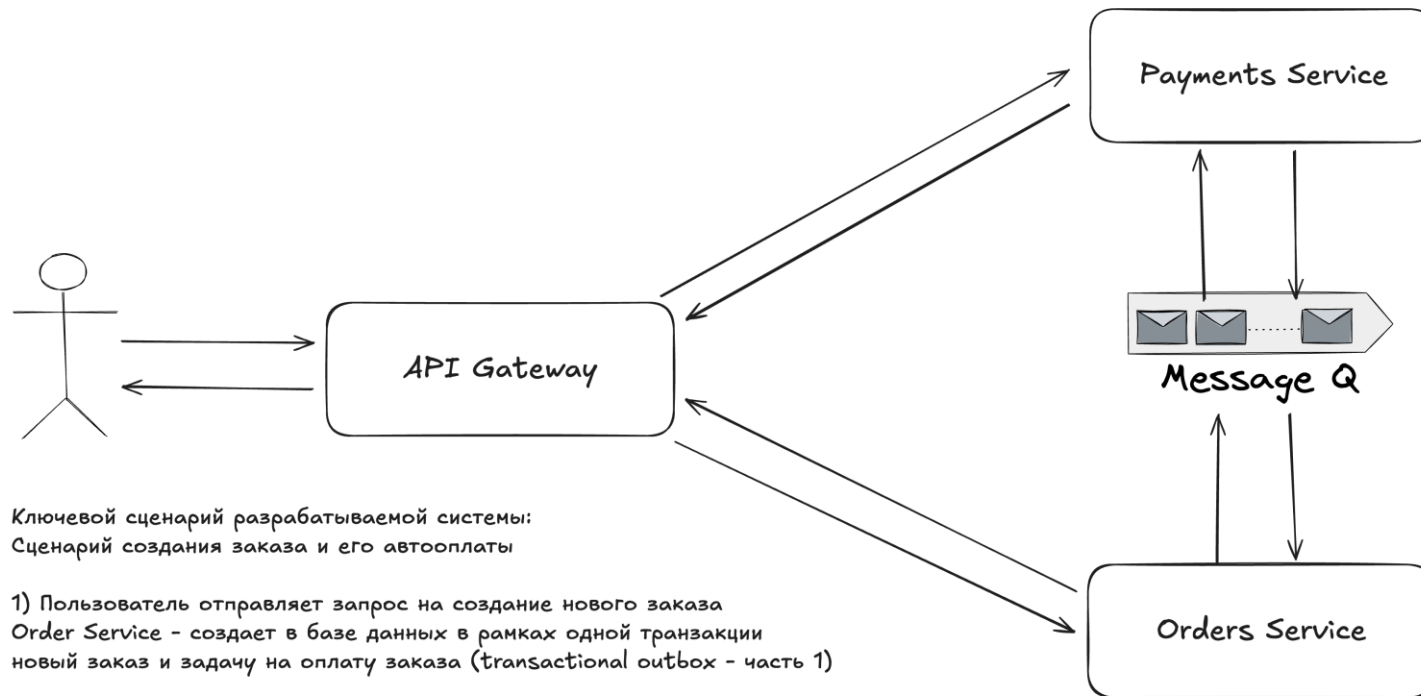
- Клиент подключается к WebSocket-серверу после создания заказа.
- При изменении статуса заказа (например, "В обработке" → "Готов к выдаче") сервер отправляет уведомление.
- На фронтенде отображается всплывающее push-уведомление (например, через toastify, notify или браузерные Notification API).

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

Штрафы

1. до – 2 баллов за наличие ошибки во время выполнения кода;
2. – 1 балл за каждый день просрочки дедлайна.

Пользовательские сценарии



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 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

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
}
```