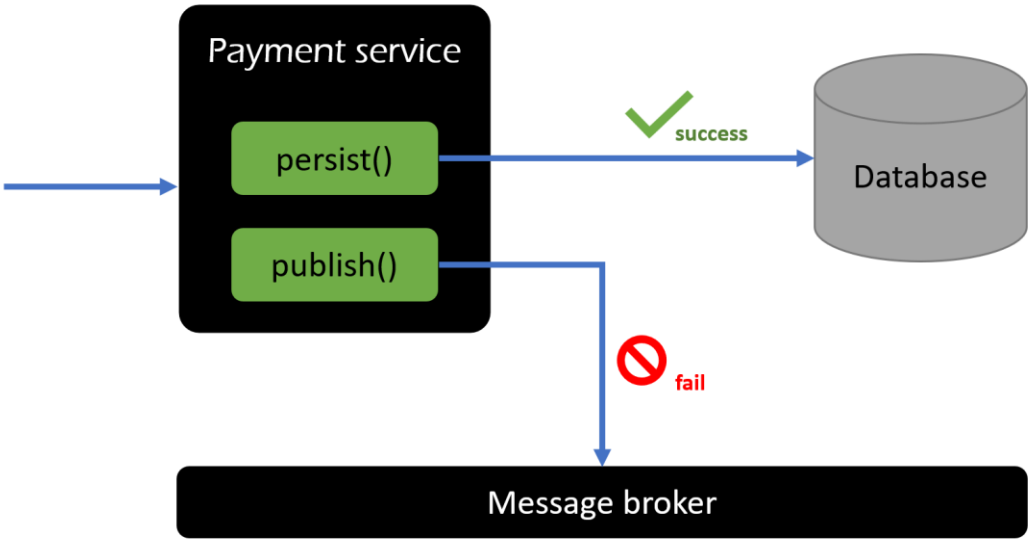
Trong trường hợp sau khi thực hiện xong business logic và persist data xuống database thành công nhưng publish message fail thì cần xử lý thế nào (application crash hoặc lost connection)?



1. Transactional outbox pattern?

Mình sẽ lấy ví dụ bài trước để những bạn chưa đọc cũng có thể hiểu được. Tuy nhiên mình khuyến khích nếu những ai chưa đọc hoặc chưa hiểu về SAGA pattern thì nên đọc trước khi tiếp tục nhé.

Thảo hiện tại là Solution Architect của team IT thuộc chuỗi cửa hàng Pizza và những con bug không thể fix. Thời buổi dịch bệnh khó khăn, chủ doanh nghiệp muốn Thảo thiết kế hệ thống bán hàng online để tăng doanh số. Vì đã có thâm niên chục năm vén váy.. à nhầm.. vén tay áo.. nên Thảo rất nhanh apply ngay Microservices Architecture với 4 service chính và sử dụng SAGA pattern để handle distributed transaction:

- Order service.
- Payment service.
- Restaurant service
- Delivery service.

Sau khi Order service tạo order thành công, publish event ORDER_CREATED và Payment service consume event này để xử lý tiếp. Payment service thực hiện business logic, commit transaction xuống database và thực hiện publish event ORDER_PAID. Tuy nhiên đúng lúc publish event thì...

toạch. Application crash, và thế là mất toi message, order tắc ở đấy và khách hàng chờ dài cổ vẫn chưa thấy pizza đâu.

1.1 Problem

Trước khi đi tìm giải pháp thì cần hiểu chính xác vấn đề cần giải quyết là gì:

- Mất connection đến Message broker dẫn đến việc không publish được event.
- Có thể mất message nếu application crash/restart.
- Cần đảm bảo tính atomic và consistent với 2 method persist() và publish(). Hiểu nôm na là nếu persist() thành công thì việc publish cũng phải thành công (thành công được hiểu là event được publish đến Message broker). Nếu không publish được, hay nói cách khác là lost event thì persist() cần được rollback.

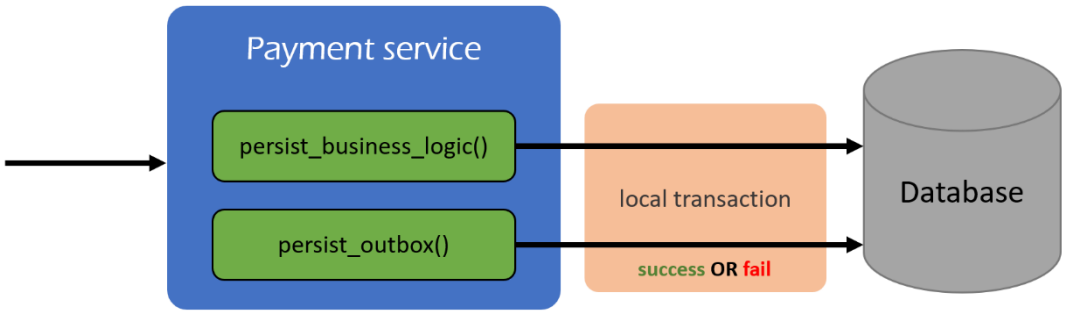
1.2 Solution

- Nếu mất connection và không publish được event thì một cách đơn giản có thể nghĩ đến là retry. Những thứ cần quan tâm là retry trong bao lâu, bao nhiêu lần?
- Cần store message ở đâu để đảm bảo nếu application crash/restart thì vẫn có message để retry: file, database, distributed storage?
- Nếu để đảm bảo vừa atomic mà vừa consistent thì chỉ có nhồi vào chung transaction thôi. Có nghĩa là message/event cần được lưu và database và xử lý chung một transaction với business logic?

Từ những idea trên, liệu bạn đã mường tượng ra tổng thể solution cần thực hiện là như thế nào chưa? Đi từng bước một nhé.

1.3 Create new table in database

Bước đầu tiên, tạo một table mới đặt tên là **outbox_table**. Khi xử lý business logic, bên cạnh việc update các table liên quan, ta insert thêm một record vào table **outbox_table**, đương nhiên record này chứa những thông tin cần thiết để publish event (order_id, state) thậm chí có thể lưu luôn event message.



Như vậy vấn đề về **atomic** và **consistent** đã được giải quyết một cách triệt để và đơn giản bằng cách sử dụng local transaction.

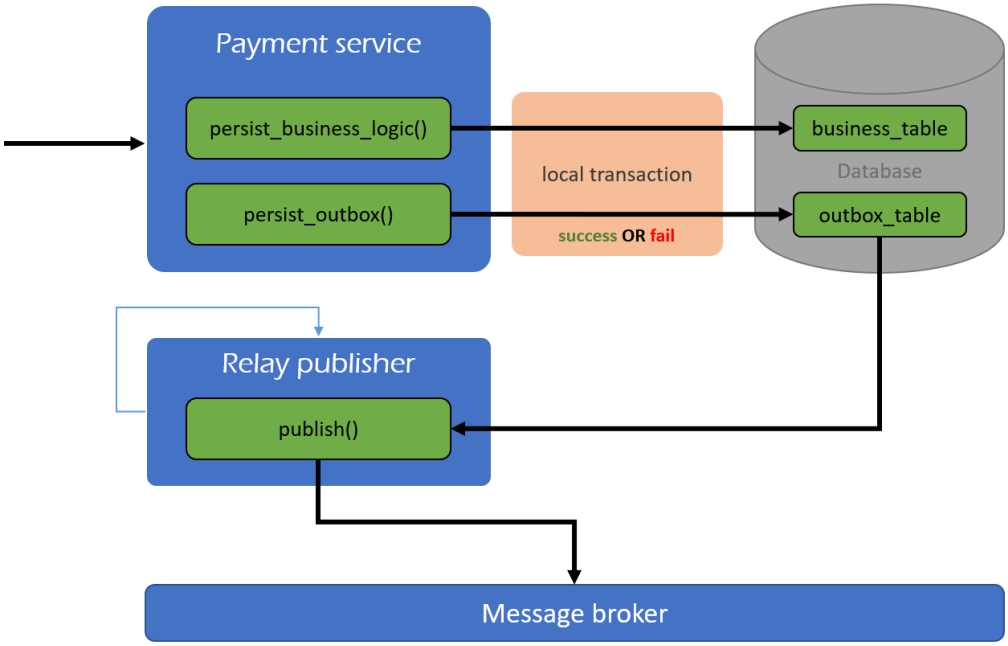
1.4 Create relay publisher

Như vậy các event được lưu trữ tại database, đảm bảo đầy đủ các tính chất quan trọng:

- **Consistency**: nếu store business data thành công thì mới có event, và ngược lại.
- **Durability**: một khi transaction commit thành công thì không thể lost message.
- **Message ordering**: message lưu trữ tại database theo thứ tự rõ ràng để đảm bảo khi publish message không có chuyện message đến sau lại publish trước.

Bây giờ đã một đống message đang chờ để publish, thì tất nhiên phải có publisher làm nhiệm vụ check xem đang có message nào không, nếu có thì publish. Tất nhiên vẫn không thể tránh trường hợp lost connection đến Message broker, vậy nên việc publish cần có cơ chế retry, và update message state sau khi publish thành công để tránh publish nhiều lần.

Pattern này là polling publisher. Đại khái publisher sẽ query liên tục (theo chu kì) đến outbox_table để tìm event và publish.



Một câu hỏi được đặt ra, vậy publisher này nên là một service độc lập hay là một service (class) nằm trong payment-service? Vì sẽ có tình huống payment-service crash còn nhiều event trong outbox_table đang chờ được publish. Nếu relay-publisher thuộc payment-service thì lúc này message không được publish. Nhưng nếu relay-publisher là service độc lập thì nó cần access vào outbox_table của payment-service, có vẻ không hợp lý?

Và đương nhiên, solution này là chính là **transactional outbox pattern**.

1.5 Alternative solution

Về cơ bản solution trên đã giải quyết được problem đưa ra ở đầu bài nếu sử dụng SQL (RDBMS). Tất nhiên, nó cũng có những nhược điểm cần chú ý:

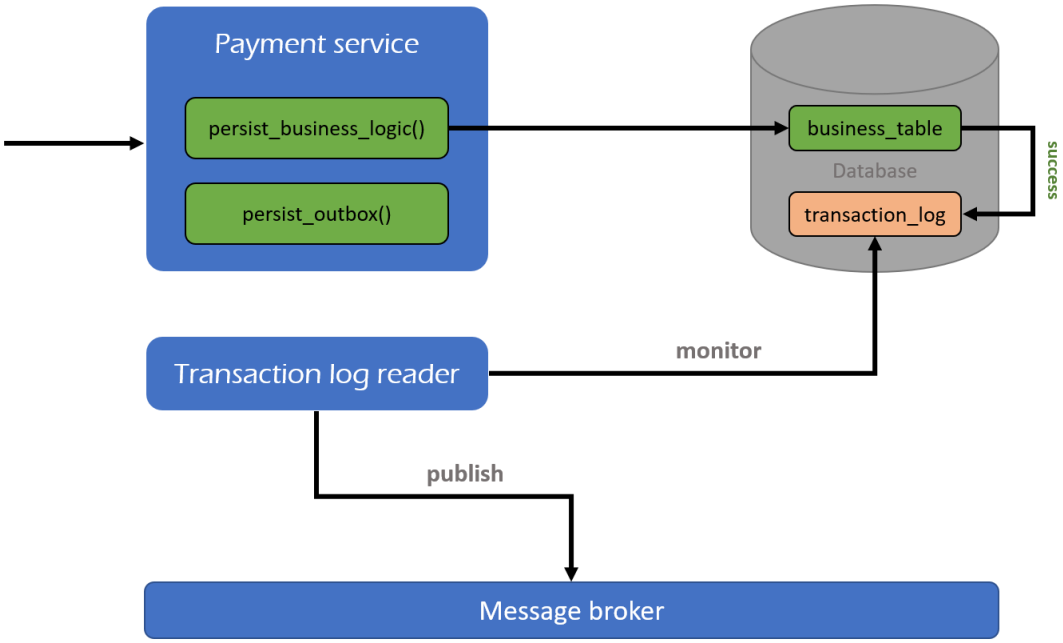
- Nếu application sử dụng NoSQL thì cần cẩn thận vì không phải NoSQL nào cũng có thể support pattern này (do không đảm bảo tính chất quan trọng của transaction).
- Extra call đến database để check có event nào cần publish không.

Quay lại vấn đề ban đầu, mấu chốt quan trọng nhất để giải quyết bài toán ở chỗ cần biết chính xác transaction cho business logic đã được commit thành công chưa để thực hiện việc publish event. Việc build event message hoàn toàn có thể thực hiện dựa trên business data... nhưng tất nhiên chẳng ai làm thế cả .

Vì vậy có một biến thể khác để implement relay publisher, và cũng để giải quyết 2 vấn đề trên là apply transaction log tailing pattern.

Nếu bạn đã làm việc MySQL thì chắc hẳn đã nghe đến binlog, hoặc nếu quen thuộc với Postgres là WAL. Có thể hiểu đơn giản rằng transaction log giống

như hộp đen của máy bay, lưu trữ tất cả lịch sử thay đổi dữ liệu của database. Khi dữ liệu bị thay đổi thì database cần lưu trữ các thay đổi đó vào log file. Và việc đọc log file này có thể giúp chúng ta biết transaction nào được commit, data nào được thay đổi. Từ có có thể build event để publish đến Message broker.



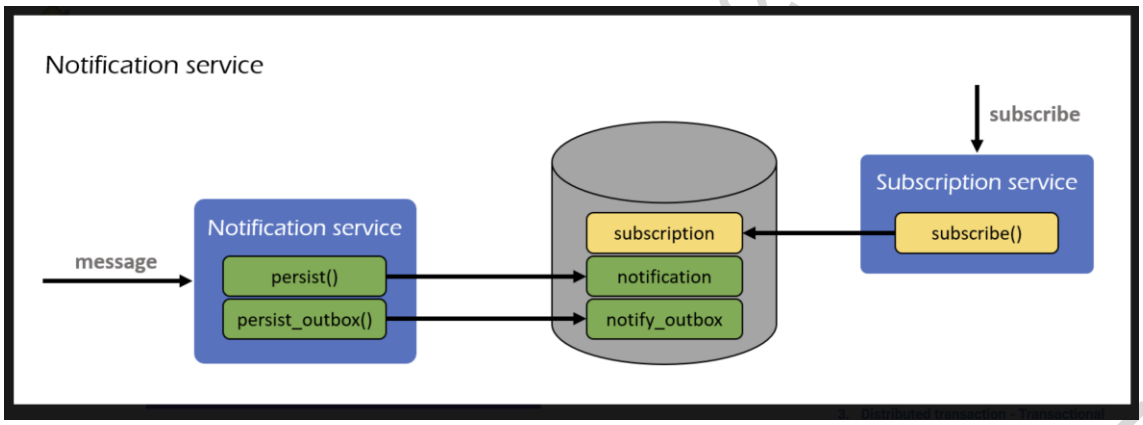
2. Case study: Notification service

2.1 Design & Flow

Việc cần làm là tạo ra Notification platform với mục đích chuyên để gửi thông báo đến người dùng thông qua các kênh khác nhau. Và một điều quan trọng là chỉ gửi đến những người đăng kí nhận thông báo.

Như vậy có thể tạm hình dung ra 3 components chính trong bài toán này:

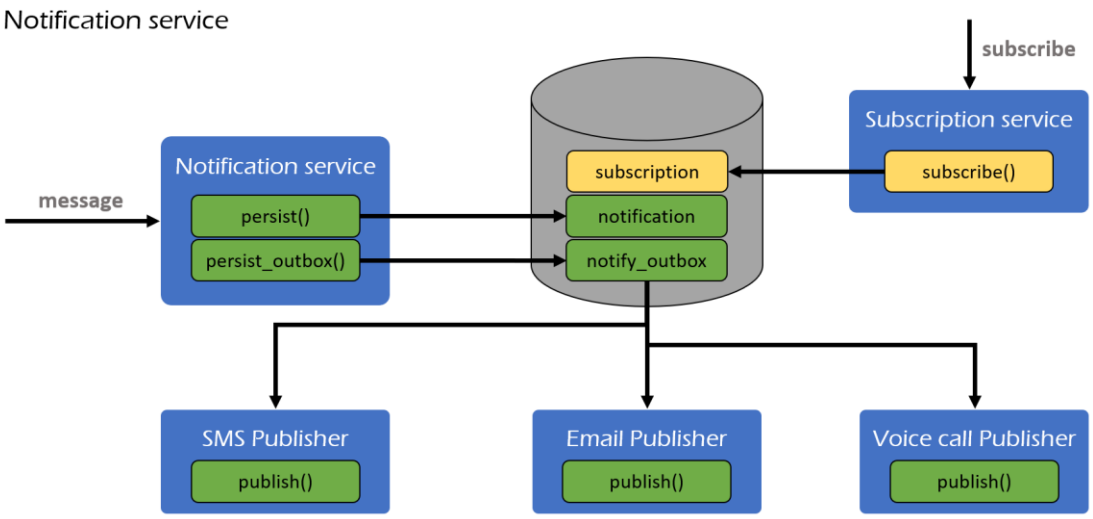
- Application: tất nhiên là notification-serivce rồi.
- Database: MySQL, Postgres... để thực hiện được transactional outbox pattern.
- Subscriber: khách hàng muốn nhận thông báo.



Đọc tiếp flow bên dưới kết hợp với hình bên trên để hiểu hơn flow nhé:

- Đầu tiên, khi client order sẽ có checkbox để lựa chọn việc có nhận thông báo hay không, nhận qua hình thức nào. Nếu có thì sau đó order-service sẽ gửi request tới notification-service để đăng kí nhận thông báo. Ví dụ thông qua HTTP POST /subscribe.
- Sau đó điều hướng đến SubscriptionService (class) để thực hiện business logic. Store thông tin vào subscription_table, có thể là một hoặc nhiều table khác, mình chỉ vẽ đại diện một table.
- Sau khi nhà bếp nhận thực đơn, hệ thống muốn gửi thông báo trạng thái order đến người dùng. Lúc này order-service hoặc restaurant-service gửi message đến notification-service. Notification-service apply transactional outbox pattern như hình trên, store business data vào notification table và outbox message vào notify_outbox table.

Tiếp theo và việc publish notification đến người dùng, hiện thời có 3 channel là sms, email, voice call tương ứng với 3 relay publisher. Mỗi publisher sẽ chủ động monitor message của riêng mình để publish đến địa chỉ đích.



Với design này có thể dễ dàng thêm các publisher một cách độc lập, dễ dàng scale. Client cũng dễ dàng trong việc lựa chọn việc nhận thông báo, và nhận qua hình thức nào.

3. Limitation

Transactional outbox pattern cũng bá đạo thật đấy nhưng vẫn có nhược điểm nhất định mà ta cần nắm rõ để xử lý bài toán cho tốt, cho triệt để.

Duplicate event: rất khó để đảm bảo việc message delivery là exactly once. Vấn đề publish message thành công và chưa kịp update lại vào database (application crash) là chuyện hết sức bình thường. Do vậy đầu consume cần đảm bảo được việc có thể xử lý duplicate message. Hay nói cách khác là cần implement idempotent consumer.

Near real-time: chắc chắn là rất khó để đạt đến trạng thái real-time application. Vấn đề là ta có chấp nhận có độ trễ không và độ trễ là bao nhiêu thì chấp nhận được.

4. Cuối cùng

Quay lại câu hỏi ở phần đầu publisher nên là một service độc lập hay là một inner-service?

Theo quan điểm cá nhân, nó sẽ phụ thuộc vào bài toán cần giải quyết là gì, vấn đề có phức tạp hay không, yêu cầu letancy thế nào, có cần mở rộng trong tương lai không?

Ví dụ về **notification-service** phía trên, chắc chắn là việc chia thành các service độc lập là hiệu quả hơn. Trong trường hợp thêm một channel mới ta chỉ

việc implement service mới mà không cần sửa code cũ. Nó giúp việc scale dễ dàng và bớt tốn kém. Chỉ có thêm vấn đề nho nhỏ là cần monitor thêm chính service đó. Và như mình nói, vấn đề nho nhỏ nên có thể coi là không thành vấn đề.