Pattern: Messaging

Context

You have applied the Microservice architecture pattern (../microservices.html). Services must handle requests from the application's clients. Furthermore, services often collaborate to handle those requests. Consequently, they must use an inter-process communication protocol.

Forces

- Services often need to collaborate
- Synchronous communicate results in tight runtime coupling, both the client and service must be available for the duration of the request

Problem

How do services in a microservice architecture communicate?

Solution

Use asynchronous messaging for inter-service communication. Services communicating by exchanging messages over messaging channels.

There are several different styles of asynchronous communication:

- Request/response - a service sends a request message to a recipient and expects to receive a reply message promptly
- Notifications - a sender sends a message a recipient but does not expect a reply. Nor is one sent.
- Request/asynchronous response - a service sends a request message to a recipient and expects to receive a reply message eventually
- Publish/subscribe - a service publishes a message to zero or more recipients
- Publish/asynchronous response - a service publishes a request to one or recipients, some of whom send back a reply

Examples

There are numerous examples of asynchronous messaging technologies

- Apache Kafka (http://kafka.apache.org)
- RabbitMQ (https://www.rabbitmq.com/)

`OrderService` from the FTGO Example application (https://github.com/microservices-patterns/ftgo-application) publishes an `Order Created` event when it creates an `Order` .

```
public class OrderService {

  ...

  public Order createOrder(long consumerId, long restaurantId,
                           List<MenuItemIdAndQuantity> lineItems) {
    Restaurant restaurant = restaurantRepository.findById(restaurantId)
            .orElseThrow(() -> new RestaurantNotFoundException(restaurantId));

    List<OrderLineItem> orderLineItems = makeOrderLineItems(lineItems, restaurant);

    ResultWithDomainEvents<Order, OrderDomainEvent> orderAndEvents =
            Order.createOrder(consumerId, restaurant, orderLineItems);

    Order order = orderAndEvents.result;
    orderRepository.save(order);

    orderAggregateEventPublisher.publish(order, orderAndEvents.events);

    OrderDetails orderDetails = new OrderDetails(consumerId, restaurantId, orderLineItems, order.getOrderTota
l());

    CreateOrderSagaState data = new CreateOrderSagaState(order.getId(), orderDetails);
    createOrderSagaManager.create(data, Order.class, order.getId());

    meterRegistry.ifPresent(mr -> mr.counter("placed_orders").increment());

    return order;
  }
```

Resulting context

This pattern has the following benefits:

- Loose runtime coupling since it decouples the message sender from the consumer
- Improved availability since the message broker buffers messages until the consumer is able to process them
- Supports a variety of communication patterns including request/reply, notifications, request/async response, publish/subscribe, publish/async response etc

This pattern has the following drawbacks:

- Additional complexity of message broker, which must be highly available

This pattern has the following issues:

- Request/reply-style communication is more complex

Related patterns

- The Saga pattern (/patterns/data/saga.html) and CQRS pattern (/patterns/data/cqrs.html) use messaging
- The Transactional Outbox pattern (/patterns/data/transactional-outbox.html) enables messages to be sent as part of a database transaction
- The Externalized configuration pattern (../externalized-configuration.html) supplies the (logical) message channel names and the location of the message broker
- The Domain-specific protocol (domain-specific.html) pattern is an alternative pattern
- The RPI (rpi.html) pattern is an alternative pattern