Scalable Microservices through Messaging

By Bilgin Ibryam
May 26, 2016

(No Ratings Yet)

Microservices are everywhere nowadays, and so is the idea of using service choreography (instead of service orchestration) for microservices interactions. In this article I describe how to set up service choreography using ActiveMQ virtual topics, which also enables scalable event based service interactions.

Service Interaction Styles

There are two main types of service interaction: **synchronous** and **asynchronous**.

With *synchronous* interactions, the service consumer makes a request and blocks until the operation completes and a response is received. The HTTP protocol is a great example for a synchronous interaction. This type of interaction is usually associated with request/response interaction style and the HTTP protocol. (Of course, it also possible to do request/response with asynchronous requests or event messaging, via registering a callback for the result, but that is a less common use case).

With an *asynchronous* interaction style, the service consumer makes a request, but doesn't wait for the operation to complete. As soon as the request is acknowledged as received, the service consumer moves on. This type of interaction allows publish/subscribe style of service communication – e.g. instead of a service consumer invoking an operation from another service, a producer raises an event and expects interested consumers to react.

Apart from these technical considerations, there is also another aspect to consider with service interactions: coupling and responsibility.

If service A has to interact with service B, is it the responsibility of service A to invoke service B (***orchestration***) or is it the responsibility of service B to subscribe for the right events (***choreography***)?

With service orchestration, there is a central entity (as the service A itself in our case), which has the knowledge of other services to be called. With the choreography approach, this responsibility is delegated to the individual services and they are responsible for subscribing for the "interesting" events.

To read more about this topic, checkout Chapter 4 from the excellent Building Microservices book . For the rest of this article, we will focus on doing service choreography using messaging.

Service Orchestration Through Messaging

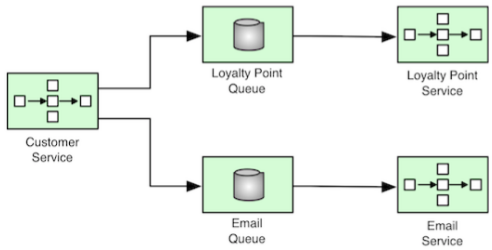
Service orchestration in messaging is achieved through queues. A queue implements load balancer semantics using the competing consumer pattern and makes sure that there is only one consumer of a message.

Let's say there is a "Customer Service" that has to interact with "Email Service".

The easiest way to implement this is to use a queue and let "Customer Service" send a message to "Email Queue". If the "Customer Service" has to interact with "Loyalty Point Service", again, "Customer Service" has to send another message — this time to "Loyalty Point Queue".

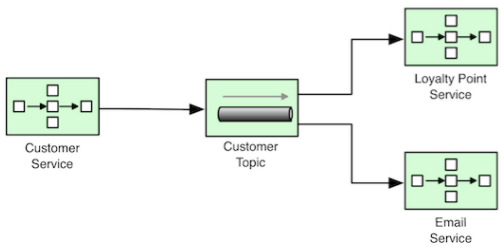
With this approach, it is the responsibility of "Customer Service" to know about "Loyalty Point Service" and "Email Service", and subsequently send the right messages to the corresponding queues. In short, the whole interaction is orchestrated by "Customer Service".

One advantage of using queues is that, it allows scaling of consumers easily. We could start multiple instances of "Loyalty Point Service" and "Email Service", and the queues will do the load balancing among the consumers.



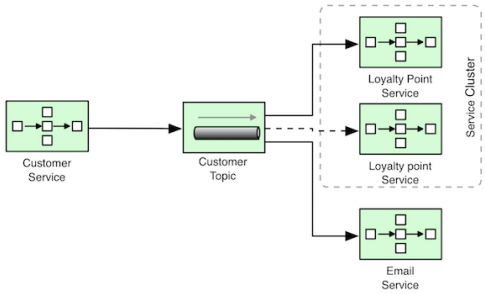
Service Choreography Through Messaging

With the service choreography approach, "Customer Service" doesn't have any knowledge of "Loyalty Point Service" or "Email Service". "Customer Service" simply emits an event to "Customer Topic", and it is the responsibility of "Loyalty Point Service" and "Email Service" to know about the Customer event contract and subscribe to the right topic — the publish/subscribe semantics of the topics will ensure that that every event is distribute to both subscribers.



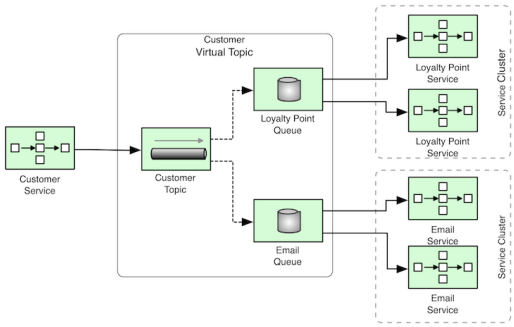
Scaling Service Choreography

Since topics implement publish/subscribe semantics rather than competing consumers, scaling the consumers becomes harder. If we (horizontally) scale "Loyalty Point Service" and start two instances, both instances of the service will receive the same event and there won't be any benefit in scaling (unless the services are idempotent).



ActiveMQ Virtual Topics to the Rescue

So what we need is some kind of mixture between topic and queue semantics. We want the "Customer Service" to publish an event using publish/subscribe semantics so that all services receive the event, but we also want competing consumers, so that individual service instances can load balance events and scale.



There are a number of ways we could achieve that with Camel and ActiveMQ:

- The very obvious one that comes to (my) mind is to have a simple Camel route that is consuming events from "Customer Topic" and sends them to both "Loyalty Point Queue" and "Email Queue". This is easy to implement, but every time there is a new service interested from the "Customer Service" events, we have to update the Camel routes. Also, if you run the Camel route on a separate process than the broker, there will be unnecessary networking overhead only to move messages from a topic to a set queues in the same message broker.
- An improvement to the above approach would be, to have Camel routes running in the ActiveMQ broker process using ActiveMQ Camel plugin. In that case, you still have to update the Camel route every time there is change to the subscribers, but the routing will happen in the broker process itself, so no networking overhead.
- And even a better solution would be to have the queues subscribed to the topic w/o any coding, but using a declarative approach using ActiveMQ virtual topics (hence the whole reason for writing this article).

Virtual topics are a declarative way of subscribing queues to a topic, by simply following a naming convention — all you have to do is define or use the default naming convention for your topic and queues.

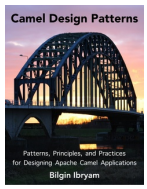
For example, if we create a topic with a name matching *VirtualTopic.>* expression, such as: *VirtualTopic.CustomerTopic*, then have the "Loyalty Point Service" consume from *Consumer.LoyaltyPoint.VirtualTopic.CustomerTopic* queue, the message broker will forward every event from *VirtualTopic.CustomerTopic* topic to *Consumer.LoyaltyPoint.VirtualTopic.CustomerTopic* queue.

Then we could scale Loyalty Point Service by starting multiple service instances, all of which consume from *Consumer.LoyaltyPoint.VirtualTopic.CustomerTopic* queue.

Similarly, later we can create a queue for the Email Service by following the same naming convention:

Consumer.Email.VirtualTopic.CustomerTopic. This feature allows us to simply name our topics and queues in a specific way, and have them subscribed without any coding.

Final thoughts



This is only one of the many patterns I have described in my recently published Camel Design Patterns book . Camel is quite often used with ActiveMQ, and as such, you can find also some ActiveMQ patterns in the book too.

Another way to scale microservices using choreography can be achieved through event sourcing. You can find a nice blog post describing it here .

Editor's note: Apache Camel is part of Red Hat JBoss Fuse , and is available for download as part of your no-cost Red Hat Developers subscription .

Posted in Java JBoss A-MQ JBoss Fuse Uncategorized EIP messaging microservices

---

TOP POSTS

No-Cost RHEL Developer Subscription now available
Java inside docker: What you must know to not FAIL
How to find and fix memory leaks in your Java application
Fedora Media Writer - The fastest way to create Live-USB boot media
10 things to avoid in docker containers

RECENT POSTS

JBoss Data Virtualization on OpenShift: Integrating a Remote SQL Server Database
SCTP Stream Schedulers and User Message Interleaving
Enabling SAML-based SSO with Remote EJB through Picketlink
Develop and Deploy on OpenShift Online Starter using Red Hat JBoss Developer Studio
Enabling Byteman Script with Red Hat JBoss Fuse and AMQ