Microservice Messaging Research document

TwitterV2

Daniil Blagoev

Table of Contents

[Introduction 2](#_Toc169513355)

[Why does TwitterV2 need microservice messaging? 2](#_Toc169513356)

[Research questions and methods 2](#_Toc169513357)

[Answers 2](#_Toc169513358)

[What types of messaging-based communication between microservices are there, and which one would be the best fit for TwitterV2’s needs? 2](#_Toc169513359)

[How can I implement the messaging-based communication from the first sub-question into TwitterV2? 3](#_Toc169513360)

[Conclusion 10](#_Toc169513361)

[Sources 11](#_Toc169513362)

# Introduction

The purpose of this research document is to find out how messaging between microservices works, what types of messaging is there, and how to implement it into TwitterV2.

## Why does TwitterV2 need microservice messaging?

In the microservice architecture it is considered best to keep the different microservices as decoupled as possible. If a TwitterV2 user’s account is deleted or the username is updated, their tweets are to be deleted/updated as well. In order to do that and keep the services decoupled, messaging should be looked into.

# Research questions and methods

Main research question:

* What would the benefits of implementing messaging between TwitterV2’s tweet and user microservice be?

Sub-questions:

* What types of messaging-based communication between microservices are there, and which one would be the best fit for TwitterV2’s needs?
  + Literature study
    - Googling.
  + Problem analysis
    - Analyze TwitterV2’s problem and see which type of communication would solve it.
* How can I implement the messaging-based communication from the first sub-question into TwitterV2?
  + Available product analysis
    - Find out if there are any examples of messaging implementation on the internet and see how/if I can make them fit TwitterV2’s needs.
  + System test
    - Write and run tests which prove the newly implemented messaging works as expected.

# Answers

## What types of messaging-based communication between microservices are there, and which one would be the best fit for TwitterV2’s needs?

There are three types of messaging-based communication between microservices:

* **Single-receiver message-based communication** - Message-based asynchronous communication with a single receiver means there's point-to-point communication that delivers a message to exactly one of the consumers that's reading from the channel, and that the message is processed just once.
* **Multiple-receivers message-based communication** - Communication from the sender will be available to additional subscriber microservices or to external applications.
* **Asynchronous event-driven communication** - When using asynchronous event-driven communication, a microservice publishes an integration event when something happens within its domain and another microservice needs to be aware of it, like a price change in a product catalog microservice. Additional microservices subscribe to the events so they can receive them asynchronously. When that happens, the receivers might update their own domain entities, which can cause more integration events to be published.[[1]](#footnote-1)

A diagram of a event

Description automatically generated

Figure 1 A diagram of asynchronous event-driven communication

The problem of TwitterV2 is the following: when a user is deleted or updates their username, the tweet and (eventually) comment microservices have no way of “knowing” that, so that they can delete/update the occurrences of the username in their own databases. This would cause many cases of data mismatch down the road.

Based on this problem description, Asynchronous event-driven communication is the way to go. Why? Because TwitterV2’s microservices need to communicate with each other only when user deletion and updates happen, and they are both events.

## How can I implement the messaging-based communication from the first sub-question into TwitterV2?

It is important to note that I will not be looking for cloud-based solutions due to the fact that I already have a few different cloud services implemented and I need my credits to last until the end of the semester.

#### RabbitMQ vs Kafka

I found multiple different ways to implement Asynchronous event-driven communication into TwitterV2. The differences come from using either Kafka or RabbitMQ, which means I need to decide which one to use. Since both of them can sufficiently solve TwitterV2’s problem, I will base the choice on performance. According to this article [Benchmarking RabbitMQ vs Kafka vs Pulsar Performance (confluent.io)](https://www.confluent.io/blog/kafka-fastest-messaging-system/), RabbitMQ outperforms Kafka in terms of latency “*only* at significantly lower throughputs”. Additionally, Twitter itself uses Kafka, so I will use it too.

#### Implementing Kafka

To implement Kafka, I mostly followed this article [Creating Microservices with .NET Core and Kafka: A Step-by-Step Approach | by Manan Patadiya | Simform Engineering | Medium](https://medium.com/simform-engineering/creating-microservices-with-net-core-and-kafka-a-step-by-step-approach-1737410ba76a). I started off by adding Kafka and Zookeeper to TwitterV2’s docker compose file:

A screenshot of a computer

Description automatically generated

Figure 2 TwitterV2's docker-compose file where I added Kafka and zookeeper

Then I reworked the user service to be a producer. This code sends a message to the user\_deletion topic once a user is deleted. The message contains the Username of the deleted user:

A screen shot of a computer program

Description automatically generated

Figure 3 The producer service

Then I reworked the tweet service to be a consumer. This code is continuously “consuming” any messages with the user\_deletion topic. Once it consumes a message, it uses the Username in the message to delete all tweets associated with the Username:

A screen shot of a computer program

Description automatically generated

Figure 4 The consumer service

In order to prove that my messaging implementation works, I used JMeter in my netlab environment where I deploy TwitterV2 on Kubernetes. This is what the flow of the test looks like:

A screenshot of a computer

Description automatically generated

Figure 5 The steps of the integration test for messaging

First a user and a tweet are created. They are “linked” through username. Then, it is checked that the new tweet exists. Afterwards, the user is deleted. This makes use of the newly implemented messaging, and it results in the tweet created by the user to be deleted as well. Finally, it is checked that the deletion of tweets actually went through.

Afterwards, I added the test to my GitHub Actions as well:

A screenshot of a computer program

Description automatically generated

Figure 6 The GitHub workflow file for the messaging test

And here it can be seen that the test passes:

A screenshot of a computer

Description automatically generated

Figure 7 The newly added messaging test passes

# Conclusion

The benefits of implementing messaging between the user and tweet microservice are:

* Performance – the implementation of messaging allows asynchronous communication between the 2 services, which allows them to continue working without having to wait for an answer from one another.
* Decoupling & Uptime – the implementation of messaging allows the 2 services to remain as decoupled as possible. If one of them is down, the other one can keep working.

# Sources

J. (2022, October 29). *Asynchronous message-based communication - .NET*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication

*Benchmarking RabbitMQ vs Kafka vs Pulsar Performance*. (n.d.). Confluent. https://www.confluent.io/blog/kafka-fastest-messaging-system/

Patadiya, M. (2023, October 23). *Creating Microservices with .NET Core and Kafka: A Step-by-Step Approach*. Medium. https://medium.com/simform-engineering/creating-microservices-with-net-core-and-kafka-a-step-by-step-approach-1737410ba76a

1. J. (2022, October 29). *Asynchronous message-based communication - .NET*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication [↑](#footnote-ref-1)