# Chapter 9: Messaging with Kafka

Please send feedback to: gunnar@sireus.se

In this chapter, the **RabbitMqAPI** project from the previous chapter has been migrated to use **Kafka** instead of **RabbitMQ**. The Kafka messaging system runs inside a Docker container, so to run it locally, you'll need **Docker Desktop** installed on your PC.

The migration was accomplished by updating the implementation of the following interfaces:

A screenshot of a computer screen

AI-generated content may be incorrect.IClientQueueRepository

A screenshot of a computer screen

AI-generated content may be incorrect.

IServerQueueRepository

## Converting the RabbitMqAPI to KafkaAPI

It took me some time to get it working—lots of reading, thinking, and trial and error. There are 100 properties in ProducerConfig 106 properties in ConsumerConfig to configure, even though most of them have default values.

While troubleshooting, I ran into this exception that had to be addressed:

*Confluent.Kafka.KafkaException: OffsetCommit failed: Broker: Static consumer fenced by other consumer with same group.instance.id*

I spent many hours trying to solve the issue, and it turned out the fix was to run the Kafka consumer as a singleton.

I also encountered rebalancing issues that severely impacted performance. To avoid this, I had to ensure the GroupInstanceId remained constant throughout execution—which is achieved by running the consumer continuously in a single instance.

Since Kafka is designed for high throughput rather than point-to-point (P2P) messaging, it batches messages by default, even when sending just one. If you need low-latency, reliable P2P messaging, it makes sense to disable batching

The app is running pretty smoothly now, and I’ve done some tuning of both **ProducerConfig** and **ConsumerConfig**. You can check out the settings in the code—look in the **ClientQueueRepository** and **ServerQueueRepository** files.

All in all, using Kafka for a simple P2P setup feels like overkill—more complicated than it needs to be. But it works.

## What is Kafka

In today’s distributed systems, efficient communication between services is key. Apache Kafka, developed in Java and primarily running on Linux, is a high-performance, fault-tolerant, and scalable messaging platform for real-time data. Kafka handles massive volumes of event-driven data in microservices, IoT, and analytics platforms.

What sets Kafka apart is its log-based architecture. Unlike traditional brokers, Kafka stores messages on disk, allowing multiple consumers to access them repeatedly. This makes it ideal for streaming, event sourcing, and asynchronous communication.

Apache Kafka is primarily designed for publish/subscribe (pub/sub) messaging and high-throughput event streaming. It’s not inherently optimized for point-to-point (P2P) communication, but with careful configuration—like using unique topics and static group.instance.ids—it’s definitely possible to implement a reliable P2P pattern in .NET.

While there are no out-of-the-box .NET libraries that treat Kafka as a traditional P2P queue, you can make it work by managing consumers and topics explicitly, as demonstrated in this project.

If you're looking for P2P communication in .NET environments, you might want to consider other libraries specifically built for that purpose.

Alternative P2P libraries for .NET include:

* MonoTorrent: A BitTorrent library for .NET that enables P2P file sharing.
* ZeroMQ: A high-performance asynchronous messaging library that can be used for P2P communication. It supports various messaging patterns, including P2P.
* Or why not use ActiveMQ Artemis or RabbitMQ, as described in the previous chapters?

These alternatives may be better suited than Kafka for P2P communication in .NET applications.

## P2P as a Subset of Pub/Sub

In Kafka (and other pub/sub systems), you can model point-to-point (P2P) messaging as a special case, or subset of Pub/Sub, where:

A producer sends messages to a topic.

Only one consumer reads each message — in other words, a single-consumer model.

This is achieved in Kafka by:

Having all consumers that are supposed to act as “recipients” in a P2P scenario belong to the same consumer group.

Kafka then guarantees that each message in a partitioned topic is delivered to exactly one consumer within the group.

## Run Kafka in a Docker Container

I started by trying to install and run Kafka as a Windows service, just like ActiveMQ and RabbitMQ in previous chapters. After two days, I gave up. It seems that this particular installation of Kafka 4.0, without ZooKeeper, on Windows is poorly tested. It's probably possible to install Kafka 4.0 as a Windows service, maybe with NSSM, which I got a tip about, but I haven't tried it. NSSM, or Non-Sucking Service Manager, is a tool that lets you run programs as Windows services. It's commonly used when an application can't natively run as a service on Windows, which is the case for many programs that aren't designed for it.

Instead, I switched things up and decided to run Kafka in a Docker container. Figuring out which environment variables to set in the docker-compose.yml file took a few hours—lots of reading and trial and error. Here’s the result:

A screen shot of a computer program

AI-generated content may be incorrect.

Docker-compose.yml

In the Visaul Studio project **KafkaAPI** there are two .BAT files, **KafkaStart** and **KafkaStop**, for starting and stopping the Kafka services. Make sure Docker Desktop is started before running these files.

A computer screen shot of a program

AI-generated content may be incorrect.

Start containers with KafkaStart

A screen shot of a computer

AI-generated content may be incorrect.

Stop containers with KafkaStop

**Verify Kafka by Confluent Control Center**

If you install Kafka using the Confluent Platform, a web UI is included: <http://localhost:9021>

A screenshot of a computer

AI-generated content may be incorrect.

## QueueRepository

The code in this chapter, as well as in the other chapters on **QueueAPI**, **ActiveMQ**, and **RabbitMQ**, is largely the same. The only difference lies in how the **IClientQueueRepository** and **IServerQueueRepository** interfaces are implemented. In the previous chapters, both interfaces were implemented in a single class, **QueueRepository**. However, with Kafka, I had to split **QueueRepository** into two separate classes—one for each interface—and register each as a singleton.

A computer screen shot of a program code

AI-generated content may be incorrect.

After initiating the consumer in the constructor, **AddClientQueueItemAsync**(**QueueEntity** entity) is implemented

A screen shot of a computer program

AI-generated content may be incorrect.ProduceClientMessageAsync(QueueEntity entity) is called by AddClientQueueItemAsync(…)

A computer screen shot of a program

AI-generated content may be incorrect.GetMessageFromServerByCorrelationId(Guid correlationId) from interface IClientQueueRepository

A computer screen shot of a program

AI-generated content may be incorrect.

Continuously listening for messages, the consumer loop stores them in a list, allowing retrieval by correlation ID

## Running the KafkaAPI Solution

The complete solution is available for download at: <https://github.com/developersplaybook/KafkaAPI>

In Visual Studio, set **Client** and **Server** as **start projects**:

A screenshot of a computer

AI-generated content may be incorrect.

In SQL Server, the databases **ASPNETDb** and **CarDb** must exist.

**"CarDbConnection": "Server=localhost;Initial Catalog=CarDb;Integrated Security=True;Max Pool Size=100;Encrypt=false".**

If you prefer “Server=.” instead of “Server=localhost”, it will also work.

A screenshot of a computer program

AI-generated content may be incorrect.

Open the solution in Visual Studio

Tere are two .BAT files in the solution, **KafkaStart** and **KafkaStop**, for starting and stopping the Kafka services. Make sure Docker Desktop is started before running these files.

A computer screen shot of a program

AI-generated content may be incorrect.

Start containers with KafkaStart

To run the solution, set **Client** and **Server** as start projects and press F5.

A screenshot of a computer

AI-generated content may be incorrect.

The Client application opens at <http://localhost:63566/>

**Swagger**

A screenshot of a computer

Description automatically generated

You open Swagger at <http://localhost:63566/swagger/index.html>