



Iryna Tyshchenko

[Follow](#)Nov 21, 2022 · 11 min read · [Listen](#)

Save



Reliable message exchange using outbox pattern with Debezium

ABSTRACT

This article describes an example of building an architecture for reliable publication of messages between microservices based on the domain area of investment and finance. The solution interacts with various microservices using the transactional log approach for the reliable asynchronous exchange of data and messages.

KEYWORDS

Change data tracking, debezium, kafka, transaction log, outbox pattern.

I. INTRODUCTION

Many will agree that when working with multiple microservices, the most difficult part of them is the data: microservices do not exist in isolation, and very often they need to coordinate and disseminate data and data changes with each other. There are different approaches to data transfer. For example, one of the microservices may call some REST, GRPC, or other (synchronous) API provided by another microservice.

However, this approach has its drawbacks. First, the sending service must know which other services to call and where to find them. Second, the sender must be prepared that these services are temporarily unavailable. Third, one service cannot actually function without the other services it calls. In addition, there is often a lack

of replayability — the ability for new consumers to receive notifications after events have been sent.

These problems can be solved by using an asynchronous data exchange approach instead. For example, using Apache Kafka. By subscribing to these event streams, each service will be notified of changes in the data of other services. The service can respond to these events and, if necessary, create a local view, tailored to its own needs, in its own data warehouse. For example, such a view may be denormalized to effectively support certain access patterns, or it may contain only a subset of the source data that is relevant to the consuming service. Reliable logs also support replayability. This means that new consumers can be added as needed and they receive all messages [1].

II. OUTBOX PATTERN CONCEPT

An easy way to reliably publish messages using a relational database is to use the Outbox pattern. This pattern uses the database table as a temporary message queue. The service reliably publishes the message by inserting it into the outbox table as part of the transaction that updates the database. The outbox table acts as a temporary queue of messages. Atomicity is guaranteed because it is a local ACID transaction.

Message Relay is a component that reads an outbox table and publishes a message to the message broker. Internal table updates made by the database client are wrapped in an outbox table update transaction so that even in the case of any failures, data consistency between them is ensured.

Meanwhile, a single program thread or process is used to constantly poll the outbox table and create data in the appropriate event streams. Upon successful creation, the corresponding entries in the outbox table are deleted. In the event of any malfunction, whether it is the database, the consumer/producer or the event mediator itself, the outbox table entries will still be stored without risk of loss. This template guarantees delivery at least once.

There are several approaches to publishing a message to an Outbox table. The first is polling, the other transaction log.

A. POLLING PUBLISHER PATTERN

If the program uses a relational database, a very simple way to publish messages inserted into an outbox table is for Message Relay to poll the table for unpublished messages. He periodically queries the table. For example, using the query:

```
SELECT * FROM outbox ORDERED BY ... ASC.
```

Message Relay then publishes these messages to the message broker, sending them to the target message topic. At the end of processing, it deletes these messages from the outbox table.

Among the advantages are:

1. easy to implement
2. works with any SQL database.

Among the disadvantages:

1. it is difficult to publish events in order
2. not all NoSQL databases support this template [3].

The principle of operation of this pattern is shown in Figure 1.

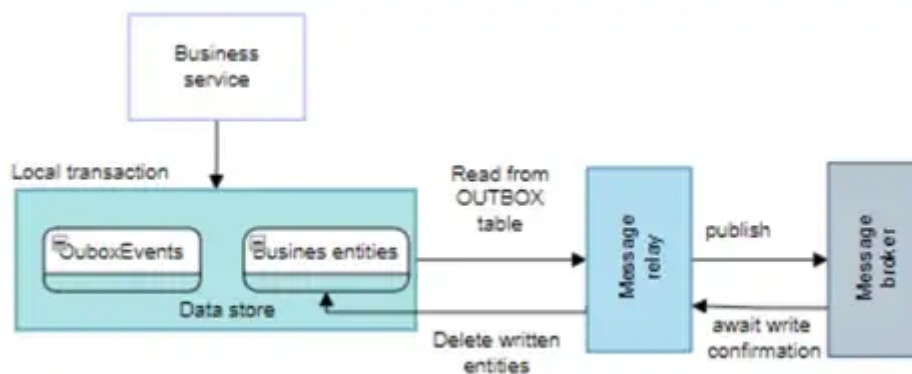


Figure 1. The principle of operation of the polling publisher pattern

Because of these shortcomings and limitations, it is often better — and in some cases necessary — to use a more sophisticated and efficient approach to database transaction logging.

B. TRANSACTION LOG TAILING PATTERN

Before considering this approach, the concept of Change Data Capture should be defined. *CDC (Change Data Capture)* is a process that identifies changes to data in

databases and provides information about those changes in real time or near real time.

This approach uses Message Relay to manage a database transaction log (also called a commit log). Each recorded update made by the program is presented as an entry in the database transaction log. The transaction log miner can read the transaction log and publish each change as a message to the message broker [2]. The principle of operation of this pattern is shown in Figure 2.

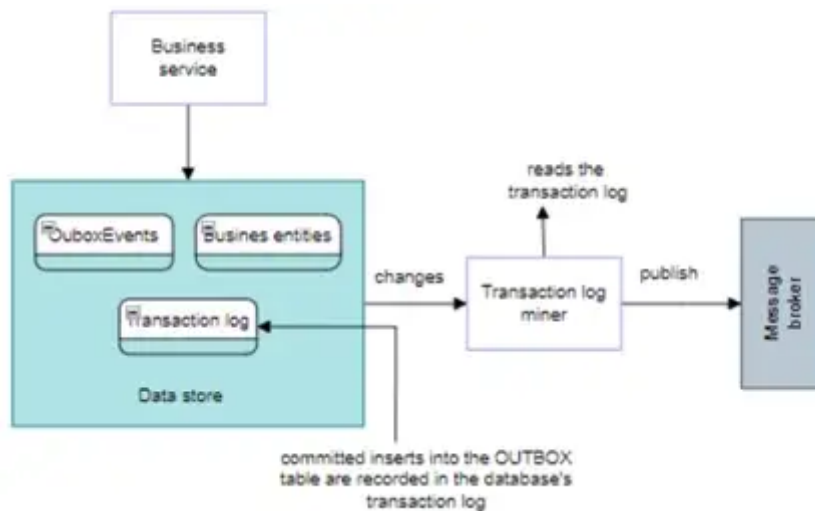


Figure 2. The principle of operation of the transaction log tailing pattern

The transaction log miner reads the transaction log entries. It converts each log entry into a message in corresponding format and publishes this message to the message broker. This approach can be used to publish messages written to the outbox table in an RDB or NoSQL database [2].

This pattern has the following advantages:

1. guaranteed accuracy
2. does not use a two-phase commit.

This template also has disadvantages:

1. complicated in relation to the previous one, although it is becoming more and more common
2. specific database solutions are required
3. it is difficult to avoid re-publication [4].

III. REASONS FOR CHOOSING DEBEZIUM

Debezium is a low-latency streaming platform that is designed primarily to implement the CDC (Change Data Capture) operation.

Debezium is based on Apache Kafka and provides compatible Kafka connectors that control specific database management systems. In particular, connectors are available for SQL Server, PostgreSQL, Oracle, MySQL, MongoDB, Db2, Cassandra, Vitess and others.

Debezium records the history of data changes in Kafka logs, where they are used by the written program. Even if the program stops suddenly, it will not miss anything. When the program restarts, it will resume starting from the event it stopped at. *CDC*. The main use of Debezium is the integration of CDC (Change Data Capture), which allows to capture and transmit real-time changes to data made in external databases.

Data monitoring. Debezium can continuously monitor, record and transmit row-level changes made to external database systems such as MySQL, PostgreSQL and SQL Server. It converts such external databases into event streams, thus allowing applications synchronized with the database to respond to and act on row-level changes made to database programs.

Data consistency. Because Debezium collects and stores CDC data based on logs, any real-time modifications or updates made to the database are securely stored and structured in an accurate sequence within the logbook.

Fault-tolerance. Because Debezium is a distributed platform, the program's architecture is designed to be fault-tolerant and flexible, even if any interruptions occur during continuous data transmission. Real-time event changes are replicated, stored, and distributed across multiple machines, thereby reducing the risk of information loss.

Data integration. Debezium can connect to various external database programs to constantly monitor and record row-level changes made to the database. It has a wide range of database connectors, such as MySQL and Oracle, which are built into the appropriate database to capture and stream changes in real time [5].

IV. IMPLEMENTATION USING TRANSACTION LOG PATTERN

The implementation will be based on the domain area of investment.

The architecture of the approach is shown in Figure 3.

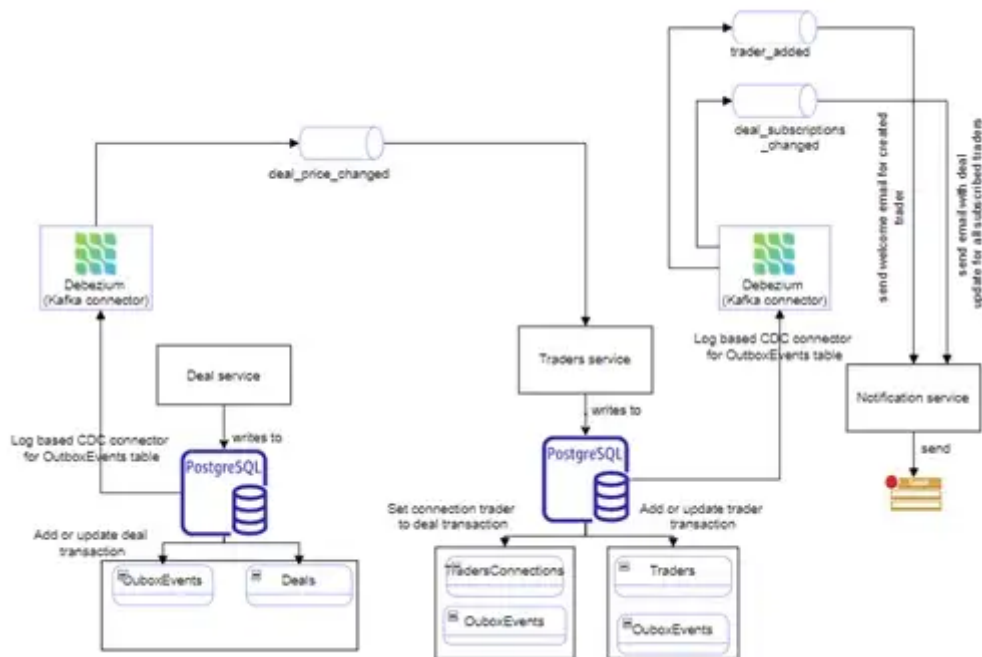


Figure 3. Implementation of the architecture using transaction log tailing pattern within the investment domain

Debezium connector publishes these changes to the Kafka topic subscriptions.events.

This topic on its turn is being monitored by the consumer of the Notification service, which sends messages with changes to the deals to all traders signed to this agreement, using the payload received from the message.

Traders service in case of adding a new trader, also makes an entry in OutboxEvents table and adds an entry to the Traders table with information about the new trader. And all this happens within one transaction.

Debezium connector publishes these changes to the Kafka topic traders.events. The consumer of the Notification service is subscribed on it. After any of messaged are received in topic, it sends a welcome message to the trader message for the registration in this system.

An example of payload that comes in kafka topic subscriptions.events is shown on Figure 4.

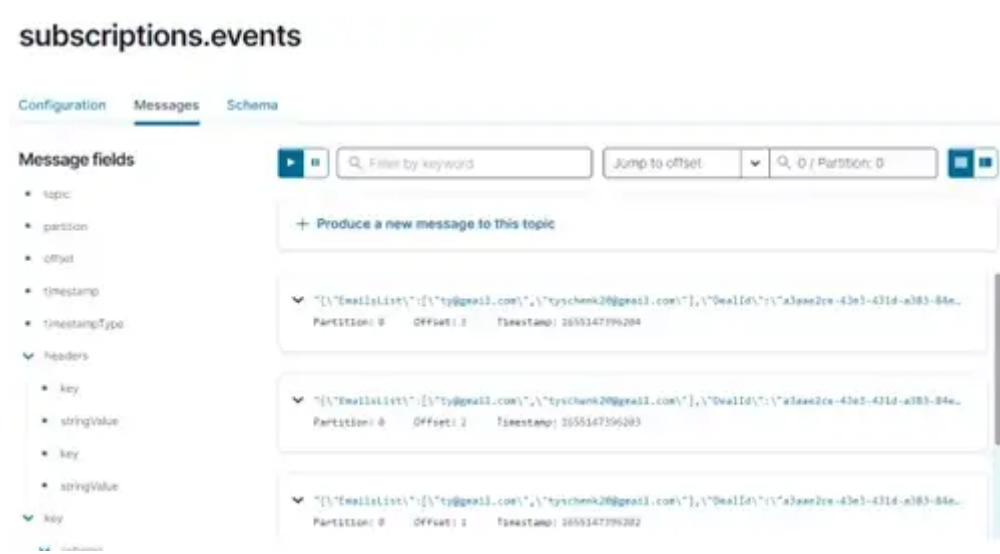


Figure 4. Payload coming to kafka subscriptions.events topic

An example data in OutboxTable shown on Figure 5.

	Id [PK] uuid	AggregateType text	Aggregate uuid	Type text	Payload text
5	869ca07b-91cc-4...	subscriptions	7d5f0...	DealSubscriptionUpdated	{\"EmailsList\": [\"tyshchenk20@gmail.com\"], \"DealId\": \"7d5f0271-37c3-4586-9481-6e7f302ee405\", \"RevisedPriceRangeLow\": 12
6	7c8cb4ef-93d1-45...	subscriptions	7d5f0...	DealSubscriptionUpdated	{\"EmailsList\": [\"tyshchenk20@gmail.com\"], \"DealId\": \"7d5f0271-37c3-4586-9481-6e7f302ee405\", \"RevisedPriceRangeLow\": 12
7	de926933-70cf-4...	subscriptions	7d5f0...	DealSubscriptionUpdated	{\"EmailsList\": [\"tyshchenk20@gmail.com\"], \"DealId\": \"7d5f0271-37c3-4586-9481-6e7f302ee405\", \"RevisedPriceRangeLow\": 12
8	cb22b511-0a3c-4...	subscriptions	7d5f0...	DealSubscriptionUpdated	{\"EmailsList\": [\"tyshchenk20@gmail.com\"], \"DealId\": \"7d5f0271-37c3-4586-9481-6e7f302ee405\", \"RevisedPriceRangeLow\": 12
9	df793cfe-bd0e-40...	subscriptions	7d5f0...	DealSubscriptionUpdated	{\"EmailsList\": [\"tyshchenk20@gmail.com\"], \"DealId\": \"7d5f0271-37c3-4586-9481-6e7f302ee405\", \"RevisedPriceRangeLow\": 12
10	a842d9c6-57ae-4...	subscriptions	7d5f0...	DealSubscriptionUpdated	{\"EmailsList\": [\"tyshchenk20@gmail.com\"], \"DealId\": \"7d5f0271-37c3-4586-9481-6e7f302ee405\", \"RevisedPriceRangeLow\": 12
11	3f5f4acd-c1b1-49...	dealSubscriptions	7d5f0...	TraderSubscriptionAdded	{\"TraderId\": \"f14490e4-0410-4d44-a2c5-9274167ceb2a\", \"DealId\": \"7d5f0271-37c3-4586-9481-6e7f302ee405\", \"Id\": \"00000000
12	ad6914be-55f4-4...	subscriptions	7d5f0...	DealSubscriptionUpdated	{\"EmailsList\": [\"tyshchenk20@gmail.com\", \"tyshchenk20@gmail.com\"], \"DealId\": \"7d5f0271-37c3-4586-9481-6e7f302ee405\", \"R
13	40bodfb5-4d3d-4...	subscriptions	7d5f0...	DealSubscriptionUpdated	{\"EmailsList\": [\"tyshchenk20@gmail.com\", \"tyshchenk20@gmail.com\"], \"DealId\": \"7d5f0271-37c3-4586-9481-6e7f302ee405\", \"R
14	d97c39af-2115-4...	subscriptions	7d5f0...	DealSubscriptionUpdated	{\"EmailsList\": [\"tyshchenk20@gmail.com\", \"tyshchenk20@gmail.com\"], \"DealId\": \"7d5f0271-37c3-4586-9481-6e7f302ee405\", \"R

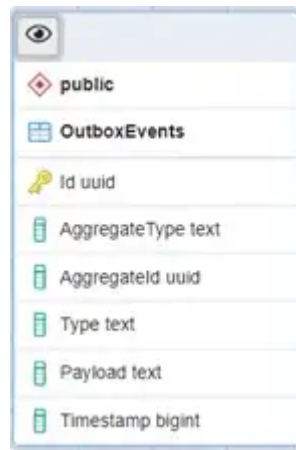
Figure 5. Data written in OutboxTable

Source code of implementation was posted on github:

<https://github.com/FairyFox5700/KafkaCDC>

A. DESIGNING OUTBOX TABLE

The structure of the Outbox table is the same for each service and looks like on Figure 5.



Field	Type
Id	uuid
AggregateType	text
AggregateId	uuid
Type	text
Payload	text
Timestamp	bigint

Figure 5. The structure of the OutboxEvents table

The type, aggregateType, and aggregateId fields provide event or message metadata. They are useful for handling an event in Apache Kafka or so that event users can filter the event they want to process.

The names of the fields and their purpose are described in table 1.

Table 1. Field names and their purpose for the outbox table

Field name	Purpose
Id	Unique event identifier. In the original message, this value is the header. Can be used to delete duplicate messages.
AggregateType	Contains the value that the transformer of an individual message adds to the name of the topic to which the connector sends the original message.
Type	Represents an additional field that allows to organize events.
Payload	Represents the event workload for the source topic. The default structure is JSON. As a rule, it marks only the body of the message that needs to be published to the consumer. However, it can be configured to include additional fields.
Timestamp	Time when message was sent. Type Int64 is used because the transformer can work only in a given format to represent the time and date of the message.
AggregateId	Contains an event key that is used to retrieve a payload id. It is used as a key in the original message to maintain the correct order of messages in Kafka topics.

B. DEBEZIUM CONNECTOR CONFIGURATION

Debezium connector configuration for transaction service:


```
{
  "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
  "tasks.max": "1",
  "database.hostname": "postgres",
  "database.port": "5432",
  "database.user": "postgres",
  "database.password": "root",
  "database.dbname": "kafkacdc.deals",
  "database.server.name": "postgres",
  "schema.include.list": "public",
  "table.include.list": "public.OutboxEvents",
  "tombstones.on.delete": "false",
  "transforms": "outbox",
  "transforms.outbox.type": "io.debezium.transforms.outbox.EventRouter",
  "slot.name": "debezium10",
  "transforms.outbox.table.field.event.id": "Id",
  "transforms.outbox.table.field.event.key": "AggregateId",
  "transforms.outbox.table.field.event.payload": "Payload",
  "transforms.outbox.route.by.field": "AggregateType",
  "transforms.outbox.route.topic.replacement": "${routedByValue}.events",
  "transforms.outbox.table.fields.additional.placement": "Type:header:eventType",
  "transforms.outbox.debezium.expand.json.payload": "true",
  "value.converter": "org.apache.kafka.connect.json.JsonConverter",
  "value.converter.schemas.enable": false,
  "plugin.name": "pgoutput"
}
```

The configuration of the debezium connector for trading service will be almost identical.

The main fields of the configuration are presented in table 2.

Table 2. The main configuration of debezium and its purpose

Configuration	Function
<i>transforms</i>	The "outbox" value is configured by the Debezium connector to support the Outbox template.
<i>transforms.outbox.route.topic.replacement</i>	The value of <code>outbox.event</code> . <code>{RoutedByValue}</code> places the message in the Kafka topic according to the specified template. When in the first record the value in the <code>aggregateType</code> column is <code>deals</code> and in the second record, the value in the <code>aggregateType</code> column is <code>traders</code> , then the connector sends the first record to the topic <code>outbox.event.deals</code> , and another in <code>outbox.event.traders</code> .
<i>transforms.outbox.route.by.field</i>	Specifies the name of the column in the outbox table. The default behavior is that the value in this column becomes part of the topic name to which the connector sends outgoing messages.
<i>transforms.outbox.type</i>	Installed in <code>io.debezium.transforms.outbox.EventRoute</code> . Additional configuration of the Debezium connector to support the outbox table pattern.
<i>value.converter</i>	Installed in <code>org.apache.kafka.connect.json.JsonConverter</code> as default.
<i>transforms.outbox.table.fields.additional.placement</i>	Specifies one or more outbox table columns to add to outgoing messages.
<i>value.converter.schemas.enable</i>	If the converter requires any additional configuration parameters, they can also be specified, for example, to disable schemas by setting this field to <code>false</code> .
<i>transforms.outbox.debezium.expand.json.payload</i>	Instructs the connector to display payload not as a string, by default, but as json.

The remaining fields have default values or are clear from the configuration name.

C. DEBEZIUM REST API CONFIGURATION

Debezium has a REST api for configuring and viewing the status of connectors.

1. [http://localhost:8083/connectors?Expand=info &expand=status](http://localhost:8083/connectors?Expand=info&expand=status) — view the status of the registered connector.
2. <http://localhost:8083/connectors/outbox-connector/config> — register or change the connector. Where 'outbox-connector' is the name of the connector.
3. <http://localhost:8083/connectors/outbox-connector> — remove the connector. Where 'outbox-connector' is the name of the connector.
4. <http://localhost:8083/connectors> — get all registered connectors.

More details on the configuration can be found in the [debezium documentation](#) [6].

D. INFRASTRUCTURE CONFIGURATION

The infrastructure is deployed using a docker.

The first part of all docker-compose is the database. In fact, it is the most important part of the whole configuration, as it will contain data that will be sent to various microservices.

Debezium has an image of a configured PostgreSQL image to use a transaction log pattern. The configuration is below:

```
image: debezium/postgres:13
container_name: postgres
environment:
  POSTGRES_PASSWORD: root
  POSTGRES_USER: postgres
ports:
  - 5438:5432
```

Next part is Zookeeper. It is used for reliable operation of Kafka. From the basic configuration, it is important to set the client port and time for health-checks.

```
zookeeper:
  image: confluentinc/cp-zookeeper:latest
  container_name: zookeeper1
  ports:
    - 2181:2181
  environment:
    ZOOKEEPER_CLIENT_PORT: 2181
    ZOOKEEPER_TICK_TIME: 2000
```

Next are the settings for Kafka. This is an important component of this architecture.

The following must be specified from the required settings:

KAFKA_ZOOKEEPER_CONNECT — the address where Kafka will communicate with Zookeeper.

KAFKA_ADVERTISED_LISTENERS — indicates how clients can get the hostname.

KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR — a value that indicates that a multicluster program is used if the value is greater than 1.

```
kafka:
  image: confluentinc/cp-kafka:latest
  container_name: kafka
  hostname: broker
  depends_on:
    - zookeeper
  ports:
    - 9092:9092
    - 29092:29092
  environment:
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT_HOST://kafka:29092
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
    KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    KAFKA_BROKER_ID: 1
    KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
```

The following settings for Kafka Connect. When the main architecture is running, you need to install the debezium connector itself, which will receive information from the database, in this case debezium/connect:latest is used.

Important settings:

BOOTSTRAP_SERVERS is the URL to connect to the Kafka cluster.

GROUP_ID — a unique identifier to identify the group of connection clusters to which this worker belongs.

CONFIG_STORAGE_TOPIC, OFFSET_STORAGE_TOPIC, STATUS_STORAGE_TOPIC — names of topics where connector and configuration data are stored.

```
connector:
  image: debezium/connect:latest
  container_name: kafka_connect_with_debezium
  ports:
```

```

- 8083:8083
environment:
  GROUP_ID: 1
  CONFIG_STORAGE_TOPIC: my_connect_configs
  OFFSET_STORAGE_TOPIC: my_connect_offsets
  STATUS_STORAGE_TOPIC: CONNECT_STATUSES
  BOOTSTRAP_SERVERS: kafka:9092
depends_on:
  - zookeeper
  - kafka
  - postgres

```

Also, for the convenience of viewing the content of topics and their management, configured the Confluent Control Center.

Open in app ↗

Sign up

Sign In



```

image: confluentinc/cp-enterprise-control-center:7.0.0
hostname: control-center
container_name: control-center
depends_on:
  - zookeeper
  - kafka
  #- schema-registry
ports:
  - 9021:9021
environment:
  CONTROL_CENTER_BOOTSTRAP_SERVERS: 'kafka:29092'
  CONTROL_CENTER_ZOOKEEPER_CONNECT: 'zookeeper:32181'
  #CONTROL_CENTER_KSQL_URL: "http://ksql-server:8088"
  #CONTROL_CENTER_KSQL_ADVERTISED_URL: "http://localhost:8088"
  #CONTROL_CENTER_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"
  CONTROL_CENTER_REPLICATION_FACTOR: 1
  CONTROL_CENTER_INTERNAL_TOPICS_PARTITIONS: 1
  CONTROL_CENTER_MONITORING_INTERCEPTOR_TOPIC_PARTITIONS: 1
  CONFLUENT_METRICS_TOPIC_REPLICATION: 1
  PORT: 9021

```

In fact, it is a web-based Apache Kafka management and monitoring tool with a user-friendly interface. It allows to get a quick overview of cluster performance, monitor and control messages, topic and schemas, develop and run ksqlDB queries.

To view the logs, Seq is configured. Configuration is below:

```
seq:
  image: "datalust/seq:2021"
  hostname: seq
  container_name: seq
  ports:
    - 5341:5341 # ingestion API
    - 5555:80 # ui
  environment:
    ACCEPT_EULA: "Y"
```

In fact, it is a real-time search and analysis server for structured application log data. It provides a user-friendly interface, an event store and a user-friendly query language. In general, it is used to monitor and diagnose problems in complex programs and microservices.

At the last stage we will configure and expose ports for microservices deals-service, traders-service, notification-service.

VI. CONCLUSIONS

In this paper, the problem of reliable publication of messages and data synchronization between microservices was considered. The architecture of the information system in the domain of investment was described, which solves this problem using transaction log tailing and debezium. Alternatives to using the outbox table pattern, as well as the use of CDC in these patterns, were also considered.

Streaming data using the CDC can also help with fault tolerance. Using this approach, the payload in the form of database records can be passed on to consumers in the form of a data stream, instead of going directly to the database, the consumer will receive the data as soon as it arrives at the topic to which the consumer is subscribed.

In addition, this approach solves the problem of guaranteed delivery. Namely, it guarantees exactly once delivery to subscribed consumers. By using the debezium connector, the full advantage of the transaction log tailing approach can be taken and ensured guaranteed accuracy, avoided two-phase commit, and implemented own logic for processing new data that appears in the outbox table.

References

[1] Morling G. *Reliable Microservices Data Exchange With the Outbox Pattern*. [Online]. Available: <https://debezium.io/blog/2019/02/19/reliable-microservices-data-exchange-with-the-outbox-pattern/>.

[2] C. Richardson, *Microservices patterns*, first ed., Manning, USA, 2018, 65–109 p.

Microservices patterns. Simon and Schuster.

[3] *Pattern: Polling publisher*, [Online]. Available at: <https://microservices.io/patterns/data/transaction-log-tailing.html>

[4] *Pattern: Transaction log tailing*, [Online]. Available at: <https://microservices.io/patterns/data/transaction-log-tailing.html>

[5] Ishwarya M. *Debezium vs Kafka Connect Simplified: 3 Critical Differences*, 2022, [Online]. Available at: <https://hevodata.com/learn/debezium-vs-kafka-connect>

[6] *Outbox Event Router*, [Online]. Available at: [Outbox Event Router : Debezium Documentation](#)

[Debezium](#)[Kafka](#)[Dotnet Core](#)[Outbox Pattern](#)[Microservices](#)[About](#) [Help](#) [Terms](#) [Privacy](#)

Get the Medium app

