TRIGO GmbH
Heiligenstädter Straße 28/3
1190 Wien
+43 (0) 699 10 38 41 85
hello@trigodev.com
www.trigodev.com

How to leverage Kafka & Kafka Connect and build system integration pipelines

Building system integration pipelines can be complex. At TRIGO we've been doubling down on Kafka to connect systems lately. In this white paper, we'll discuss why a two-way synchronization is important, common ways you can approach synchronization, and what role Kafka plays throughout the process.

Why flexible two-way-synchronization makes sense **3**

Common ways you can approach synchronization **3**

- Typical small to medium size solutions 3
- Handling large systems 4
- Centralized Synchronization Service 4
  - Advantages 4
  - Disadvantages 4
- Data Change Streams 5
  - Advantages 5
  - Disadvantages 5

What role does Kafka play? **6**

Kafka Connect: produce and consume events without writing code **7**

Example Kafka architecture you can use as a starting point **7**

- Fetching legacy changes 8
- Data Processing 9
- Updating the legacy system 10

What you could achieve with this architecture? **11**

All in all: System synchronization with Kafka is a winner **12**

Get in touch with us **12**

Why flexible two-way-synchronization makes sense

We've seen time and time again in our client work, that when it comes to large-scale projects, making a hard switch and replacing an old system with a new one from one day to the next — is just not feasible. You've got too many stakeholders involved, integrations that need to be carried out etc. And even if it's possible, the new system will need to be in place for some time to prove it's just as or ideally even more reliable than the legacy software, before the client can completely trust it.

So, quite a big part of a large-scale project, and its technical challenges, lie in implementing a reliable two-way synchronization mechanism between both worlds. Especially in large-scale systems with a ton of applications and more complex setups, the integration needs to be rock solid, easy to extend, and maintainable in the long run. Whenever you add a new service or migrate an existing component, you will need to adapt it, which is why the underlying code will need to change quite often — and have the flexibility to do so.

Common ways you can approach synchronization

There are a few different approaches and concepts when it comes to synchronizing decentralized systems. Generally speaking, we can distinguish between two different scenarios:

- The components know about each other. This is typical for small or medium size solutions with a few well-known integrations.
- Systems built of numerous independent and autonomous parts, e.g. microservices, that are unaware of each other.

Typical small to medium size solutions

- Push Change, each component knows about its counterpart(s) and is extended to actively push changes using Rest APIs, RPC interfaces of similar
- Pull Change, each component knows its counterparts and queries for changes

Generally speaking, this just works for small systems. For larger systems, this approach is insufficient and is far too complex. This often leads to dysfunctional changes and cascading errors. We won't go into much detail about that, as it's not something we would recommend.

Handling large systems

When it comes to large systems, you would normally design the synchronization around a centralized, specialized component or directly build it into the solution's architecture, enabling a decentralized implementation.

Centralized Synchronization Service

With this approach, you implement a single service that knows about all the integration parts and connected APIs, as well as performs the synchronization. We've often seen consultants build these type of solutions upon proprietary workflow or business process engines.

Advantages

- all the sync logic is in a single service ⇒ less moving parts

Disadvantages

- huge and complex
- requires profound knowledge about each interacting system, and its APIs
- single point of failure
- Vendor lock-in

Data Change Streams

Applications publish their changes to a "change log" from where other applications that are interested in these data entities can pull and process those changes.

Advantages

- Expandability, adding a new application does not require any central component or a data-producing application to be updated. The new application can simply "hook" into the data stream and start to consume other applications' data.
- Reduced Complexity, as each component knows how to publish changes, consume changes, the domain knowledge stays local to each service.
- Observability, as data change streams can be consumed by multiple systems, it's easy to build a monitoring solution to gather insights into the flow of data.

Disadvantages

- Each component has to implement the event publish and consumption logic. This disadvantage can be fully mitigated by using the application's database as the source for the data change stream (see Apache Kafka Connect)

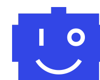
As you can see, the data change stream approach has its advantages that go above and beyond all other solutions. In our humble opinion, it's the only solution that can grow over time without creating a system you will no longer be able to manage.

To counteract the problem of introducing a single point of failure, Kafka itself is deployable in a high-available and scalable manner.

Now, you may ask yourself, how does that work with enterprise applications that have been around for 20 years+? They don't know anything about an event stream, how to publish and consume events.

This is true, and it's definitely the most complex issue.

The great thing is, most of those applications tend to be SQL database-based applications. If it's not possible to extend the application's code to directly publish its data changes, we can search the database for those changes.



This is for sure a complex task, but it's not specific to our event streaming solution. Every synchronization mechanism needs this function in one way or the other. The same goes for importing the changes from the new application, without being able to consume changes, the integration won't work.

When dealing with SQL Database-based applications, you will always also have the possibility to use the database directly — i.e., maintaining change log tables using triggers, etc.

What role does Kafka play?

Kafka is our data change log that captures data changes from all connected systems and stores them in a central place. To organize different types of changes, Apache Kafka introduces the concept of topics.

Apache Kafka and its topics are in many ways similar to message queues. You put a message into it, that gets consumed from one or many consumer applications. What distinguishes Apache Kafka from most message queues is, that it is perfectly suitable to act as a persistent data store. In most queuing systems, messages get deleted once they are consumed.

A topic is meant to contain events of the same type or that somehow belong to each other. The organization of the topics and the definition of the event data structures is key for providing a future-proof and expandable solution.

As a general rule of thumb, topics should only contain a single type of events and the events should be structured to provide a service-independent general representation of the data. e.g. a `PersonAdressChanged` event is structured in a way that is easily produced and consumed.

All connected systems in such an event streams-based application need to be able to produce and interpret those events.

Kafka Connect: produce and consume events without writing code

As Apache Kafka acts as the engine of our event stream, we also need a way to easily — in the best case without writing any code — produce and consume such events. The Apache Kafka Connect server acts as such a container for those event producers and consumers.

In Apache Kafka Connect terms, those are called sink and source connectors.

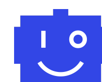
Source connectors query the 3rd party system and generate Apache Kafka Messages. The sink connectors subscribe to an Apache Kafka topic and translate those messages for the connected system.

We deploy the connectors as independent plugins into an Apache Kafka Connect cluster and configure them using the Rest API that is exposed for that purpose. Connectors are available for most of the currently used systems in the typical enterprise landscape, and of course for almost all databases, message queues, cloud infrastructure services, etc. to allow the implementation of custom solutions.

(https://docs.confluent.io/home/connect/kafka_connectors.html)

Example Kafka architecture you can use as a starting point

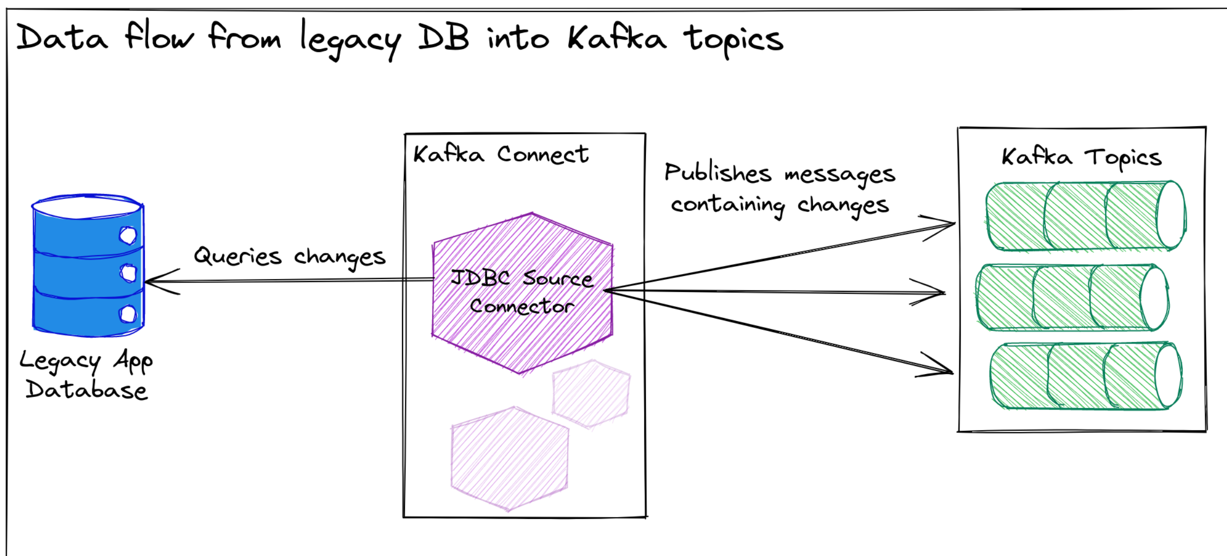
We will now take you through an example, we make some assumptions, and they might not match your specific use cases. It can, however, act as a great starting point to demonstrate how two-way sync might work. If you have any questions, don't hesitate to drop us an email at hello@trigodev.com — we'd be happy to answer them.



Overall, we can assume that:

1. Our new application architecture uses events as an asynchronous communication pattern for the different services to integrate.
2. Legacy System provides access to a changelog. In this particular example, we assume that a SQL Server is available with changelog tables.
3. We do not cover security or access management. Those are separate topics not related to the synchronization concept.
4. We assume that the databases and Kafka run in a trusted environment

Fetching legacy changes



Considering a setup that uses a JDBC source connector. The legacy data import may look similar to this diagram:

JDBC Kafka Connect Source Connector(s) query the change log tables for recent changes and publish those changes into legacy import topics. The Kafka Connect connector handles all the complex logic to fetch and process changes in a proper and reliable way. With properly built change log tables, providing timestamp/ordering information, we can bring our data into Kafka without a single line of code or deployment.

Data Processing

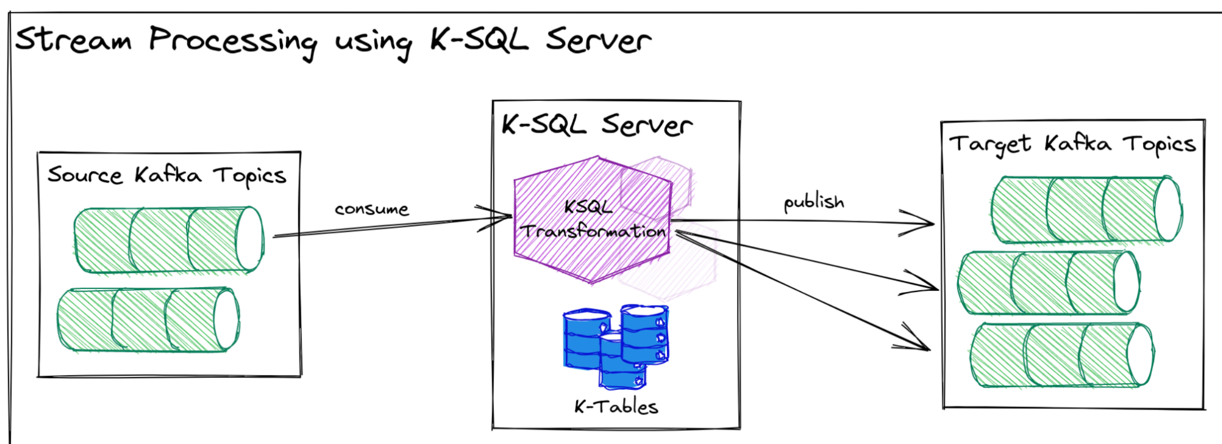
Depending on the actual data source and structure, for simple structures, you can configure the source connector to generate the final data change events the system expects, and it will publish the messages directly into the corresponding topics. In that case, we finished our work, as our new services already subscribe to the change events and import the updates.

Quite often, this is not possible due to more significant differences in the domain models and data structures between the old and the new system. We need to implement some form of transformation logic. E.g., the old system uses a single table containing users and their addresses, but the new implementation handles those entities isolated.

A single imported change from a legacy system would need to be transformed into a *UserChangedEvent* and a *AddressChangedEvent*.

We call this kind of transformation stream processing and can be achieved by using either K-SQL, a SQL-based Stream Processing server, or by implementing a Custom Kafka Streaming Application.

Both approaches solve the same issue. Transform messages from the input topic to one or more messages and publish them to the output topic(s).



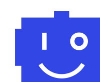
Whether to use K-SQL or a custom application depends on the actual use case. You may hit some K-SQL feature limitations in terms of complexity that may require you to write a custom streaming application.

Updating the legacy system

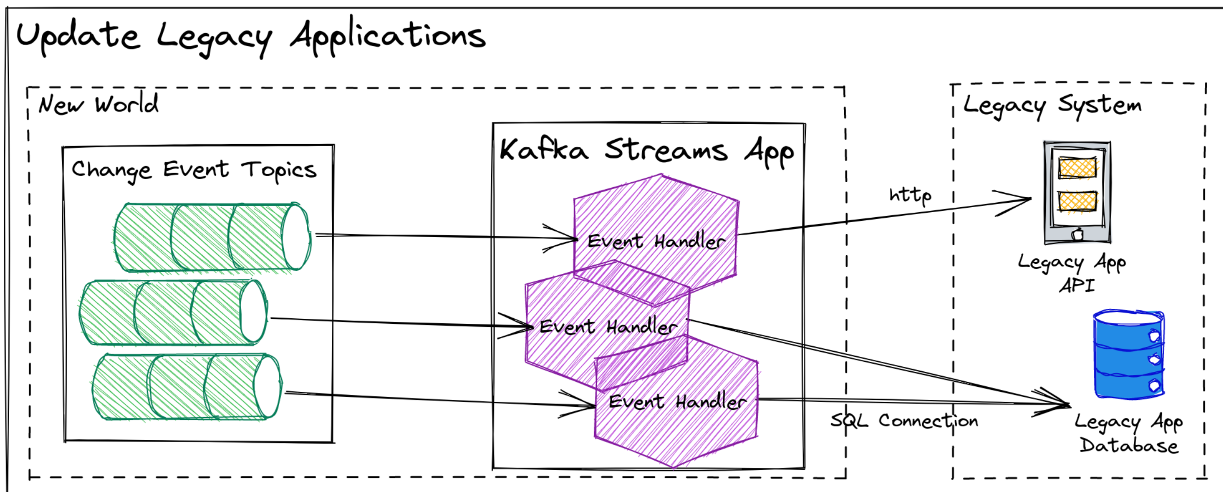
All we did up to this point was to make sure that updates from the legacy application become visible to our new system. This was achieved using a legacy import pipeline that transformed changes from the existing applications to our system's native change events that can be processed by our apps & services.

What's still missing is the other direction, changes from the new services to be pushed into our legacy application.

As before, all changes that need to be imported are already available in the system's Apache Kafka topics containing the corresponding events. Now we can implement a set of stream processing applications that consume those change events in the same way our replacement applications do and import the changes by updating the legacy system's database.



Of course, when available, the streaming application may also call APIs or use other interfaces provided by the legacy system. SQL Database access is just an example.



What you could achieve with this architecture?

Using an approach like this, we managed to solve a bunch of problems, and you can too.

- We separated the logic by type of change in the import/export implementation
- We converted changes triggered by the legacy application to our platform's native change stream, format.
- By integrating each type of change (or closely related changes) it is easy to remove those parts whenever the legacy system is finally deprovisioned.
- Kafka topics, correctly configured, provide us with full historical data for all changes. This allows us to reprocess those changes later when something failed to correct errors.
- Big parts of the system are "no-code" — the best way to avoid bugs
- Clear separation of the data, in each stage of the pipeline.

All in all: System synchronization with Kafka is a winner

System synchronization, especially in large-scale systems is, and will always be, a field of various complex problems. There is no one-fits-all solution, and it always depends on the actual environment and requirements, how you should best handle all the details, and special cases found in real-life systems.

The synchronization solution, we described here, is based on architecture concepts, tools and, design patterns that have proven to work in several real-life implementations. Kafka's advantages outweigh its drawbacks, and we're convinced it's the way to go if you want to future-proof your system, keep it flexible and easy to manage.

Get in touch with us

Did you find this white paper helpful, or maybe even have some constructive criticism? Get in touch and let's continue the discussion.