Kafka

Shared Kafka cluster is used for non-prod environments, so isolation is achieved with proper naming conventions and permissions management.

Naming rules

1. Topic name must be defined in such a way to make the business purpose clear. Some examples: *antifraud-requests*, *contact-prepared*, *contract-signed-by-bank*, *email-responses*. Naming pattern *${domain_resource}-events* is reserved for REST API conventions.

2. To have the same topic running in multiple environments the name of the environment is attached to the topic name as a suffix. For example, *antifraud-requests-dev* and *antifraud-requests-uat*.

3. Topic versioning could be introduced for some reason like backward-incompatible changes in the event structure. Then version could be specified right after the logical topic name. For example, *antifraud-requests-1-dev*.

4. Each consumer group must have a unique name to avoid overlapping events consumption. For services, the service name must be used as a consumer group ID. For local access, each developer must define a dedicated unique consumer group ID.

Proposed naming rules (DRAFT)

According to domains encapsulation policies topic names should have domain name and purpose/visibility in addition to the logical name. The proposed approach is based on this article.

1. Topic name has the following format: *${env}.${domain}.${classification}.${description}*. '.' is used as a separator between parts. Each part is in lower case with '-' between words. Parts explanation:

- *env* could be either *dev*, *uat* or *prod*.
- *domain* is the official name of the domain owning the topic (like *cards*, *clients*, *payments*, etc.);
- *classification* is used to specify the type and purpose of the topic (like *events*, *cdc*, *requests*, *responses*, *internal*, etc.);
- *description* is a logical name of the topic.

2. Here is how examples from the current naming conventions are transformed:

- *antifraud-requests-dev → dev.clients.requests.antifraud*;
- *contact-prepared-uat → uat.clients.events.contract-prepared*;
- *email-responses-prod → prod.clients.internal-responses.email*.

Open questions:

1. Do we need to add a zone prefix (DMZ, LAN) in the topic name?
2. Does AVALaunch resources provisioning support this new schema?

Usage patterns

1. All accesses to the Kafka cluster must be secured (SSL/TLS configuration is needed). For each service dedicated certificate is generated to manage READ/WRITE permissions and access to particular topics.

2. Topics are divided into EXTERNAL (part of public API) and INTERNAL (used only for internal communication between services). So there are could be two Kafka clusters configured in the same service (logical separation is needed to avoid issues with the physical separation of Kafka clusters). To simplify configuration Spring Boot Kafka starter library could be used.

3. Each topic must have a properly defined retention period to specify how long events are stored in the Kafka cluster. Don't rely on default configuration because it is almost never correct for a particular use case.

4. Each event must specify what partition key to use. It is critically important to not rely on random partition selection when events must be processed in a particular order. Usually, some unique business ID is used for this purpose.

5. It is important to specify a retry policy for each consumer group to implement needed failover behavior (retry forever, store failed events in dead letter queue, store event data in DB for future usage, log failed event, and forget, etc.). Concrete policy depends on the business scenario but using default implementation could cause lost events.

6. Each event should have metadata either included in the event body or passed as additional headers. Following metadata fields are mandatory:

- unique event ID to enable idempotent processing on the consumer side;
- event creation time for better tracing.

Schema management

1. Each topic must aggregate events of the same structure to avoid conditional logic on the consumer side and possible deserialization errors.

2. Event structure should be defined in the Avro schema format and stored in a dedicated Kafka schema registry. Then additional validation could be done at the producer level to verify event structure before sending it to the Kafka topic.

3. DTO classes could be generated from the Avro schemas automatically and stored as separate versioned JAR to reuse across all services using a particular topic.

Frameworks and tools

1. To simplify development with Kafka either Spring Kafka or Kafka Binder for Spring Cloud Stream must be used instead of a low-level Kafka Java client. The first option works well for simple communication with Kafka, the second is well suited for event streaming and data processing.

2. Distributed tracing must be configured with Spring Cloud Sleuth. Kafka is supported out of the box so no additional configuration is needed.

3. For local access to the non-prod Kafka cluster KafkaTool could be used with dedicated credentials.