Event-driven APIs and schema governance for Apache Kafka: Get ready for Kafka Summit Europe 2021

Author: **Hugo Guerrero**, APIs & Integration Technical Evangelist, Red Hat

As a developer, I'm always excited to attend the Kafka Summit, happening this year from May 11 to 12. There are so many great sessions addressing critical challenges in the Apache Kafka ecosystem. One example is how changes to event-driven APIs are leading developers to focus on contract-first development for Kafka.
In preparation for the upcoming Kafka Summit, this article discusses the journey Kafka users have taken to get on the API bandwagon and how developers are using contracts to describe brokers without losing control of their data in the cluster. A critical component for effective schema governance is having a schema registry such as Apicurio Registry. See the end of the article for information about Red Hat's sessions during the Kafka Summit Europe 2021.

Note: *Contract-first development* is a design approach where business users or developers agree on a service's desired structure and result upfront, before sharing it with the development team.

Distributed, decoupled, and highly connected systems

Modern application development has shifted toward distributed, decoupled, and highly connected systems over the past several years. Distributed applications involve deployments across multiple data centers, cloud providers, and geographical regions.

Multiple teams working on different applications require the components to be decoupled in terms of technology, development, and even time. The last type of decoupling is critical for systems that produce events and data changes to be consumed by applications at a future time. In some cases, the application is not online when the change is made. In other cases, the application that will consume the events and data doesn't even exist yet. Finally, no application lives in a vacuum: Every application must be appropriately and efficiently connected to other components or legacy applications.

In response to these demands, we have seen the rise of microservices, HTTP-based applications using a request-and-response pattern. Microservices offer a better way for teams to develop their domain services and applications without impacting other groups. REST APIs simplify connections between applications using the well-understood, debuggable HTTP protocol. Simultaneously, we've seen the revival of event-driven architectures, powered by the rise of modern message brokers. All of these factors have converged in a turn toward democratized processing patterns such as event stream analytics.

Bottlenecks in the code-first approach

Implementing an event-driven architecture using Apache Kafka alongside the traditional API approach has brought new challenges and expectations. The conventional code-first workflow (of implementing the code first and then sharing the resulting API specification) includes many bottlenecks that prevent efficient progress. Developers are seeking a new direction for discoverability and access to event-stream endpoints.

The code-first process leads to questions about how to address the relationship between event providers and consumers. From a developer's perspective, the Apache Kafka endpoint must be clearly documented. The documentation should include information about the Kafka bootstrap server or the cluster’s security model, such as authentication. In this way, the documentation becomes a description of your Kafka brokers.

On the other hand, Apache Kafka transfers the responsibility for processing data structures to clients. This delegation allows Kafka to handle high throughput and be

highly scalable. But it also requires that developers are aware of the data format and data validation when sending or receiving data from Kafka topics. It's easy to share this information through informal channels within small teams, but when you increase the reach to external users and third parties, it can become a nightmare. Addressing these issues is an important challenge.

Event-driven APIs and contract-first workflows

Having a simple format for sharing contracts for your services among all of its users is highly valuable. The contract-first process creates the contract for the service beforehand. In this way, producers and consumers know upfront what to expect from the service provider.

Knowing the endpoint definition in advance allows developers to work independently. It also ensures consistency between both parties. It provides strong guarantees about service contracts, so that teams, users, and partners can collaborate more effectively. The contract-first approach also lets developers save time by using code generators and testing tools.

Distributed services using REST APIs have promoted the contract-first workflow by consolidating the OpenAPI specification to model their endpoints. The rallying around this format helped create a new practice of managing API contracts as products by themselves. As a result, specification editors such as SwaggerHub and Apicurio, OpenAPI code generators, and additional tooling have become available. Going further, the specification document can even help to mock services using platforms like Microcks.

Another secret weapon for handling contract-first agreements is the AsyncAPI specification. AsyncAPI is an open source initiative that seeks to improve the current state of event-driven architectures. The AsyncAPI specification describes event-driven APIs using messaging technologies like MQTT, AMQP, and Apache Kafka. It started as a sister specification to OpenAPI, using the same syntax and JSON Schema under the hood. Like any contract defined with OpenAPI, AsyncAPI helps you achieve visibility and agreement for your event-driven APIs.

AsyncAPI: Schemas as event contracts

If you are familiar with OpenAPI, you will easily recognize the similarities between the AsyncAPI and OpenAPI specifications. The server information is still there, paths become *channels*, and operations are simplified to *publish* and *subscribe*.

However, the most crucial part is the *message section*. This section defines the content type and structure of a message's headers and payloads. Developers coming from the Kafka world will see that, in practice, this part of the specification is for defining the messages' schema. The message section can also refer to schemas of different types, such as an Apache Avro or JSON Schema. It can also include examples of payloads and headers.

AsyncAPI reflects the need to express the Kafka notion of schemas for the data sent and received from Kafka, tackling one of the producer's and consumer's challenges in different teams. In the path to event-driven APIs, schemas have become the contracts used for events. For this reason, the same benefits that the contract-first strategy derives from conventional REST APIs apply to event-driven APIs.

To better understand the value of this approach, consider the following scenario: A producer sends data to Kafka, and a consumer retrieves the data. However, Kafka's asynchronous communication does not allow the producer to know who will consume the data, or when. Also, because the data stored in Kafka topics is in a shared state, new clients can process the data. If the producer team makes a breaking change in the record structure, they might reach out to the original consumer team about the change. But new clients added later will likely miss that information, so they'll keep using their old deserialization process. As a result, the execution will break when the latest events arrive, as shown in Figure 1. All of this is why a central registry where data schemas are stored and made available for consumers is critical to effective governance.

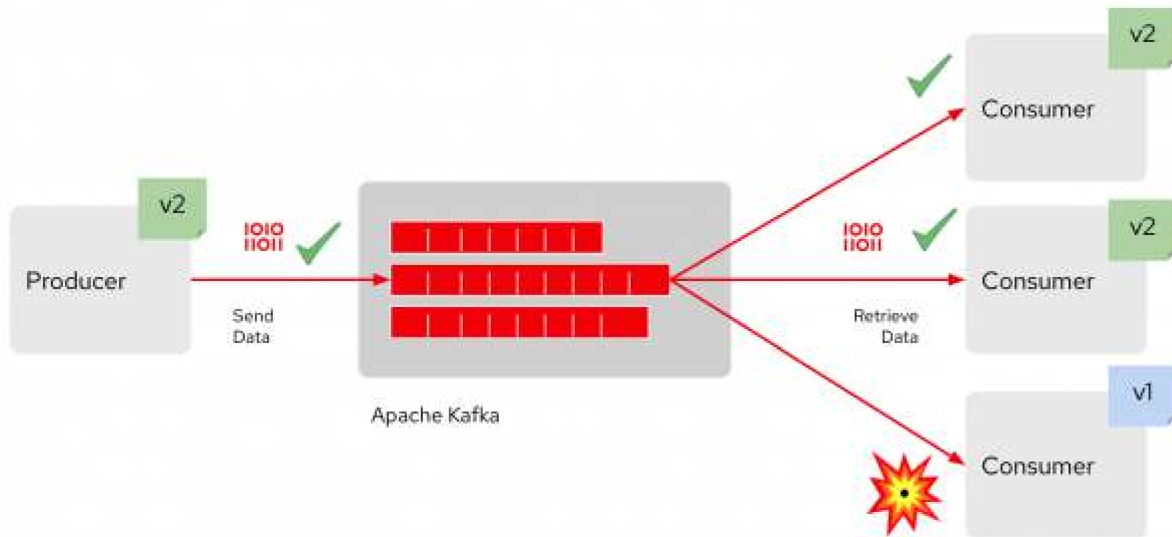

Figure 1: A change to the schema creates a compatibility problem.

Apicurio: A registry for event schemas

One common way to solve the schema governance problem is to use a *registry*. This registry needs to address three main capabilities for successfully managing schemas: First, the registry needs to manage artifacts comprehensively. It must include the ability to store, browse, retrieve, and search the managed items. Second, it must support different data structures, like Apache Avro, Google Protocol Buffers, and JSON Schema. Finally, the registry should use rules about validity and compatibility to follow and control the evolution of the artifact's content.

If the registry implements these capabilities, it can become the central repository for schemas where producers and consumers share their data structure and Kafka topics. However, following the suggested approach for contract-first development, the registry should not be only for schemas. The registry should also store the definition of the *endpoint* and *channel*, as implemented by the Kafka brokers and topics. In this way, you can provide a streamlined experience for developers.

Coming back to our example from Figure 1, let's say that the producer client's serializer code could reach a registry API and query for the schema to use to encode the new data. There are several ways to feed schemas to the registry, from the producer doing

self-registration to another team managing the schemas. After retrieving the correct schema, the producer could use it to serialize the data in the expected format and send it to Kafka. Finally, the consumer would use the same strategy to retrieve the schema from the registry to deserialize the data, as shown in Figure 2.

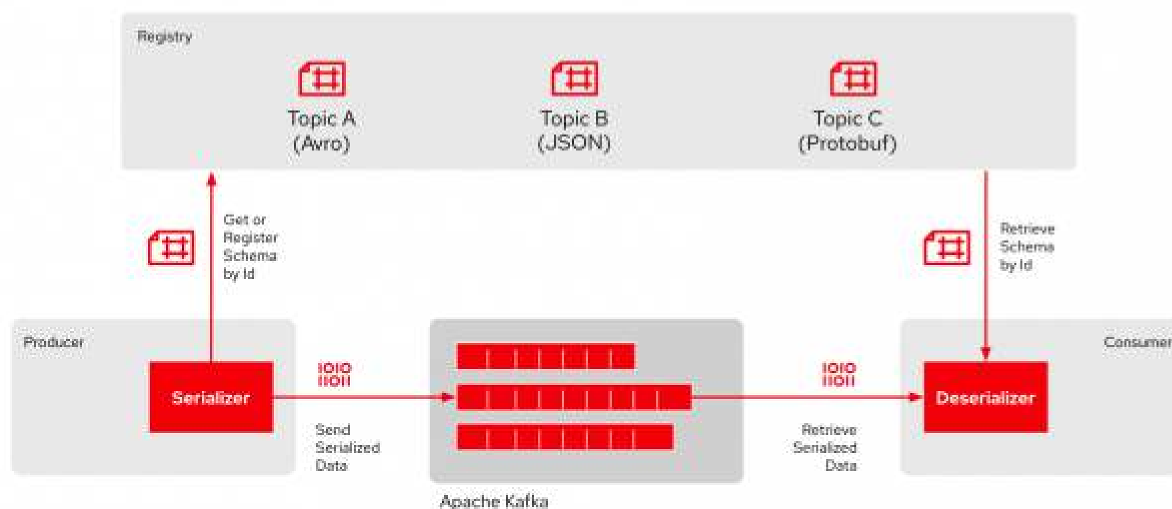

Figure 2: A registry for schemas in action.

Conclusion

The shift in modern application development has pulled event-driven architecture back into the spotlight. Kafka is undoubtedly on the rise and is now ubiquitous. The evolution of APIs allows developers to evolve from traditional messaging systems to fully enabled, event-driven APIs.

Adopting the contract-first model is critical for easing this evolutionary process. In a contract-first approach, you can use the AsyncAPI to define access to your Kafka brokers and topics in the same way you have used OpenAPI to define HTTP endpoints. The Kafka record schema becomes an *event contract*, and as such, needs to be managed as a product in itself. Finally, having a registry for schemas helps you with schema governance.

As a schema registry, Apicurio Registry is an end-to-end solution for storing API definitions and schemas for Kafka applications. The project includes serializers, deserializers, and additional tooling. The registry supports several types of artifacts, including OpenAPI, AsyncAPI, GraphQL, Apache Avro, Google Protocol Buffers, JSON,

Kafka Connect, WSDL, and XML Schema (XSD). Apicurio also checks the validity and compatibility rules for these artifacts.

For developers who want an open source development model with enterprise support, Red Hat Integration lets you deploy your Kafka-based event-driven architecture on Red Hat OpenShift, the enterprise Kubernetes. Red Hat AMQ Streams, Debezium, and the Apache Camel Kafka Connector are all available with a Red Hat Integration subscription.

Note: We've just announced the new fully managed service with Red Hat OpenShift Streams for Apache Kafka. Get started now with a free Apache Kafka cluster!