MICROSERVICES

Microservices, Apache Kafka, and Domain-Driven Design

KAI WAEHNER

JUNE 26, 2019

Microservices have a symbiotic relationship with domain-driven design (DDD)—a design approach where the business domain is carefully modeled in software and evolved over time, independently of the plumbing that makes the system work. I see this pattern coming up more and more in the field in conjunction with Apache Kafka®.

In these projects, microservice architectures use Kafka as an event streaming platform. Domain-driven design is used to define the different bounded contexts which represent the various business processes that the application needs to perform. These are joined together with events, creating a unidirectional dependency graph that decouples each bounded context from those that arise downstream, to create rich event streaming business applications.

This blog post discusses why Apache Kafka became the de facto standard and backbone for microservice architectures—not just replacing other traditional middleware but also building the microservices themselves using DDD and Kafka-native APIs like Kafka Streams, ksqlDB, and Kafka Connect.

Microservices

Everybody is talking about creating an agile and flexible architecture with microservices, a term that is used today in many different contexts.

Although microservices are not a free lunch, they do provide many benefits, including decoupling. Decoupling is the process of organizing a system around business capabilities to form an architecture that is decentralized. Smart endpoints and dumb pipes ensure the following:

> *Applications built from microservices aim to be as decoupled and as cohesive as possible – they own their own domain logic [that applies to their part of the business problem], and act more as filters in the classical Unix sense – receiving a request, applying logic as appropriate and producing a response.*
>
> Martin Fowler

Apache Kafka – An event streaming platform for microservices

What technology should you use to build microservice architectures? This can be answered in two parts:

1. How should microservices communicate with each other?

When we consider microservice communication, the approach that most people start with is REST, i.e., communication with synchronous HTTP(S) calls. This works well for many use cases. However, the request-response pattern creates point-to-point connections that couple both sender to receiver and receiver to sender, making it hard to change one component without impacting others.

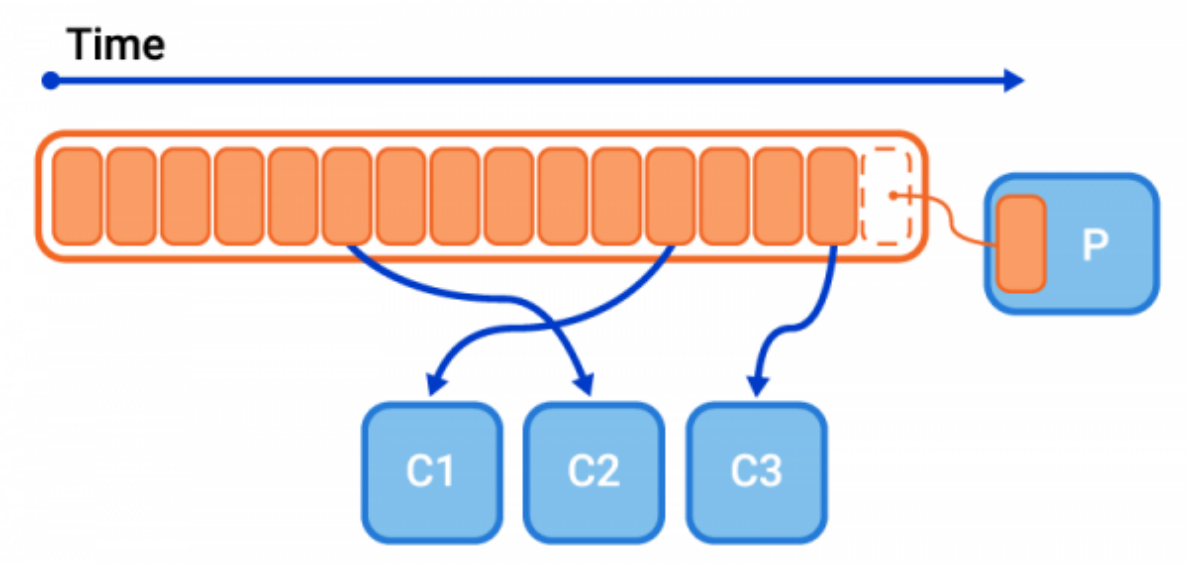
Due to this, many architects use middleware as a backbone for microservice communication to create decoupled, scalable, and highly available systems. Middleware can be anything—some custom glue code or framework, a messaging system like RabbitMQ, an ETL tool like Talend, an ESB like WSO2, or an event streaming platform like Apache Kafka.

2. What middleware do you use–if any?

The main reason why Apache Kafka became the de facto standard for microservices is its combination of three powerful concepts:

1. **Publish and subscribe** to streams of events, similar to a message queue or enterprise messaging system
2. **Store** streams of events in a fault-tolerant way
3. **Process** streams of events in real time, as they occur

With these three pillars built into one distributed event streaming platform, you can decouple various microservices (i.e., producers and consumers) in a reliable, scalable, and fault-tolerant way.



To better understand the benefits of Apache Kafka compared to traditional middleware like MQ, ETL, or ESB tools, see Apache Kafka vs. Enterprise Service Bus – Friends, Enemies or Frenemies?

Let's now understand how an event streaming platform like Apache Kafka is related to domain-driven design.

Domain-driven design (DDD) for building and decoupling microservices

Domain-driven design (DDD), first coined in a book by Eric Evans, is an approach used to build systems that have a complex business domain. So you wouldn't apply DDD to, say, infrastructure software or building routers, proxies, or caching layers, but instead to business software that solves real-world business problems. It's a great technique for separating the way the business is modeled from the plumbing code that ties it all together. Separating these two in the software itself makes it easier to design, model, build, and evolve an implementation over time.

DDD is built on the following premises:

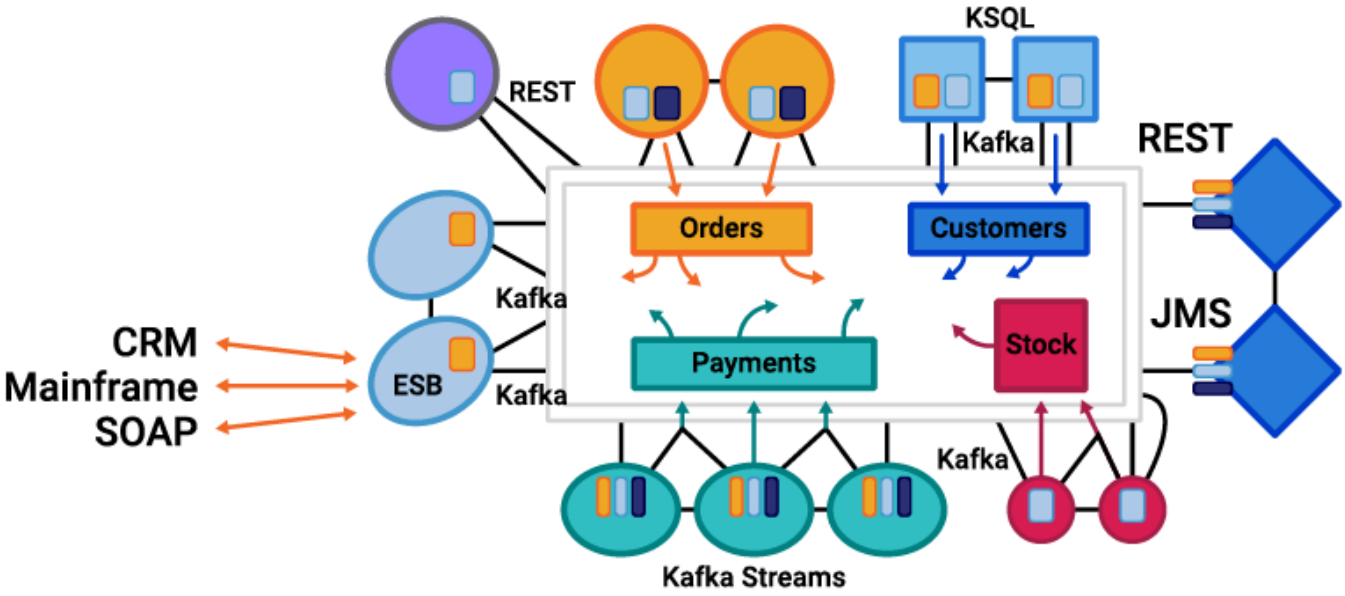
- Capture the domain model, in domain terms, through interactions with domain experts
- Embed the domain terminology in the code
- Protect the domain knowledge from corruption by other domains, technical subdomains, etc.

A main concept of DDD for this discussion is the bounded context. Large projects often have many different domain models and many bounded contexts. Yet when code in different bounded contexts is combined, software can become buggy, unreliable, and difficult to understand. Communication

among team members becomes confusing, and it is often unclear in what context a model should not be applied.

Therefore, DDD prescribes that we explicitly define the context within which a model is applied. We are required to set boundaries in terms of which team owns that model, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keeping the models strictly consistent within these bounds makes each part simpler to implement and understand, because we only need worry about a single bounded context. We aren't distracted or confused by code that may have leaked in from others. As Dan North puts it: Build code that fits in your head.

In relation to an event streaming platform, it could look something like this:



Here, each microservice has its own bounded context. From a technology perspective, this could include different APIs, frameworks, communication protocols, and datastores. Some will follow the request-response pattern, and others will use events depending on the problem that needs to be solved. Regardless, each will be a separate bounded context and have both a separate domain model and mappings between that model, business processes, and the data it shares with others.

So why is Kafka such a good fit?

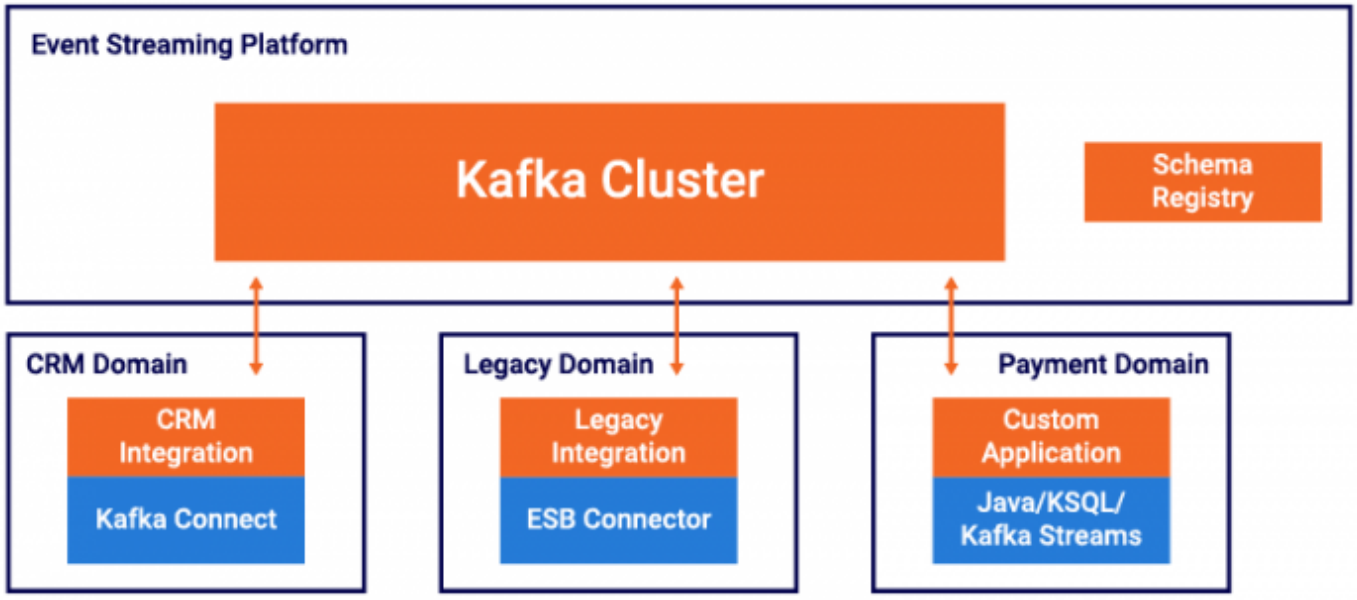
Apache Kafka and domain-driven microservices

Apache Kafka combines messaging and storage so that different producers and consumers are fully decoupled:

- The server side (Kafka broker, ZooKeeper, and Confluent Schema Registry) can be separated from the business applications.

- Producers do not know or care about who consumes the events they create. Kafka handles backpressure, scalability, and high availability for them.
- Producers can produce while consumers are down.
- New consumers can be added at any time, even if they need to start consuming events from an earlier timestamp.
- Consumers can process data at their own speed (batch or real time).
- Consumers can reprocess data again and again (e.g., to train different analytic models or to recover from errors or data corruption).

With these characteristics, each project team owns its own domain, including responsibilities, SLAs, versioning, and technology choices.



This may apply not just to business applications, but also to operations within the company's IT team, which owns the Kafka cluster for internal self-service offerings. The deployment of the Kafka cluster is often executed on a PaaS infrastructure, such as Kubernetes and Confluent Operator. In cloud deployments that leverage managed services like Confluent Cloud, such infrastructure teams are not usually required.

Domain models, bounded contexts, and ubiquitous language

As discussed above, one of the key elements of domain-driven design (DDD) is separating a business problem into a collection of independent bounded contexts. Each context has a domain model, which in software fully encapsulates the data it needs, the business operations it needs to perform, and the language used to describe each of them. But how does a domain model in one bounded context relate to a domain model in another? How do we ensure that changes in one model do not adversely affect the domain models in others?

The answer is to use what DDD calls an anti-corruption layer: a layer that maps the data used in a domain model to the data transmitted between different microservices or different bounded contexts. This pattern is implementation agnostic, meaning it can be used regardless of whether your services communicate through events or with a request-response protocol. In both cases, there is typically a wire format (be it the schema used to return data from a REST endpoint or the schema used to describe an event, such as an Avro message stored in the Schema Registry).

The anti-corruption layer has two goals:

1. It insulates the domain model from change
2. It encapsulates the boundary between contexts, describing how they map both in a technical sense—field A in a message maps to field B in the model, but also in terms of DDD's ubiquitous language—the *counterparty* in the event schema maps to the *customer* in the domain model

For the evolution of the model in each bounded context, the interfaces that join them together are a critical part of the design process that teams should go through. This should always performed by first establishing a ubiquitous language not constructed solely by developers but rather is developed in conjunction with the business stakeholders that own the system.

Connecting domains with Apache Kafka, Kafka Streams, ksqlDB, and Kafka Connect

There is one important point we have not yet discussed: Apache Kafka is not just a messaging system or integration layer—it's an event streaming platform. This means that it is not just responsible for providing the middleware that decouples microservices, but also allows you to perform complex data operations like splits, joins, filters, and summarizations in your client code. Here is another key difference between Apache Kafka and traditional middleware, as explained by ThoughtWorks:

> *…we're seeing some organizations **recreating ESB antipatterns with Kafka**by centralizing the Kafka ecosystem components — such as connectors and stream processors — instead of allowing these components to live with product or service teams. This reminds us of seriously problematic ESB antipatterns, where more and more logic, orchestration and transformation were thrust into a centrally managed ESB, creating a significant dependency on a centralized team. We're calling this out to dissuade further implementations of this flawed pattern.*
>
> Recreating ESB Antipatterns with Kafka

This point is important. ESBs incorporated complex logic, orchestration, and transformation not on a whim, but because the business processes that were being built needed them. The problem that ThoughtWorks call out is not that these processes were in some way hard or unnecessary, but rather that they were pushed into a centralized infrastructure, outside the confines of the applications that should own them. This centralized infrastructure and centralized business logic led to fragile software that was hard to evolve—everything you don't want in a modern and agile software project.

Event streaming systems approach this in a different way. They are built with dumb (but highly scalable) pipes and smart filters, but most notably, the filter is significantly more powerful and functional than anything before. The filter embedded in the microservice is equipped with all the power of a modern stream processing engine. There is no centralized logic. Everything is wholly decentralized, meaning *each bounded context owns it's own business logic, orchestration, transformation, etc.*

So with Kafka as the central nervous system, you can use the higher abstraction provided by streaming processing tools to map between the models created in different bounded contexts.

This enables you to leverage all the good stuff that comes with DDD but without the headache that came with ESBs—tightly coupled, centralized management of overall business processes.

How you choose to implement each microservice is entirely up to the team doing the work. A microservice can choose to use a simple technical interface like REST or JMS that only mediates communication. As an alternative, you can build real event streaming microservices that leverage the full power of stream processing under the hood to manipulate event streams of data and map them to your internal model.

In the above example, we have different microservices. Some use REST (for example, to communicate between microservices and the UI), and some use Kafka Streams or ksqlDB to join events from different sources together. Others use Kafka Connect to simply push events into a database where they can be manipulated further or used directly via event sourcing patterns. This technical implementation can be done for each microservice regardless of other microservices and their used technology.

How to build systems like this

Building decoupled event streaming microservices with REST, gRPC, or some other request-response protocol is rather obvious. But for many, building systems with events requires a larger conceptual leap. Events can be used to build systems in many different ways. They can be used as fire-and-forget messaging, but they can also be used as a means for collaboration. Ben Stopford's blog post Build Services on a Backbone of Events explains these patterns in greater detail.

You must also choose how to manage state. This can be done in a database with technologies like Kafka Connect or with managed in-service using the Kafka Streams API. More on building stateful event streaming microservices can be found in Ben Stopford's blog post Building a Microservices Ecosystem with Kafka Streams and ksqlDB.

For a more comprehensive view of how Connect, Kafka Streams, and microservices fit together, there is a great post by Yeva Byzek that summarizes the state of play. Finally, building a microservices ecosystem is only the first part of the puzzle. Once it's built, you need to instrument, control, and operate it. Find out how to do that here.

It's also worth mentioning Confluent's RBAC features, which allow role-based access control across the Confluent Platform. You can configure in detail what resources (like a Kafka topic, Schema Registry or connector) each domain team has access to.

Just pick the architecture which is the best fit for you. For example, you could either let each Kafka Connect cluster be operated by the responsible domain team or host Kafka Connect as part of the Kafka cluster and allow teams to deploy connectors to it.

I hope this wealth of tooling for building and running microservice-based systems using DDD techniques proves helpful to you.

Apache Kafka + Domain-driven design (DDD) = Decoupled event streaming microservices

While there are many ways to construct microservice architectures, the techniques described in domain-driven design (DDD) are undoubtedly some of the most powerful, particularly if you are building systems that have a complicated business domain (e.g., healthcare, finance, insurance, retail).

Many of the design principles found in DDD are directly applicable to event-driven systems, including several that go beyond the scope of this post, but I've highlighted the most common technical challenges: how to separate an application into bounded contexts, why these separate contexts are important, why domain models are needed, and how this relates to messaging, Apache Kafka, and the use of events.

With the rich set of tools available today, microservice architectures gain much from using an event streaming platform to decouple each microservice from the next. Implementations can be request driven, simple event-driven systems, to whole event streaming business applications. They can also be any mixture of these different patterns. DDD provides an essential technique for managing the interplay between each of the parts, be it a ubiquitous language, bounded contexts, schemas, and anti-corruption layers. As we have seen, events and messaging provide key ingredients for making these systems work well in practice.

Interested in more?

If you'd like to know more, you can download the Confluent Platform to get started with the leading distribution of Apache Kafka.

Kai Waehner works as technology evangelist at Confluent. Kai's main area of expertise lies within the fields of big data analytics, machine learning/deep learning, cloud/hybrid architectures, messaging, integration, microservices, streaming processing, Internet of Things, and blockchain. He is regular speaker at international conferences such as JavaOne, O'Reilly Software Architecture or ApacheCon, and he writes articles for professional journals.
