gravitee.io

Leverage API Management to best implement Event-Driven Architectures

How API Management can bridge the gaps and eliminate the pains associated with managing synchronous and event-driven APIs

Table of Contents

Introduction: **Modernization is imminent**

Consumer demands are changing. Whether it be expectations around no downtime, real-time, accurate data, or connected experiences, organizations now have a new benchmark for what it means to be a winner in today's digital economy. And this applies to both B2B and B2C providers.

These demands have ushered in a need for organizations to rely less on more traditional request and response architectures and APIs and begin to implement event-driven, real-time architectures built on event-driven APIs.

Services in today's API ecosystems are triggered by the occurrence of events. Examples of events include:

- A user submitting a payment
- A logistics company scanning a package on acceptance
- A water quality sensor raising an alert on some failed criteria

Similarly, as users interact with enterprise applications, they expect real-time notifications triggered by events. These include:

- Bank transaction alerts
- Shipment tracking updates
- Ridesharing ETA

While the shift towards EDA is one that organizations should embrace, it does come with its own sets of pains and challenges associated with managing, securing, and governing the synchronous and asynchronous APIs that will typically undergird event-driven systems. This whitepaper is meant to help teams figure out how they can eliminate those pains. We'll cover:

- A brief overview of APIs and EDA
- How API Management can be used to ease EDA-related modernization
- The need for API Security in an event-driven world

A brief overview of APIs and EDA

API stands for Application Programming Interface. An application can interact with another application—invoke actions, query for data, and so on—through an API. An API enables communication between applications while not exposing the internal logic or implementation of the applications. When two applications are developed by different developers, teams, or organizations, APIs are especially important. APIs are the only way to ensure that complex systems can interoperate.

Recent trends in the adoption of APIs

Today, the usage of APIs is primarily between remote systems. As organizations shift toward microservice architectures deployed to the cloud, these distributed systems comprise multiple components running on different machines, and they all need to talk to one another.

The massive amount of data generated by today's systems has also contributed to the uptick in API adoption. The big data revolution highlighted the value that can be derived from analyzing data and basing business decisions on that data. Extracting data, aggregating it with data from other sources, and generating insights from that data—all these processes require APIs.

And the market has responded to these trends. For example, the API Management market is expected to **grow by about 35% by 2025**. And, at the time of this writing, the **ProgrammableWeb API directory** listed nearly 25,000 public APIs. In addition to the already-existing private APIs and partner APIs.

EDA explained

Event-driven architecture (EDA) is an architectural paradigm that emphasizes indirect interaction between components. This approach is often very useful for use cases that require real-time data stream processing and notifications. Event-driven architectures satisfy these use cases due to two key characteristics:

- **Loosely-coupled interactions:** Componen ts can perform their tasks without heavy dependence on other components. This makes it easier to add new components or new functionality to an application, swap in or out certain components, and reuse components in different contexts.

- **Asynchronous communication:** When component A communicates asynchronously with component B, component A is not blocked from continuing its work while it waits for a response from component B.

- **The exchange of messages:** Components communicate by sending messages to a central message broker. Other components wait for the broker to notify them of specific messages, and then those components respond accordingly. With this approach, components are loosely coupled and communicate asynchronously.

BROKERS, PRODUCERS, AND CONSUMERS

The message broker is the arbitrator of these loosely-coupled asynchronous interactions. It is known by both senders and receivers. It organizes and processes messages, and it can provide mechanisms for persistent storage of messages or message send retries. Common message brokers include Apache Kafka and RabbitMQ.

Producers are components that produce (or "send" or "publish", which are all synonymous terms in EDA) messages (or "events") to the message broker. Consumers are components that consume or handle messages received by the message broker.

The traditional API versus the event-driven API

The traditional, remote procedure call (RPC) approach to the interaction between remote systems is vastly different from that of EDA. Because of these differences, the APIs used to i nteract with these architectures are also different. Let's consider these differences.

REQUEST-RESPONSE VERSUS PUB-SUB

Traditional APIs use the request-response paradigm. Every call to a traditional API has two parts: a request from the client and a response from the server. With this paradigm, the server can't arbitrarily send information to the client.

Event-driven APIs, on the other hand, use the pub-sub (publish-subscribe) paradigm. The client (consumer) subscribes to the broker to receive messages. The server (producer) sends messages to the broker, which delivers them to all the subscribed consumers.

SINGLE RESPONSE VERSUS A STREAM OF EVENTS

Traditional APIs have exactly one response per request. Even if the response is empty, the fact that the call returned is considered a response. Consumers of an event-driven API will receive a stream of events that are not tied to any specific call.

TIMEOUTS VERSUS ENDLESS WAITING

If the backend of a traditional API suddenly becomes unavailable or is otherwise prevented from sending a response back, then the caller is blocked. To address this possibility, a best practice for traditional APIs is to define a timeout. If no response (including an error response) is received within the timeout, then the caller is unblocked and can continue running, dealing with the non-response accordingly.

Event-driven APIs are not blocking. When a consumer subscribes for some events, and no event arrives, then the consumer can't tell if there is a problem on the other side or if there simply are no events to handle. To detect backend problems, best practices for event-driven APIs include health checks and keepalive signals.

TRADITIONAL APIS AND THE COST OF POLLING

Traditional APIs can use polling to mimic event-driven APIs. The client can periodically call the server to check if there are any new events, and the server will return the events since the last call. However, there are disadvantages to this approach:

- The client doesn't receive events in real time. The longer the polling period, the longer the delay between the occurrence of an event and its processing.
- If the polling period is very short, then the caller may call for events too often, when there aren't any new events available. This can lead to wasted resources, and the costs would multiply in the case of multiple consumers.

How to leverage API Management and API Security to help ease event-driven modernization initiatives

The shift towards event-driven architectures has many implications across business and technical teams, and individuals from CTOs, to Architects, to Developers now have a large set of new requirements on their plates. The impacts will be felt intensely when trying to figure out new ways to manage and secure API and Event landscapes as these kinds of architectures call for a shift towards new protocols, query technologies, and backend services.

Realistically, the vast majority of organizations will still have to maintain and support certain API ecosystems across the business built on more traditional protocols. As a result, organizations now find themselves asking:

"How can we manage, secure, govern, and productize all of these APIs at scale?"

In the sections below, we explore how organizations can leverage API Management and API Security solutions to manage, govern, and productize their synchronous and event-driven API ecosystems, securely.

Utilize API Management and API Gateways to manage, secure, and productize heterogenous API ecosystems

API Management and Gateway solutions have existed for a long time in order to ensure that APIs are available and accessed securely and reliably. External requesters of your system interact with the API gateway, which functions analogously to a front-of-house security guard. The only entity that interacts with the underlying services is the API gateway. The use of an API gateway can significantly reduce the overall complexity and potential security vulnerabilities of the system through application of methods such as data logging masking, traffic shaping, authentication, usage contracts, policy application, and more.

While this all sounds great and obviously worth implementing, the trouble comes when organizations who are trying to modernize and make the move to event-driven systems find themselves asking:

"We have all of these synchronous APIs that run critical applications and services, but now we're introducing event-driven and asynchronous APIs and services. We'll need these components to communicate with each other, but how can we do it securely and reliably at scale?"

Traditional API Management and Gateway solutions have focused heavily on the synchronous use cases. On the event side, event brokerage solutions are able to broker events and messages well, but they can't apply all of the management, governance, and security measures that an API Gateway can.

The way forward for API-first organizations who also want to implement event-driven architectures and systems is to either build or invest in API Management and Gateway solutions that can manage both synchronous and event-driven APIs. These solutions would leverage advanced protocol mediation so that teams could apply the many benefits of an API Gateway to traditional synchronous relationships as well as synchronous to asynchronous relationships and asynchronous to asynchronous relationships.

Since the world has already heard plenty on how to manage fully synchronous API ecosystems, let's explore examples of how API Management solutions can be used to manage synchronous to asynchronous systems and fully asynchronous systems.

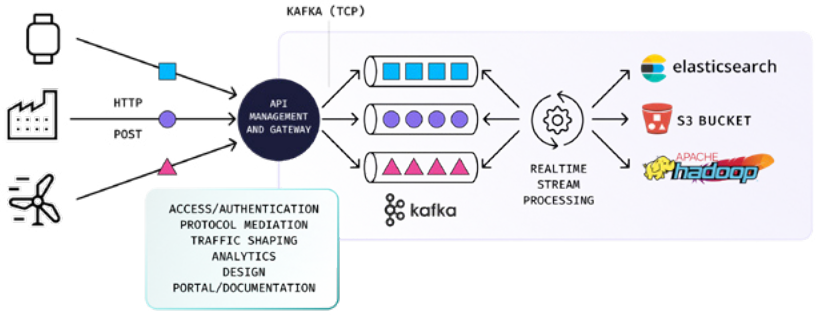
Synchronous to asynchronous management via an API Gateway

Yes, event-driven modernization is imminent. That said, "legacy" components of systems are often critical, and sometimes still most fit for a specific use case, and can't be "ripped out" in the blink of an eye just for the sake of modernization (just think about it: how many organizations tried to blow up their monolith too quickly and ended up blowing up everything else as well… including their uptime and availability). Organizations may need to keep the synchronous components of their systems up and running while they implement and integrate event-driven components.

Let's take an example of an organization who has client-side REST-based applications and are implementing Kafka as pub-sub at the backend for real-time messaging and stream processing. Here are the multiple use cases that their modern Gateway and API Management solution could enable:

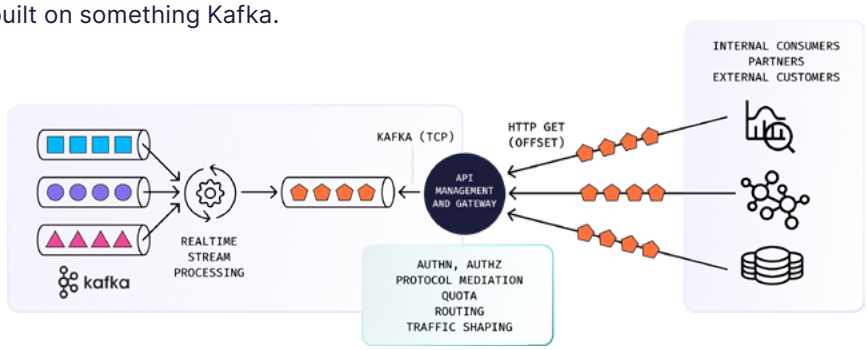
PRODUCTION AND DATA INGESTION:

Produce and push data from REST/HTTP-based APIs and applications (synchronous) through a Gateway so that it can be authenticated, traffic-shaped, measured, and ultimately ingested by an asynchronous, event-driven backend built on something Kafka.



EVENT CONSUMPTION VIA HTTP POLLING

Produce and push data from REST/HTTP-based APIs and applications (synchronous) through a Gateway so that it can be authenticated, traffic-shaped, measured, and ultimately ingested by an asynchronous, event-driven backend built on something Kafka.

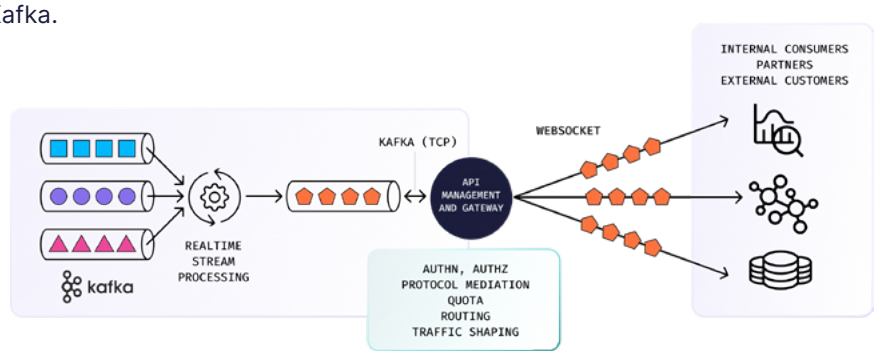


Asynchronous to asynchronous management via an API Gateway

Perhaps the organization from the first example has now cleaned up all of their tech debt and they're ready to implement "event-driven everything everywhere." This organization likely has an event brokerage solution, but, if they're API-first and understandably don't want to give up the additional benefits of that Gateway (traffic shaping, authentication, a wide variety of policy application, etc.), they'll need to set up a Gateway that can moderate between fully asynchronous components. We'll show three examples of what this could look like:

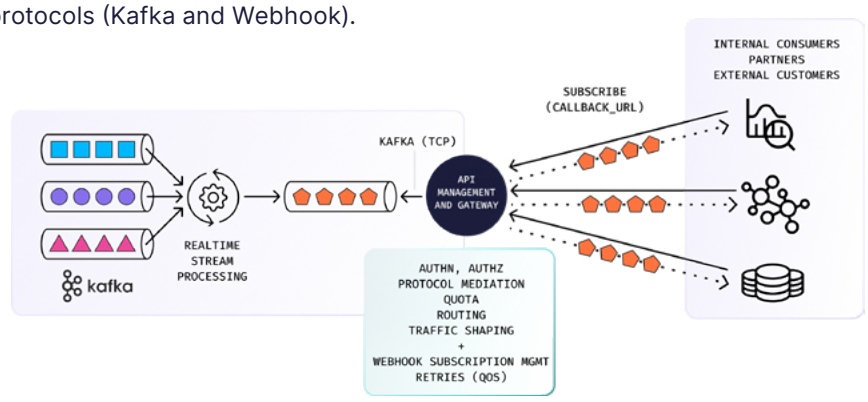
EVENT CONSUMPTION VIA STREAMING WITH KAFKA AND WEBSOCKET:

The API Gateway sits between Kafka and Websocket-driven consumer services for true asynchronous to asynchronous communication. The data that's passed between the two is authenticated, quota'd, routed, and traffic shaped–all without the need for constant HTTP polling of your Kafka backend. built on something Kafka.



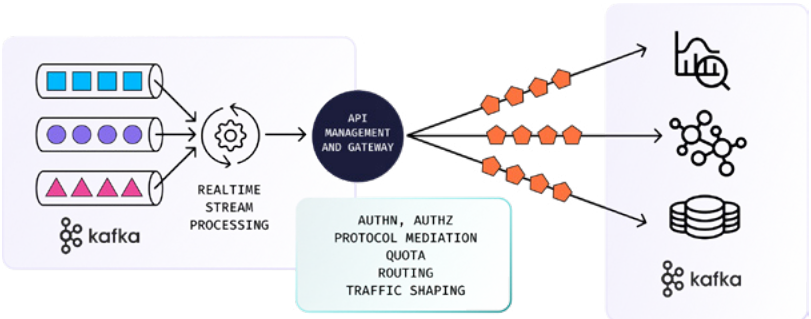
EVENT CONSUMPTION VIA WEBHOOK

In this example, events are pushed through a Gateway for authentication, quotas, routing, traffic shaping, and webhook subscription management, and retries, all triggered by the asynchronous communication via Webhook from consumers.. This is possible due to the Gateway being able to mediate between different protocols (Kafka and Webhook).



"ALL KAFKA" ASYNC TO ASYNC EVENT CONSUMPTION

Whereas the prior examples rely on protocol mediation so that different protocols and services can "shake hands," this example illustrates a Gateway sitting between both a Kafka server and a Kafka client. Because it's Kafka to Kafka, the Gateway might not be as valuable for altering traffic, but this would allow you to garner other benefits such as analytics and the ability to expose APIs built on this system to consumers via something like a Developer Portal (see next section)



More than just API Management: how to properly secure synchronous and event-driven APIs ecosystems

Security is paramount, and APIs present a serious risk. As organizations expose more and more critical business services via internal or external APIs, the attack vectors naturally increase. The subsequent result is that malicious actors and their attempts to attack your system will focus on your API layer. To avoid this, organizations should consider two main categories of API Security beyond the base-layer of security measures offered by your API Management and API Gateway solutions:

- **Access control:** securing your APIs by means of controlling which applications and users can access and ultimately consume your APIs
- **API threat protection:** the practice of automating how attacks against AIs are detected and blocked before security issues are caused.

In the following two sections, we will explore each in more detail.

Implement proper access control for synchronous and event-driven APIs

Access control has been around for a while, and many organizations are already practicing it. Common methods of access control include authentication, authorization, and identity propagation. This would look like organizations implementing Web Tokens, Multi-factor authentication, OAuth 2.0, SAML 2.0, etc, and these are typically accomplished with Identity and Access Management and API Gateway solutions.

Stop API attacks before they happen with API threat protection

While some aspects of API threat protection have existed for quite some time, there are some newer practices that have begun to surface due to APIs becoming more crucial security threats. These include, but are not limited to:

- **API Discoverability and auditing:** discover andit APIs that exist in your ecosystem and flag those that are not secured adequately given certain standards and policies

- **API Monitoring and alerting:** monitor how APIs are being consumed in real-time and configure notifications and alerts when malicious behavior, bots, etc. are detected for quick isolation and remediation.
- **API Firewalls and runtime policy enforcement:** build a specific firewall for securing APIs so that security policies can be applied to APIs at runtime.

Securing everything

Oftentimes, API security tooling will plug into an API Gateway in order to apply many of their API security solutions. While this is often effective, it can present challenges for teams who are either using multiple Gateways to manage and govern synchronous and event-driven APIs and/or teams who might only have robust Gateway solutions for their synchronous APIs and nothing in place for their event-driven APIs. Because of this, we recommend that teams look for single API Management and/or API Security solutions that can support both synchronous and event-driven APIs.

Sometimes missing from an API Management strategy: use a developer portal to drive consumption of event-driven and synchronous APIs

APIs are most useful when they're actually used. **Full stop.**

This means that, beyond just designing, implementing, and securing APIs, you have to make them consumable. More often than not, APIs are consumed by developers, both internal to your business and external via customers and/or partners. To harvest the most value from your APIs, developers need the ability to discover APIs and learn how to use them effectively. Investing in a Developer Portal facilitates discovery, and it also provides a central location for API documentation, examples, and tooling support.

Developer portals don't just enhance visibility externally. They can also improve internal visibility and, depending on the use case, turn complex API ecosystems into profitable revenue generation machines. Some API Management solutions will have Developer Portals "baked-in" whereas some Portals and API Management solutions will simply be externally integrated. The "baked-in" portals often provide extra benefits around analytics into

external or internal API consumption so that you can always keep a "single pane of glass" view into how your APIs are being used in the real world. Developer portals also often provide options for defining usage plans, contracts, and creating custom monetization strategies so that API consumption can begin to directly drive revenue generation.

As great as Developer Portals are, they aren't all created equal. This can often present problems for the organization interested in implementing event-driven architectures and asynchronous APIs, as many portals offer specific support for either synchronous (most common) or event-driven/asynchronous APIs (less common). This prevents forward-thinking organizations from "getting the most out of" their APIs, and, for this reason, we recommend investing the necessary resources on either a commercial or DIY portal solution that can support both synchronous and event-driven APIs.

Revenue generation example:

Imagine you are a B2B delivery service that offers accurate, real-time delivery updates as a service to businesses along various points of a supply chain.

You could implement event-driven APIs that help you gather information in real-time around traffic patterns, delivery fulfillment, remaining stops, and speed of the vehicle all asynchronously to deliver real-time delivery estimates to customers.

With a proper monetization strategy in place, this API could exist in accordance with a contract that says **"the consumer of this API will pay 10 cents per every 100 times this API is called."**

Conclusion

Today's systems handle more data, coming from more diverse data sources like sensors and IoT devices in the form of events. Event-based data and real-time processing requirements have led to a massive uptick in adopting event-driven architecture for building systems.

The shift towards event-driven and real-time architectures is imminent. For the majority of organizations, this means incorporating event-driven services and APIs into their technology strategy in addition to existing synchronous APIs and API ecosystems. Ultimately, this results in more and more data that's necessary to process.

Various data technologies offer solutions for managing, accessing, and processing all of this event-based data, but. However, this can bring more problems than solutions. Bloated toolsets can result in sky-high IT budgets, tool-driven data silos, and more Engineering toil related to trying to wrangle and integrate different systems. To try and avoid this problem, organizations will often try to implement DIY solutions in-house. While this is a fine strategy for some organizations, many will find that the initial investment in building the solution, maintaining it over time, and improving it as new technologies and protocols arise ends up costing more money, more time, and more toil than even bloated commercial toolsets.

Gravite offers an alternative.

Gravitee customers are able to manage, secure, govern, and productize their synchronous and asynchronous API ecosystems–with one platform. With solutions that range from no-code/low-code API design, to an API Gateway that can proxy and mediate between synchronous and asynchronous protocols, to Identity and Access Management, to API Security, Monitoring, to a Developer Portal that also supports synchronous and event-driven APIs, Gravitee customers can be sure that they'll be ready for whatever the API and event universe throws (or streams) at them.

When you're ready to start optimizing your shift to event-driven architectures, make sure to **book a demo** with Gravitee to learn what all the hype is about.

How to **Contact Us**

gravitee.io**/contact-us**

If you're interested, and want to reach out, you can contact us here

gravitee.io**/demo**

If you'd like to skip (some of) the Sales pitch and see a demo, you can book one of those here

community.gravitee.io

If you want to give OSS a go, check out our community forum, where you can find links to our github repo and connect with the folks who have driven over 350,000 Docker pulls / month


gravitee.io