Event-driven APIs in Microservice Architectures

Version Q4-2020

Author

- Dakshitha Ratnayake, Associate Director - Technology Evangelism dakshitha@wso2.com @techieducky

> *This paper will focus on event-driven APIs and their management. Event-driven as a concept has been around for a long time but event-driven APIs are a more recent topic, which entails using event-driven architecture to support scalable, real-time (or near-real-time), push-based communication in APIs published to third parties.*

1.0 Introduction

As architectures and technologies evolved to enable 'digital transformation', monolithic applications got broken down into coarse-grained services and now into more fine-grained loosely-coupled microservices that can be implemented using a variety of technologies. Docker and Kubernetes have simplified the containerized deployment and scaling of microservices across different environments. To enable consumption for customers, partners, and other third parties, these microservices have been exposed via managed RESTful APIs and they communicate directly and synchronously with each other over HTTP. Developers have designed resource-based APIs using CRUD actions. And of course, API management platforms and rich tooling have made the consumption experience all the more easier and better. The grand majority of APIs that make up the web today are synchronous APIs. However, synchronous, request/reply interactions happen one at a time, in a pre-arranged sequence, and each interaction blocks the progress of the process until its completion. This means that the response time to the user is a cumulative sum of response times of the microservices. Also, if a client wishes to know about an update, it must continuously invoke the API to check for data modifications. This pattern is known as polling and has been a common solution for clients that need to become aware of new data or notified of backend events. And most of these 'polling' calls are wasted because the data hasn't changed and the resources (CPU, network, etc.) were used unnecessarily. Thus, the term 'polling madness' was coined. Furthermore, rate limiting (mostly enforced by API managers) prevents clients from polling often enough to support the solution's needs for event updates.

Making APIs event-driven or asynchronous can systematically address these issues and really deliver on the (near) real-time experience, and adopting an event-driven architecture at a system level can truly provide sizable benefits and bring comfort to end-users and efficiency to the web. This paper will discuss how.

1.1 Event-driven Systems

What is event-driven? An event-driven architecture (EDA) centers around the concept of events. An event can be defined as a change in state and can closely represent what happens in the real world. Online purchases, human actions, command and control instructions, sensor readings, news feeds, stock ticks, interest rate changes, process alerts, workflow notifications, fault detection, and fraud detection are all real-world incidents that can be captured as events in an event-driven system. Being event-driven is about taking advantage of events as they happen; in other words, it's about sensing, capturing, and responding to these events in (near) real-time to deliver a higher quality of service, offer a better customer experience, and make more informed business decisions.

Real-time (or near real-time) experiences are built with event-driven architectures.

It's about the power of being able to do something in that instant as it happens. What is real-time varies for different industries (usually milliseconds or less) with their own metrics. In capital markets, 1 millisecond is worth $8m. This means that apps need to be highly reactive to change. One of the biggest advantages of EDA is that it optimizes the time it takes between the occurrence of an event and the reaction by the company to that event. So, instead of polling, consumers can register their interest (subscribe) and react to events in real-time. With the real-time transmission of events (pushing instead of polling) and services being executed in parallel, everything interacts with each other efficiently. Better response times result in better user engagement and satisfaction and, therefore, better business.

EDAs are loosely-coupled and agile.

In event-driven architectures, applications and the underlying services are loosely coupled. Consequently, event producers simply transmit events (and do not care if anyone is interested). They fire, usually to a broker, and forget. Consuming apps and microservices do not have to know how many different applications they are consuming data from; they are only concerned about the data. So, the consuming applications are registered with the broker to receive the data updates they want. Decoupling, as we all know, offers many benefits. Because things are dealt with independently, consumers can be scaled independently of the sender. This means allowing millions of devices to consume events. Furthermore, bottleneck issues can be more easily tracked to a single component due to the decoupling.

It also makes systems flexible and agile, such that new systems can be easily plugged into the architecture without making any changes to the source systems or impacting other systems, thus future-proofing the architecture. It allows developers to spend less time worrying about the systems and more about the business logic, allowing them to build products faster. An event-driven microservice architecture means faster time-to-market for new and modified business capabilities.

Becoming event-driven creates efficient systems.

EDAs not only improve latency (so that users don't have to refresh their UIs to get the latest updates) but also support reliable delivery. With most brokers, if a service fails, the message can be stored by the event broker and can be delivered when the service comes back online—reliable delivery is a basis for eventual consistency and no data loss. A stateful intermediary with the right

capabilities also allows for additional fault tolerance, speed mismatches, and shock absorption during peak volumes.

With these gains due to EDA, an API vendor servicing clients with events can not only make them happier but also reduce their cloud bill on CPU and network drastically.

2.0 Architecture

2.1 A Synchronous API-first Microservice Architecture

There are mainly three components in the architecture:

- The consumer applications: Anything that can consume an API.
- The API exposure layer: Ideally an API gateway or a micro gateway but sometimes also a load balancer, a layer 7 firewall, a CDN, or any other layer 7 application on top of core service(s).
- The microservices: A microservice (a core service or integration service) basically encapsulates a logic.

Figure 1: A Synchronous API-driven Microservice Architecture

A synchronous API call involves a single thread to process a request and its response. As shown in Figure 1, for each request (REST, GraphQL, gRPC), a response follows. Synchronous communication is ideal for many scenarios especially if you need an instant response; however, in other cases, especially when the processing required for the response happens at a different time, ordinary synchronous messaging becomes tricky. As a solution, developers have resorted to using polling techniques to get updates. But the data keeps changing constantly. For example, a retail application wanting to know price changes will need to poll the API continuously, thus overloading the backend system. As in most cases, if an API management system is present, it will throttle out these requests.

Figure 2: Pushing Events via Async APIs in a Retail Application

However, if the retail application is implemented in an event-driven way, the price changes are pushed to the client instead. First of all, any application that is interested in the price change event will subscribe to the event. When a price change occurs, that data will get distributed out to the subscribed apps. Such an app will now no longer need to continuously query for the price because the latest price is in its system; the app can keep this data in its cache or keep it persisted. For such scenarios, it makes more sense to receive push events via ‘async’ APIs than polling via ‘sync’ APIs as shown in Figure 2.

2.2 An Event-driven API-first Microservice Architecture

Application languages and backend architectures have relied on events before REST was created.

Back then, financial industries had events from exchanges pushed to hedge funds to allow them to stay in front of the market using IP-based tweaks and proprietary middleware. Since then, message buses have evolved to open standards adopted outside finance. Although EDA does not explicitly require the use of middleware, using such an intermediary between event producers and consumers helps to implement corresponding patterns and deliver more manageable and scalable solutions. We refer to this middleware as the event broker. A few notable examples of brokers are RabbitMQ, Apache Kafka, and Apache ActiveMQ, among many others in the market today. Moreover, a scalable microservices architecture (MSA) is the optimal architecture for complex event-driven backend services. These event-driven microservices can act as event subscribers or publishers in order to process events, handle errors, and persist event-driven states.

In contrast to REST-fashioned APIs (which are usually implemented in polling scenarios), push or streaming APIs are event-driven. An event-driven API requires two capabilities—a mechanism to allow a consumer to subscribe (this can be user-controlled or programmatic) and the means to deliver events to consumers that are subscribed. The event-enabled APIs and/or services can connect to the broker and clients can subscribe to a channel of interest. Eventually, when an event takes place, it triggers a data flow to a client that's waiting for the inbound data in order to process it in real-time. Additionally, when it comes to two-way communication, the client application should be able to publish events to the backend via the event-driven API as well.

Figure 3: An Asynchronous API-driven Microservice Architecture

To be more specific, an event broker, as shown in Figure 3, can be used to:

- Behave as a backbone or the event distribution layer for microservices that publish and process events.
- Liberate data from existing systems/brownfield. This can be done via an Enterprise Service Bus or Change Data Capture (CDC) tools. When something changes on the mainframe, they can publish events to the broker.
- Be fed by IoT devices (alarms, sensors, devices), connected systems (PoS terminals), etc.
- Interact with API gateways to expose data as events to the outside. These gateways (or micro gateways) can take data from a request/response and transform it into an event or can even event-enable existing REST APIs.

3.0 Event-driven APIs

Receiving notifications about someone liking a picture or reacting to a story on Instagram, a new Whatsapp message, a stock ticker, or social stream displaying the latest updates are all made possible through various modes of asynchronous communication. So how exactly should we perform asynchronous event-driven communication in the world of APIs where synchronous communication is predominant and most firewalls block non-HTTP traffic? There really is no one-size-fits-all solution.

The technology landscape for asynchronous APIs is rapidly booming. The older protocols such as WS-Eventing and XMPP (a polling protocol that opens an HTTP thread and never closes it) are now making way for newer technologies (even though Whatsapp is still based on XMPP at the time of this writing). Since 2011, with the advent of social networks and to support reactive UI, the web has standardized protocols for low-volume bi-directional/peer-to-peer traffic (Websockets) and server to client push over HTTP (Server-Sent Events). For battery constrained IoT devices, one would generally prefer using MQTT. Webhooks have become popular to handle low-volume events. Even though Kafka, which is based on TCP, is great for dealing with asynchronous communication between internal microservices, it is not optimized to be exposed to provide API consumers with easy access to real-time data and it will create issues with firewalls.

The payloads are still mostly JSON but Avro in Kafka and protocol buffers in gRPC are becoming more prevalent. In synchronous REST APIs, queries can be appended to the URL string that addresses an API's endpoint (a GET request), or package the same query as a JSON payload instead (a POST request). Push/streaming approaches can be very different to each other. For example, Websockets allow for bi-directional streams while Webhooks and Server-Side Events are unidirectional. XMPP and SMPP differ substantially, such that XMPP is primarily based on XML payloads while SMPP relies on binary data.

A message broker is usually required in the backend architecture to capture and filter events and manage event subscriptions. Moreover, languages traditionally used to create application layers linked to databases have also gone through their event revolution with reactive extensions such as RxJava widely adopted by the Java community. And then there's AsyncAPI, which is now widely adopted to document event-driven APIs.

Let's dig deeper into a few commonly-used and emerging standards in the event-driven API space.

3.1 Defining Event-driven APIs with AsyncAPI

One of the reasons why API management platforms really took off is their ability to streamline the development and drive adoption for their APIs via developer portals and API definition standards like OpenAPI. Once the provider defines the API interface as per the specification, consumers can pass the interface definition into a code generation framework (such as Swagger) to generate documentation and code for execution on a runtime platform. This minimized the amount of manual effort and communication required from an end-to-end perspective.

A similar specification to standardize event-driven APIs has come up fairly recently: AsyncAPI. AsyncAPI is designed along the same elements of OpenAPI and shares many common constructs to simplify the adoption, but it also comes with additional features to accommodate eventing. It supports a wide variety of messaging protocols and transports (such as AMQP, MQTT, WebSockets, Kafka, JMS, STOMP, HTTP, etc.) and event schema formats. Therefore, the API definition will contain the event payload definition, channel name, application/transport headers, protocols, and other eventing semantics to connect, publish, and subscribe to the API.

3.2 Asynchronous Messaging Protocols for APIs

Webhooks

Also known as “reverse APIs”, webhook APIs completely detach the HTTP request and response from each other.

Figure 4: Event-driven APIs with Webhooks

With webhooks:

1. An API developer can subscribe to a REST API with an API Key and provide a callback URL (an HTTP endpoint, also known as the webhook) via the API provider’s developer portal.
2. During runtime, the developer’s application will call the API with the API Key.
3. For such a request, a 2xx response will be sent as an acknowledgment.
4. The API provider will "stream" back to the callback endpoint through an HTTP POST request, when the expected process triggers events that should be reported back to the consumer.

Webhooks are rather easy to implement. However, compared to other mechanisms, webhooks are better for pushing notifications to one or a small number of endpoints. Webhooks are unsuitable to push events directly to client applications, (i.e., mobile/browser apps or other private apps hosted inside a firewall), because each client has to host an HTTP endpoint and be in possession of a publicly addressable domain name, and securing this network (typically using basic authentication or mutual SSL), would involve an unmaintainable administration overhead. The clients are almost always servers themselves.

The OpenAPI 3.0.0 specification supports webhooks through a callback element that can be used to define a webhook. However, in general, there are no formal standards around webhooks.

WebSockets

The WebSocket protocol allows for constant, bi-directional communication between the server and client, which means both parties can communicate and exchange data as and when needed. WebSockets are point-to-point connections modeled on TCP sockets and lack any native pub-sub support.

Figure 5: Event-driven APIs with WebSockets

WebSockets don’t use a request/response strategy where a connection is opened in the course of making the request and then closed after it's initially fulfilled. In the case of WebSockets, the connection remains open until closed explicitly. A WebSocket connection is established by making an HTTP call and then asking for an upgrade on that connection. After that, the communication

takes place over a single TCP connection using the WebSocket protocol. In other words, instead of sending a request to `http://<target>`, the request is sent to `ws://<target>`.

Using a single TCP connection reduces resource usage by transferring only essential information (the HTTP header overhead is reduced) and thereby optimizes performance. To use WebSockets, the browser must be compatible. However, this limitation is insignificant because a majority of the browsers today already support the WebSocket protocol. The WebSocket interface can be defined with the AsyncAPI specification. WebSockets have been combined with MQTT, AMQP, and proprietary protocol implementations to provide pub-sub communications.

Server-Sent Events

With Server-Sent Events (SSE), an open, lightweight, subscribe-only protocol, a browser application can subscribe to a stream of events or message updates generated by a server. First of all, the client creates a new EventSource object, and passes the URL of an endpoint to the server over a regular HTTP request. The client then waits for a response with a stream of event messages. The server leaves the HTTP response open until it has no more events to send. The server will terminate a stale connection (if the connection has been open long enough) or wait for the client to explicitly close the initial request.

Figure 6: Event-driven APIs with Server-Sent Events

SSE is appropriate if an efficient (due to better latency compared with other HTTP-based streaming methods) unidirectional communication protocol that doesn't add unnecessary server load (as with HTTP long polling) is needed. SSE also comes with a predefined standard for handling errors and is widely used in the industry today to push news alerts, live sports scores, stock price updates, among many other server-generated updates. The only major limitation of SSE is the inability to pass information to a server from a client, thus making it a highly popular choice to implement server-to-browser asynchronous communication.

GraphQL

There are two common ways to perform asynchronous communications from a GraphQL server: subscriptions and live queries. Facebook developed GraphQL subscriptions internally in 2015 and used it to power global-scale features like live comments and streaming reactions. Live queries came into the picture much later and are not as widely used as subscriptions, yet.

Subscriptions observe events; live queries observe data. Both are request/stream operations where the server responds to a client request with a stream of GraphQL responses, in the format designated by the client's request document. Subscriptions are GraphQL operations, defined in the specification. Live queries are not formally defined in the specification, and are generally modeled by adding a special directive to a query operation.

Both methods are technology agnostic. In the spec, subscriptions only specify algorithms for the creation of a stream, the content of each payload on that stream, and the closing of that stream. There are intentionally no specifications for serialization formats, the transport mechanism, message acknowledgment, buffering, resend requests, or any other quality of service (QoS) details. These are left to be chosen by the implementation service. So, those programming GraphQL subscriptions or live queries can use a messaging technology that best suits their needs. Both methods can use a variety of protocol (e.g., MQTT, AMQP, RSocket, Redis, Socket.io, and other formats) and transport (e.g., WebSocket, TCP, HTTP long-polling, SSE, etc.) combinations.

Subscriptions require a message broker. Live queries that outgrow polling will need reactive data sources (such as a database that allows to tail a query) and an accompanying programming model (such as Rx). In both cases, the server needs to store each operation execution request, subscribe to underlying source streams, and maintain an index of long-lived connections back to the client via a real-time gateway.

GraphQL Subscriptions

GraphQL subscriptions is a streaming mechanism built into GraphQL and is designed in a way that both synchronous HTTP communication and asynchronous event-driven interactions are available from a single API experience, i.e., a single API defined in GraphQL can support a mix of request-response queries and mutations (commands) as well as asynchronous event notifications. Subscriptions provide the ability to emit messages (not bi-directional) asynchronously out of the GraphQL API from within query or mutation execution logic.

Figure 7: Event-driven APIs with GraphQL Subscriptions

Subscriptions are client-driven, meaning that consuming applications define what data must be included in the event. Clients can indicate a subscription operation by using the subscription keyword, so that whenever the event defined occurs, it executes the defined selection and sends the result.

Implementing a GraphQL subscription requires the use of a message broker. The Apollo GraphQL Server uses the WebSocket protocol and comes with a message broker installed by default. Sangria (another implementation of GraphQL) uses Server-Sent Events. While WebSockets is the popular choice, other protocols such as AMQP or XMPP can be used too.

GraphQL Live Queries

Figure 8: Event-driven APIs with GraphQL Live Queries

A GraphQL query can be transformed into a live query by including a directive, such as '@live'.

Clients can indicate a subscription operation by using the subscription keyword, so that whenever the event defined occurs, it executes the defined selection and sends the result. In short, the client is telling the server to evaluate the defined selection and inform whenever the 'selection' would yield a different response. Just like subscriptions, a live query is a read-only push stream (not bi-directional). It requires the implementation of a reactive data layer (e.g., RxJS) in the GraphQL server (e.g., to tail a query). The reactive data layer listens to database level changes, which can be streamed back to the consumer.

Hasura supports both Subscriptions and Live Queries, and the Apollo client can approximate the behavior of a Live Query with its built-in polling feature.

gRPC

Despite being widely used for synchronous communication between internal microservices (microservices within an organization), gRPC, unbeknownst to many, supports bi-directional streaming because of its ties to HTTP/2. In other words, it allows defining a service operation that either accepts an incoming stream or emits an outgoing stream, or both. This is popular for microservices that are built with gRPC and need to be able to communicate back and forth.

gRPC is a standard that takes advantage of HTTP/2 and protocol buffers (instead of JSON) to ensure increased performance and maximum interoperability. gRPC uses protocol buffers (or protobufs) as serialization format over HTTP/2 and also to create the interface definition of the gRPC service. In other words, the .proto file acts as the service definition (instead of a Swagger file) and can be used to generate client stubs and server-side skeletons. Compared to REST APIs, gRPC APIs are faster. However, there is a considerable learning curve and it isn't always as intuitive and easy to learn as REST.

Figure 9 - Event-driven APIs with gRPC

Even though external-facing communication based on gRPC is rare, there are methods to do so. For example, a gRPC web client, which is a special client for browsers, can be used for client-server interactions. However, this is complicated. Therefore, as a general practice, we can use gRPC for synchronous and asynchronous communications between internal microservices and in B2B environments.

3.3 Event-Enabling API Management

Because dynamic access to data and the federation of data across ecosystems can create groundbreaking capabilities, existing API management platforms have also begun to support several asynchronous API protocols (the most common ones being webhooks and WebSockets at the time of this writing), thereby facilitating an end-to-end event-driven architecture. By leveraging both OpenAPI and AsyncAPI specifications to document and discover these event-driven APIs, enterprises and SaaS companies have been able to produce streamlined real-time applications, API

products, and services for their customers, B2B partner companies, and internal business units.

An event-driven backbone will manage the overall real-time data flow securely and at scale, while the asynchronous APIs can be managed for external and internal consumption with a traditional API management solution that comes with inherent or extended capabilities to support event-driven semantics. The API manager will handle security, monitoring, auditing, throttling, discovery, and tooling capabilities for the event-driven APIs and also provide ways to enable commercial models around the APIs

Figure 10: Event-driven API Management

While each eventing protocol will have nuanced needs to event-enable the management of APIs, they will have a set of common core needs for user on-boarding, authentication, logging, analytics, etc. Typically, at a high level, the flow would be as follows:

- API developers will point to an existing service endpoint (e.g., WebSockets) or a topic URL via an API publisher portal. They can define one or many API paths by providing HTTP, HTTP/2, or TCP access. The publisher portal is effectively an integrated development environment that lets developers create the API interface definitions using OpenAPI (for webhooks) and AsyncAPI (for WebSockets) and also generate documentation that allow consumers to discover and write applications that utilize the service. The portal is also used to manage the lifecycle of APIs
- Developers interested in the event-driven API will discover and subscribe to the API through the developer portal. They will have to provide a callback URL in the case of webhooks. This request is managed within the developer portal, which generates the required API keys.
- Under the hood, via an API proxy in the gateway or otherwise, the message broker will be configured with the correct access control lists, internal queues, subscriptions, and various other settings to ensure the secure delivery of the selected data product streams via the broker and API proxy to the developer's webhook or to the client application directly. During runtime, the API gateway will handle the runtime proxying of service requests and responses and perform authentication, throttling, logging, mediation, and transformation by communicating with other key components such as the security, throttling, and analytics components of the API manager.
- The security components will provide one or multiple ways for API consumers to authenticate and get access to the event-driven APIs.
- The API analytics tools will continue to understand how APIs are being consumed and how applications are putting API resources to use, and identify patterns related to API consumption across ecosystems to help create commercial models. They will help visualize all events within an enterprise regardless of the underlying infrastructure.
- The API monetization tools (in conjunction with the analytics capabilities) allow businesses to generate revenue out of their APIs by implementing different API monetization models such as pay-as-you-go, subscription-based usage, etc. They either come with the ability to bill users

directly or integrate with external billing systems.

To sum up, even though most organizations have basic event processing infrastructure, many don't have the capabilities to design, develop, test, and manage event-centric APIs. Combining traditional API management capabilities—particularly governance, access control, monitoring and analytics—with an event-driven architecture provides a tremendous value addition in terms of expanding the business reach and adoption by providing managed access.

4.0 Conclusion

Making APIs asynchronous reverses the paradigm by letting the backend push streams of events to the API client. Augmenting a REST API with event-driven capabilities allows for better and (near) real-time experiences, and drastically increases efficiency in CPU and network use for even the most demanding traffic.

Event-enabling APIs comes with many challenges. There are multiple complexities to deal with, starting from what frameworks and networking protocols to choose to building the reliability of delivery and ensuring the scalability of the web-scale solution. Unlike the REST-fashioned APIs, which follow the same basic patterns, there are multiple approaches to push/streaming that feature some significant differences from one another. The asynchronous protocols commonly used today solve the problem of (near) real-time communication, but they solve different aspects of this problem in different ways, thereby making some protocols serve different purposes better than others. Managing these APIs is another crucial aspect because everyone managing physical assets (from cars to windmills) and other event-driven systems is trying to manage and monetize the data streams generated by them. And that requires API management tools and API definition standards for push/real-time APIs.

Event-driven APIs have been the missing piece to realizing an end-to-end event-driven architecture. With these in place, the entire chain – from the data plane, API, network, up to the front-ends – can be event-driven. This allows organizations, their customers, and partners to truly reap the benefits of an event-driven architecture including the simplicity, agility, and efficiency it brings to all components individually and collectively. Having said that, event-driven client applications still need traditional request/response synchronous communication for interactive, non-event-driven interactions between the client and the backend. Combining event-driven APIs (underpinned by an event-driven architecture) with traditional request/response design brings the best of both worlds to build reactive client applications that delight customers and work just right.

5.0 References

[1] Emmelyn Wang - An API Strategist Explores Event-Driven APIs

[2] Luis Weir - Event driven API Strategies: from WebHooks to GraphQL Subscriptions

[3] Chris Wood - What is a Webhooks Push-Styled API and How Does It Work?

[4] Bob Reselman - How to Build a Streaming API Using GraphQL Subscriptions

[5] Ably.io - Server-Sent Events (SSE): A Conceptual Deep Dive

[6] Robert Zhu - GraphQL Subscriptions vs. Live Queries

[7] Kasun Indrasiri - Build Real-World Microservices with gRPC

[8] Kristopher Sandoval - 7 Protocols Good For Documenting With AsyncAPI

[9] Ricardo Gomez-Ulmke and Florian Geiger - Bringing Asynchronous Messaging to a RESTful world with Solace and Apigee

[10] Meshvi Patel - Why RESTful APIs Can't Compete with the Event-Driven Approach to Microservices

[11] Arshardh Ifthikar - Introducing WebSocket APIs with WSO2 API Manager

[12] WSO2 API Manager Documentation - Create and Publish a WebSocket API