The InfoQ eMag / Issue #86 / October 2020

The Real-Time APIs
Design, Operation, and Observation

InfoQ

The Challenges of Building a Reliable Real-Time Event-Driven Ecosystem

Real-Time APIs: Mike Amundsen on Designing for Speed and Observability

Real-Time APIs in the Context of Apache Kafka

The Real-Time APIs
Design, Operation, and Observation

IN THIS ISSUE

PRODUCTION EDITOR Ana Ciobotaru / COPY EDITORS Lawrence Nyveen & Susan Conant / DESIGN Dragos Balasoiu
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

Matthew O'Riordan

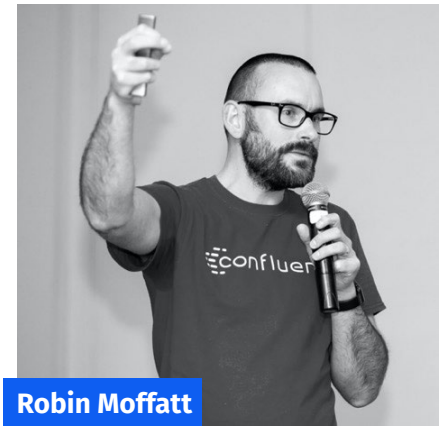
is the technical co-founder of Ably, a global, cloud-based real-time messaging platform that provides APIs used by thousands of developers and businesses. Matthew has been a programmer for over 20 years and first started working on commercial internet projects in the mid-90s when Internet Explorer 3 and Netscape were still battling it out. While he enjoys coding, the challenges he faces as an entrepreneur starting and scaling businesses is what drives him. Matthew has previously started and successfully exited from two previous tech businesses.

Karthik Krishnaswamy

is Director, Product Marketing at F5 Networks, drives marketing initiatives for NGINX API Management and F5 Cloud Services. He is an experienced product marketer with a proven track record of developing and promoting IT solutions. Prior to F5, Karthik held similar positions at Fluke Networks, Cisco Systems and Nimble Storage, a Hewlett Packard Enterprise company.

Guillaume Corre

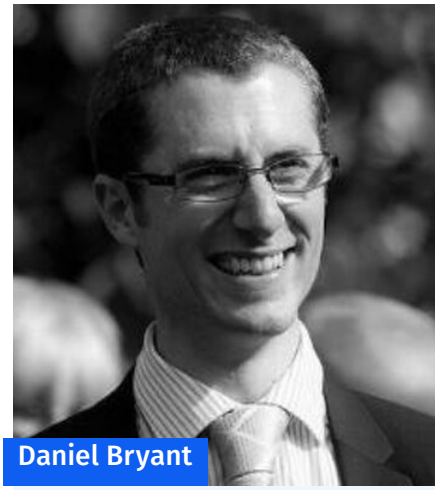
works as a software engineer, consultant and Tech Lead at Gatling Corp, based in Paris at Station F, world's biggest startup campus. Swiss army knife by nature, he enjoys simple things such as data viz, optimization and crashing production environments using Gatling, the best developer tool to load test your application, preventing applications and websites from becoming victims of their own success and help them face critical situations, like go-lives or Black Fridays



Robin Moffatt

is a Senior Developer Advocate at Confluent, the company founded by the original creators of Apache Kafka, as well as an Oracle ACE Director (Alumnus). He has been speaking at conferences since 2009 including QCon, Devoxx, Strata, Kafka Summit, and Øredev. You can find his talks online, subscribe to his YouTube channel (thanks lockdown!), and read his blog.

A LETTER FROM THE EDITOR



Daniel Bryant

works as a Product Architect at Datawire, and is the News Manager at InfoQ, and Chair for QCon London. His current technical expertise focuses on 'DevOps' tooling, cloud/container platforms and microservice implementations. Daniel is a leader within the London Java Community (LJC), contributes to several open source projects, writes for well-known technical websites such as InfoQ, O'Reilly, and DZone, and regularly presents at international conferences such as QCon, JavaOne, and Devoxx.

Application Programming Interfaces (APIs) are seemingly everywhere. Thanks to the popularity of web-based products, cloud-based X-as-a-service offerings, and IoT, it is becoming increasingly important for engineers to understand all aspects of APIs, from design, to building, to operation.

Research shows that there is increasing demand for near real-time APIs, in which speed and flexibility of response is vitally important. This emag explores this emerging trend in more detail.

In "Real-Time APIs: Mike Amundsen on Designing for Speed and Observability", we learn how to meet the increasing demand placed on API-based systems, and explore three areas for consideration: architecting for performance, monitoring for performance, and managing for performance.

Amundsen argues that understanding a system and the resulting performance -- and being able to identify bottlenecks -- is critical to meeting performance targets. Services must also be monitored, both from an operational perspective (SLIs), and also a business perspective (KPIs).

According to Akamai research, API calls now make up 83% of all web traffic. In "Four Case Studies for Implementing Real-Time APIs", Karthik Krishnaswamy explores the idea that competitive advantage is no longer won by simply having APIs; the key to gaining ground is based on the performance and the reliability of those APIs.

The four case studies all provide interesting lessons. However, a key takeaway is that when revenue is correlated with speed, performance trumps feature richness in API management solutions. Teams should recognize when this is the case, and design system applications and infrastructure accordingly.

In "Load Testing APIs and Websites with Gatling: It's Never Too Late to Get Started", Guillaume Corre discusses how to conduct load tests against APIs

and websites that can both validate performance after a long stretch of development and also get useful feedback from an application in order to increase its scaling capabilities and performance.

Corre argues that engineers should avoid creating "the cathedral" of load testing and end up with little time to improve performance overall. Instead, write the simplest possible test and iterate from there. It is important to establish the goals, constraints, and conditions of any load test. And always identify and verify any assumptions.

In "Real-Time APIs in the Context of Apache Kafka", we change gears, and Robin Moffatt explores one of the challenges that we have always faced in building systems: how to exchange information between them efficiently whilst retaining the flexibility to modify the interfaces without undue impact elsewhere.

Moffatt argues that events offer a Goldilocks-style approach in which real-time APIs can be used as the foundation for applications that are flexible yet performant; loosely-coupled yet efficient. He goes on to provide examples using Apache Kafka, a scalable event streaming platform with which you can build applications around the powerful concept of events.

In "The Challenges of Building a Reliable Real-Time Event-Driven Ecosystem", Matthew O'Riordan argues that to truly benefit from the power of real-time data, the entire tech stack needs to be event-driven.

When it comes to event-driven APIs, engineers can choose between multiple different protocols. Options include the simple webhook, the newer WebSub, popular open protocols such as WebSockets, MQTT or SSE. In addition to choosing a protocol, engineers also have to think about subscription models: server-initiated (push-based) or client-initiated (pull-based).

We hope you enjoy this collection of articles focused on real-time APIs. Please provide feedback via editors@infoq.com or find us on Twitter.

The Challenges of Building a Reliable Real-Time Event-Driven Ecosystem

by **Matthew O'Riordan**, Co-Founder of Ably

Globally, there is an increasing appetite for data delivered in real time. Since both producers and consumers are more and more interested in faster experiences and instantaneous data transactions, we are witnessing the emergence of the real-time API.

This new type of event-driven API is suitable for a wide variety of use cases. It can be used to power real-time functionality and technologies such as chat, alerts, and notifications or IoT devices. Real-time APIs can also be used to stream high volumes of data between different businesses or different components of a system.

This article starts by exploring the fundamental differences between the REST model and real-time APIs. Up next, we dive into some of the many engineering challenges and considerations involved in building a reliable and scalable event-driven ecosystem, such as choosing the right communication protocol and sub scription model, managing client, and server-side complexity, orscaling to support high-volume data streams.

What exactly is a real-time API?

Usually, when we talk about data being delivered in real time, we think about speed. By this logic, one could assume that improving REST APIs to be more responsive and able to execute operations in real time (or as close as possible) makes them real-time APIs. However, that's just an improvement of an existing condition, not a fundamental change. Just because a traditional REST API can deliver data in real time does not make it a real-time API.

The basic premise around real-time APIs is that they are event-driven. According to the event-driven design pattern, a system should react or respond to events as they happen. Multiple types of APIs can be regarded as event-driven, as illustrated in Figure 1.

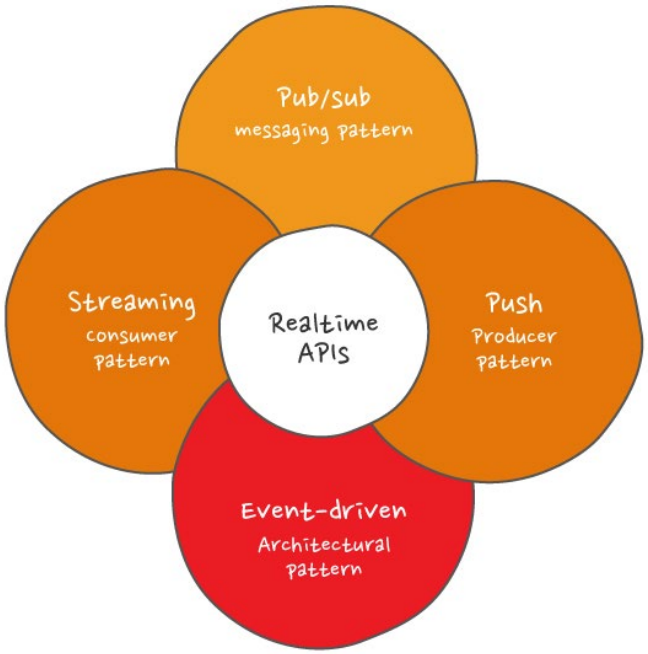


Figure 1 The real-time API family. Source: Ably

Streaming, Pub/Sub, and Push are patterns that can be successfully delivered via an event-driven architecture. This makes all of them fall under the umbrella of event-driven APIs.

Unlike the popular request-response model of REST APIs, event-driven APIs follow an asynchronous communication model. An event-driven architecture consists of the following main components:

- **Event producers**—they push data to channels whenever an event takes place.
- **Channels**—they push the data received from event producers to event consumers.
- **Event consumers**—they subscribe to channels and consume the data.

Let's look at a simple and fictional example to better understand how these components interact. Let's say we have a football app that uses a data stream to deliver real-time updates to end-users whenever something relevant happens on the field. If a goal is scored, the event is pushed to a channel. When a consumer uses the app, they connect to the respective channel, which then pushes the event to the client device.

Note that in an event-driven architecture, producers and consumers are decoupled. Components perform their task independently and are unaware of each other. This separation of concerns allows you to more reliably scale a real-time system and it can prevent potential issues with one of the components from impacting the other ones.

Compared to REST, event-driven APIs invert complexity and put more responsibility on the shoulders of producers rather than consumers. (See Figure 2)

This complexity inversion relates to the very foundation of the way event-driven APIs are designed. While in a REST paradigm the consumer is always responsible for maintaining state and always has to trigger requests to get updates. In an event-driven system, the producer is responsible for maintaining state and pushing updates to the consumer.

Event-driven architecture considerations

Building a dependable event-driven architecture is by no means an easy feat. There is an entire array of engineering challenges you will have to face and decisions you will have to make. Among them, protocol fragmen-

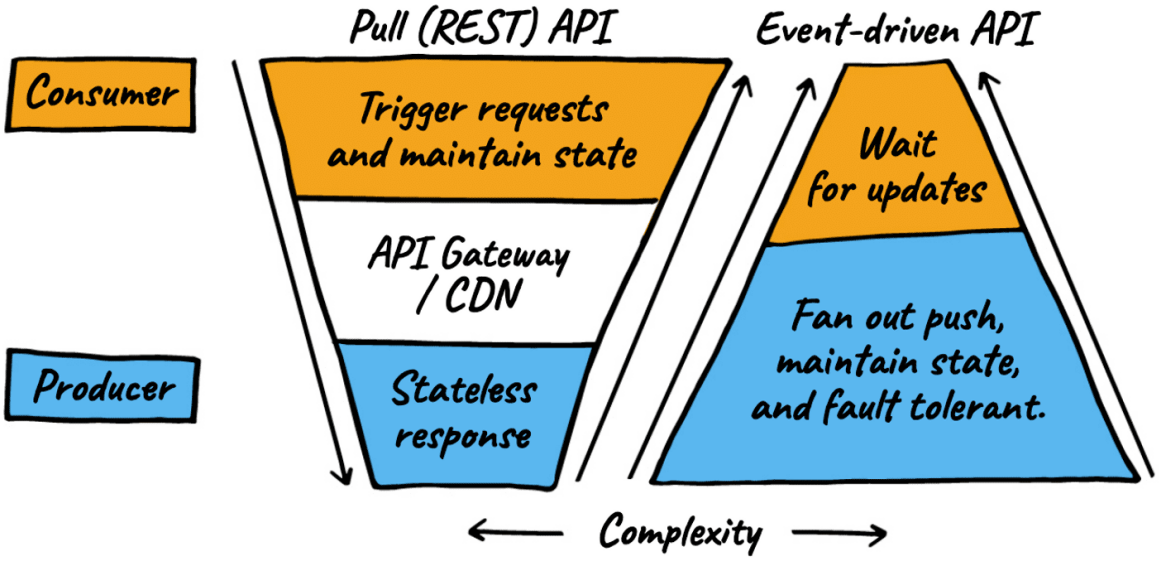


Figure 2 REST vs event-driven: complexity is inverted. Source: Ably

tation and choosing the right subscription model (client-initiated or server-initiated) for your specific use case are some of the most pressing things you need to consider.

While traditional REST APIs all use HTTP as the transport and protocol layer, the situation is much more complex when it comes to event-driven APIs. You can choose between multiple different protocols. Options include the simple webhook, the newer WebSub, popular open protocols such as WebSockets, MQTT or SSE, or even streaming protocols, such as Kafka.

This diversity can be a double-edged sword—on one hand, you aren't restricted to only one protocol; on the other hand, you need to select the best one for your use case, which adds an additional layer of engineering complexity.

Besides choosing a protocol, you also have to think about subscription models: server-initiated (push-based) or client-initiated (pull-based). Note that some protocols can be used with both models, while some protocols only support one of the two subscription approaches. Of course, this brings even more engineering complexity to the table.

In a client-initiated model, the consumer is responsible for connecting and subscribing to an event-driven data stream. This model is simpler from a producer perspective: if no consumers are subscribed to the data stream, the producer has no work to do and relies on clients to decide when to reconnect. Additionally, complexities around maintaining state are also handled by consumers. Wildly popular and effective even when high volumes of data are involved, WebSockets represent the most common example of a client-initiated protocol.

In contrast, with a server-initiated approach, the producer is responsible for pushing data to consumers whenever an event occurs. This model is often preferable for consumers, especially when the volume of data increases, as they are not responsible for maintaining any state—this responsibility sits with the producer. The common webhook—which is great for pushing rather infrequent low-latency updates—is the most obvious example of a server-initiated protocol.

We are now going to dive into more details and explore the strengths, weaknesses, and engineering complexities of client-initiated and server-initiated subscriptions.

Client-initiated vs. server-initiated models—challenges and use cases

Client-initiated models are the best choice for last-mile delivery of data to end-user devices. These devices only need access to data when they are online (connected) and don't care what happens when they are disconnected. Due to this fact, the complexity of the producer is reduced, as the server-side doesn't need to be stateful. The complexity is kept at a low level even on the consumer side: generally, all client devices have to do is connect and subscribe to the channels they want to listen to for messages.

There are several client-initiated protocols you can choose from. The most popular and efficient ones are:

- **WebSocket**. Provides full-duplex communication channels over a single TCP connection. Much lower overhead than half-duplex alternatives such as HTTP polling. Great choice for financial tickers, location-based apps, and chat solutions.
- **MQTT**. The go-to protocol for streaming data between devices with limited CPU power and/or battery life, and networks with expensive or low bandwidth, unpredictable stability, or high latency. Great for IoT.
- **SSE**. Open, lightweight, subscribe-only protocol for event-driven data streams. Ideal for subscribing to data feeds, such as live sport updates.

Among these, WebSocket is arguably the most widely-used protocol. There are even a couple of proprietary protocols and open solutions that are built on top of raw WebSockets, such as Socket.IO. All of them are generally lightweight, and they are well supported by various development platforms and programming languages. This makes them ideal for B2C data delivery.

Let's look at a real-life use case to demonstrate how WebSocket-based solutions can be used to power a client-initiated event-driven system. Tennis Australia (the governing body for tennis in Australia) wanted a solution that would allow them to stream real-time rally and commentary updates to tennis fans browsing the Australian Open website. Tennis Australia had no way of knowing how many client devices could subscribe to updates at any given moment, nor where these devices could be located throughout the world. Additionally, client devices are generally unpredictable—they can connect and disconnect at any moment.

Due to these constraints, a client-initiated model where a client device would open a connection whenever it wanted to subscribe to updates was the right way to go. However, since millions of client devices could connect at the same time, it wouldn't have been scalable to have a 1:1 relationship with each client device.

Tennis Australia was interested in keeping engineering complexity to a minimum—they wanted to publish one message every time there was an update and distribute that message to all connected client devices via a message broker.

In the end, instead of building their own proprietary solution, Tennis Australia chose to use Ably as the message broker. This enables Tennis Australia to keep things very simple on their side—all they have to do is publish a message to Ably whenever there's a score update. The message looks something like this:

```
var ably = new Ably.
Realtime('API_KEY');
var channel = ably.
channels.get('tennis-score-
updates');

// Publish a message to
the tennis-score-updates
channel
channel.publish('score',
'Game Point!');
```

Ably then distributes that message to all connected client devices over WebSockets, by using a pub/sub approach, while also handling most of the engineering complexity on the producer side, such as connection churn, backpressure, or message fan-out.

Things are kept simple for consumers as well. All a client device had to do is open a WebSocket connection and subscribe to updates:

```
// Subscribe to messages on
channel
channel.subscribe('score',
function(message) {
  alert(message.data);
});
```

Traditionally, developers use the client-initiated model to build apps for end-users. It's a sensible choice since protocols like WebSockets, MQTT, or SSE can be successfully used to stream frequent updates to a high number of users, as demonstrated by the Tennis Australia example. However, it's hazardous to think that client-initiated models scale well when high-throughput streams of data are involved—I'm referring to scenarios where businesses exchange large volumes of data, or where an organization is streaming information from one system to another.

In such cases, it's usually not practical to stream all that data over a single consumer-initiated connection (between one server that is the producer, and another one that is the consumer). Often, to manage the influx of data, the consumer needs to shard it across multiple nodes. But by doing so, the consumer also has to figure out how to distribute these smaller streams of data across multiple connections and deal with other complex engineering challenges, such as fault tolerance.

We'll use an example to better illustrate some of the consumer-side complexities. For ex-

ample, let's say you have two servers (A and B) that are consuming two streams of data. Let's imagine that server A fails. How does server B know it needs to pick up additional work? How does it even know where server A left off? This is just a basic example, but imagine how hard it would be to manage hundreds of servers consuming hundreds of data streams. As a general rule, when there's a lot of complexity involved, it's the producer's responsibility to handle it; data ingestion should be as simple as possible for the consumer.

That's why in the case of streaming data at scale you should adopt a server-initiated model. This way, the responsibility of sharding data across multiple connections and managing those connections rests with the producer. Things are kept rather simple on the consumer side—they would typically have to use a load balancer to distribute the incoming data streams to available nodes for consumption, but that's about as complex as it gets.

Webhooks are often used in server-initiated models. The webhook is a very popular pattern because it's simple and effective. As a consumer, you would have a load balancer that receives webhook requests and distributes them to servers to be processed. However, webhooks become less and less effective as the volume of data increases. Webhooks are HTTP-based, so there's an overhead with each webhook event (message) because each one triggers a new request. In addition, webhooks provide no integrity or message ordering guarantees.

That's why for streaming data at scale you should usually go with a streaming protocol such as AMQP, Kafka, or ActiveMQ, to name just a few. Streaming protocols generally have much lower overheads per message, and they provide ordering and integrity guarantees. They can even provide additional benefits—idempotency, for example. Last but not least, streaming protocols enable you to shard data before streaming it to consumers.

It's time to look at a real-life implementation of a server-initiated model. HubSpot is a well-known developer of marketing, sales, and customer service software. As part of its offering, HubSpot provides a chat service (Conversations) that enables communication between end-users. The organization is also interested in streaming all that chat data to other HubSpot services for onward processing and persistent storage. Using a client-initiated subscription to successfully stream high volumes of data to their internal message buses is not really an option. For this to happen, HubSpot would need to know what channels are active at any point in time, to pull data from them.

To avoid having to deal with complex engineering challenges, HubSpot decided to use Ably as a message broker that enables chat communication between end-users. Furthermore, Ably uses a server-initiated model to push chat data into Amazon Kinesis, which is the data processing component of HubSpot's message bus ecosystem. (See Figure 3)

Consumer complexity is kept to a minimum. HubSpot only has to expose a Kinesis endpoint and Ably streams the chat data over as many connections as needed.

A brief conclusion

Hopefully, this article offers a taste of what real-time APIs are and helps readers navigate some of the many complexities and challenges of building an effective real-time architecture. It is naive to think that by improving a traditional REST API to be quicker and more responsive you get a real-time API. Real time in the context of APIs means so much more.

By design, real-time APIs are event-driven; this is a fundamental shift from the request-response pattern of RESTful services. In the event-driven paradigm, the responsibility is inverted, and the core of engineering complexities rests with the data producer, with the purpose of making data ingestion as easy as possible for the consumer.

But having one real-time API is not enough—this is not a solution to a problem. To truly benefit from the power of real-time data, your entire tech stack needs to be event-driven. Perhaps we should start talking more about event-driven architectures than about event-driven APIs. After all, can pushing high volumes of data to an endpoint (see HubSpot example above for details) even be classified as an API?

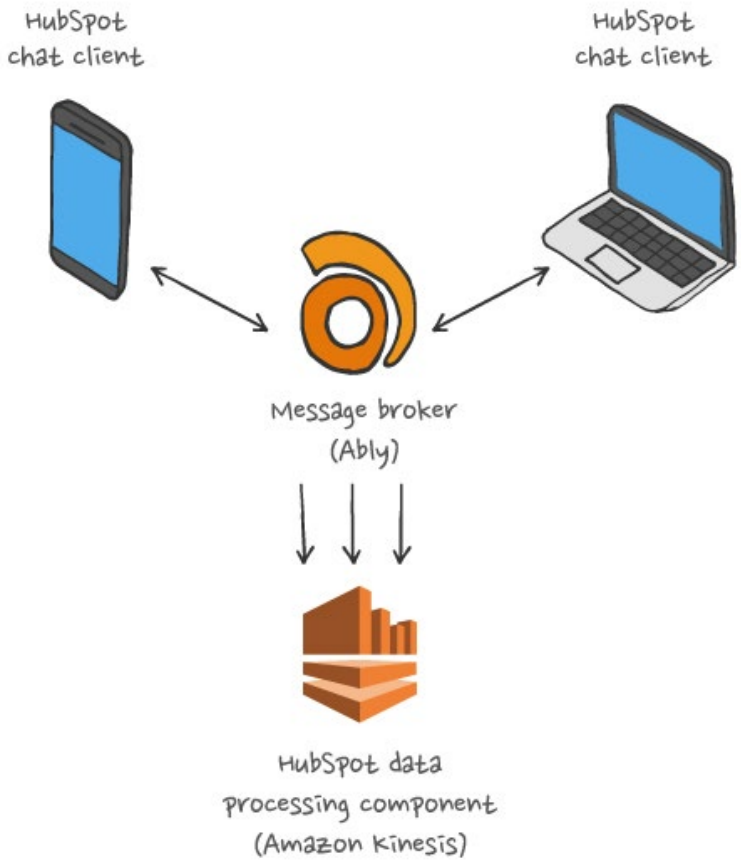
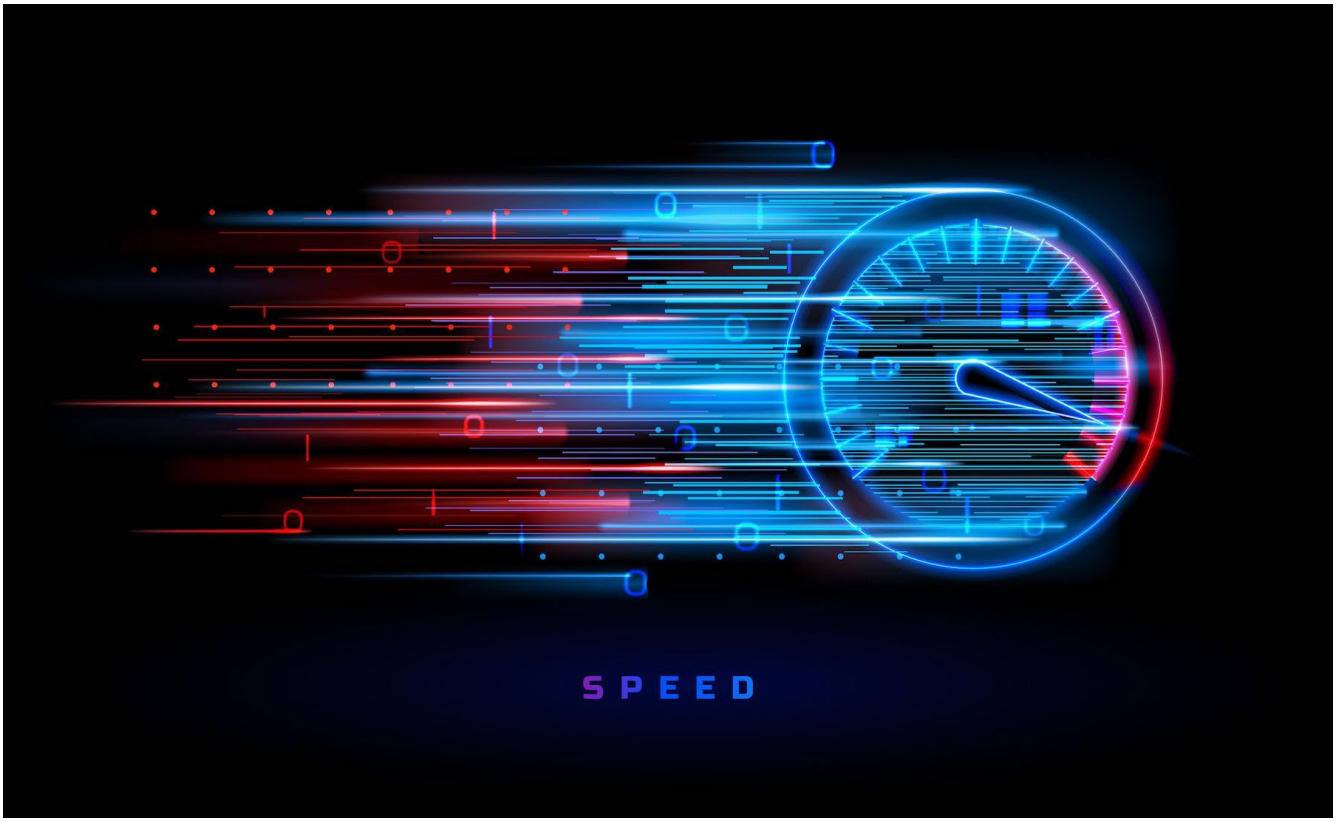


Fig 3 High-level overview of HubSpot chat architecture.
Source: Ably

TL;DR

- Globally, there is an increasing appetite for data delivered in real time. As both producers and consumers are more and more interested in faster experiences and instantaneous data transactions, we are witnessing the emergence of the real-time API.
- When it comes to event-driven APIs, engineers can choose between multiple different protocols. Options include the simple webhook, the newer WebSub, popular open protocols such as WebSockets, MQTT or SSE, or even streaming protocols, such as Kafka. In addition to choosing a protocol, engineers also have to think about subscription models: server-initiated (push-based) or client-initiated (pull-based).
- Client-initiated models are the best choice for the "last mile" delivery of data to end-user devices. These devices only need access to data when they are online (connected) and don't care what happens when they are disconnected. Due to this fact, the complexity of the producer is reduced, as the server-side doesn't need to be stateful.
- In the case of streaming data at scale, engineers should adopt a server-initiated model. The responsibility of sharding data across multiple connections and managing those connections rests with the producer, and other than the potential use of a client-side load balancer, things are kept rather simple on the consumer side.



Real-Time APIs: Mike Amundsen on Designing for Speed and Observability

by **Daniel Bryant**, Product Architect @ambassadorlabs | News Manager @InfoQ | Chair @QConLondon

In a recent apidays webinar, Mike Amundsen, trainer and author of the recent O'Reilly book API Traffic Management 101, presented "High Performing APIs: Architecting for Speed at Scale". Drawing on recent research by IDC, he argued that organizations will have to drive systemic changes to meet the upcoming increased demand for consumption of business services via APIs. This change in requirements relates to both an increased scale of operation and a decreased response time.

The changing nature of customer expectations, combined with new data-driven products and an increase in consumption at the edge (mobile, IoT, etc.), has meant that the need for low latency "real-time APIs" is rapidly becoming the norm.

The recent adoption of cloud technology and microservices-based architecture has enabled innovation and increased speed of development throughout the business world.

However, this technology and architecture style is not always conducive to creating performant applications; in the cloud nearly everything communicates over a virtualized network, and in a

service-oriented architecture multiple separate processes are typically invoked to fulfill each business function. Amundsen stated that a series of holistic changes to how systems are designed, operated, and managed is required to meet the new demands.

The performance imperative

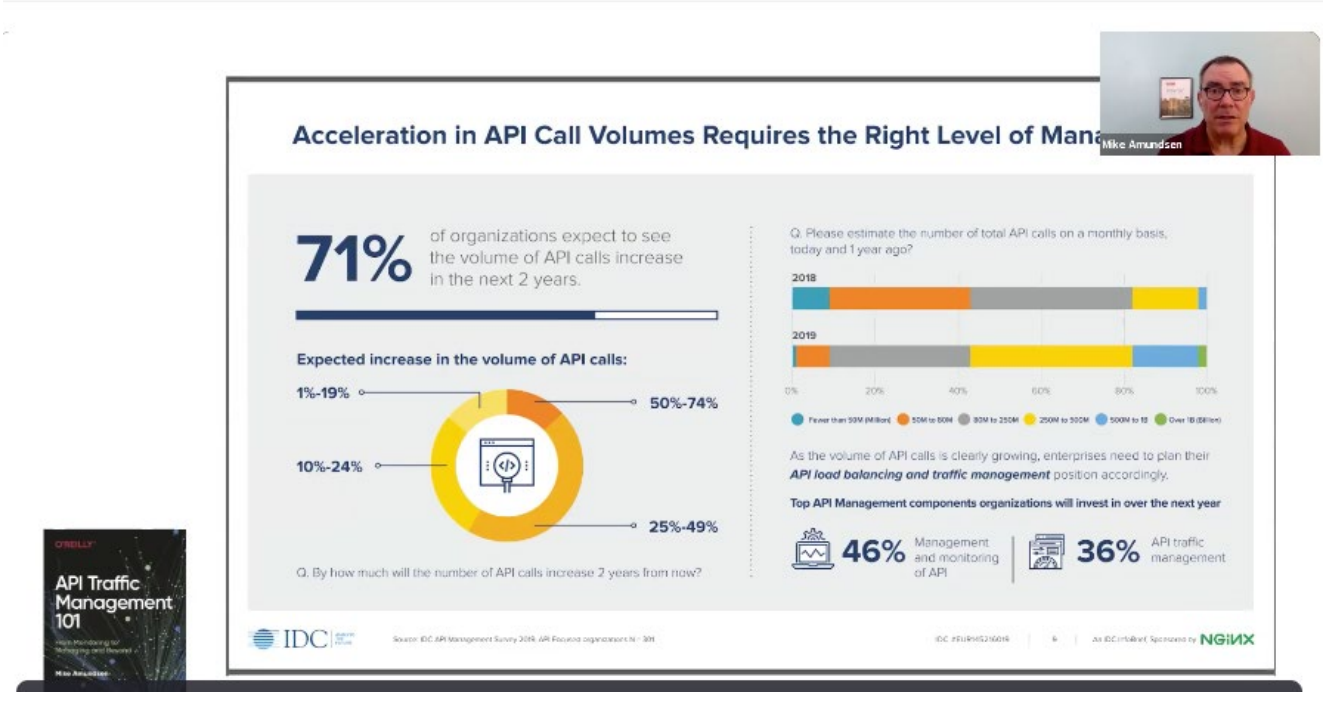
Amundsen began his presentation by describing the "performance imperative" he is seeing throughout the IT industry. Focusing first on the ecosystem transformation, he referenced a 2019 IDC research report "IDC MaturityScape: Digital Transformation Platforms 1.0." This report states that 75% of organizations will be completely digitally transformed in the next decade, with those companies not embracing a modern way of working not surviving. By 2022, 90% of new applications being built will feature a microservice architecture, and 35% of all production applications will be "cloud native."

API call volumes are increasing as more organizations embrace digital transformations. According to the same report, 71% of organizations expect to see the volume of API calls increase in the next 2 years. About 60% expect over 250 million API calls per month (~10 million per business day). There is an increasing focus on transaction response time, with 59% of organizations expecting the latency of a typical API request to be under 20 milliseconds, and 93% expecting a latency under 50 milliseconds.

Amundsen argued that to meet the increasing demand placed on organizations, there are three areas for consideration: architecting for performance, monitoring for performance, and managing for performance. (See figure below)

Architecting for performance

Migrating applications to a cloud vendor's platform brings many benefits, such as the ability to take advantage of data processing or machine learning-based services to innovate rapidly, or reducing the total cost of ownership (TCO) of an organization's platform. However, cloud infrastructure can be significantly different than traditional hardware, both in terms of configuration and performance. Amundsen cautioned that performing a "lift and shift" of an existing application is not enough to ensure performance.



The vast majority of infrastructure components and services within a cloud platform are connected over a network—e.g. block stores are often implemented as network-attached storage (NAS). Colocation of components is not guaranteed—e.g. an API gateway might be located in a different physical data center than the virtual machine (VM) a backend application is running on. The global reach of cloud platforms opens opportunities for reaching new customers and also provides more effective disaster recovery options, but this also means that the distance between your customers and your services can increase.

Amundsen suggested that engineering teams redesign components into smaller services, following the best practices associated with the microservices architectural style. These services can be independently released and independently scaled. Embracing asynchronous methods of communicating, such as messaging and events, can reduce wait states. This offers the possibility of an API call rapidly returning an acknowledgment and decreasing latency from an end-user perspective, even if the actual work has been queued at the backend. Engineers will also need to build in transaction reversal and recovery processes to reduce the impact of inevitable failures. Additional thought leaders in this space, such as Bernd Ruecker, encourage engineers to build "smart APIs" and learn more about business process modeling and event sourcing.

For systems to perform as required, data read and write patterns will frequently have to be reengineered. Amundsen suggested judicious use of caching results, which can remove the need to constantly query upstream services. Data may also need to be "staged" appropriately throughout the entire end-to-end request handling process. For example, caching results and data in localized points of presence (PoPs) via content delivery networks (CDNs), caching in an API gateway, and replication of data stores across availability zones (local data centers) and globally. For some high transaction throughput use cases, writes may have to be streamed to meet demand, for example, writing data locally or via a high throughput distributed logging system like Apache Kafka for writing to an external data store at a later point in time.

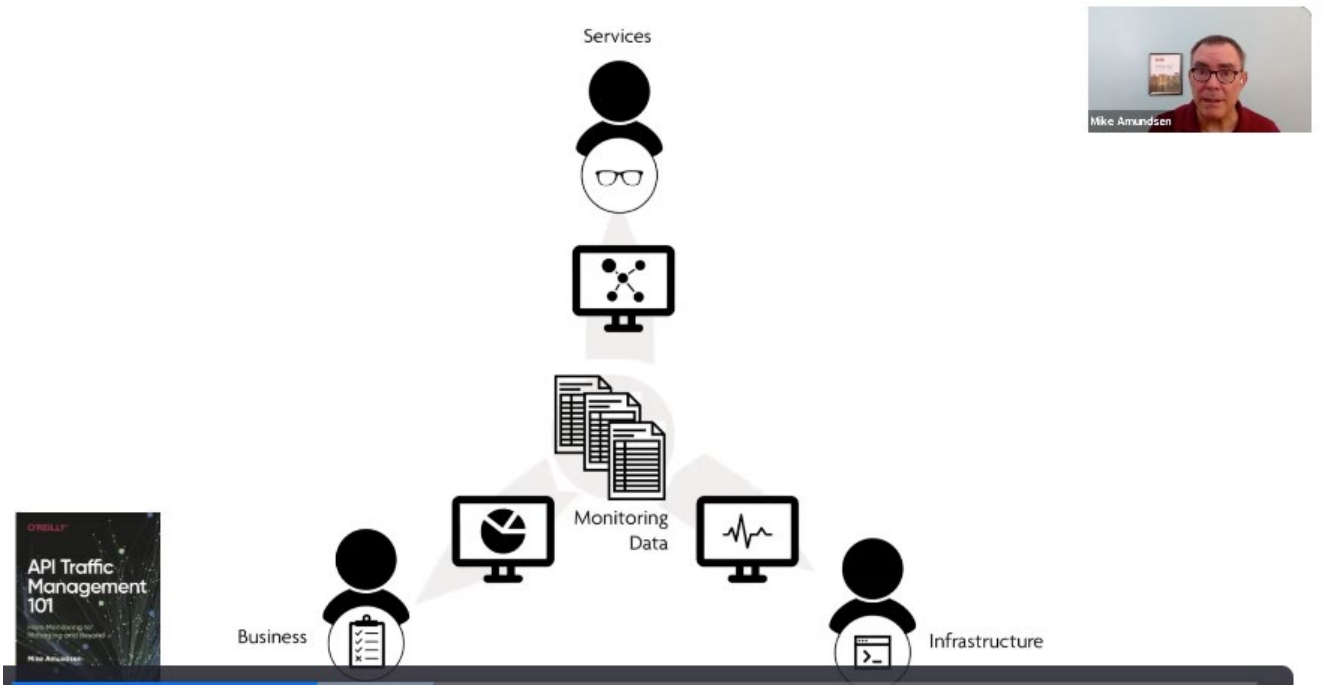
Engineers may have to "rethink the network," (respecting the eight fallacies of distributed computing), and design their cloud infrastructure to follow best practices relevant to their cloud vendor and application architecture. Decreasing request and response size may also be required to meet demands. This may be engineered in tandem with the ability to increase the message volume. The industry may see the "return of the RPC," with strongly-typed communication contracts and high-performance binary protocols. As convenient as JSON is, compared to HTTP, a lot of computation (and time) is used in serialization and deserialization, and the text-

based message payloads sent over the wire are typically larger.

Monitoring for performance

Understanding a system and the resulting performance, and being able to identify bottlenecks, are critical to meeting performance targets. Observability and monitoring are critical aspects of any modern application platform. Infrastructure must be monitored, from machines to all aspects of the networking stack. Modern cloud networking may consist of many layers, from the high-level API gateway and service mesh implementations to the cloud software-defined networking (SDN) components to the actual virtualized and physical networking hardware—and these need to be instrumented effectively.

Services must also be monitored, both from an operational perspective and also a business perspective. Many organizations are increasingly adopting the site reliability engineering (SRE) approach of defining and collecting service level indicators (SLIs) for operational metrics, which consist of top-line metrics such as utilization, saturation, and errors (USE), or request rate, errors, and duration (RED). Business metrics are typically related to key performance indicators (KPIs) that are driven by an organization's objectives and key results (OKRs).



With infrastructure emitting metrics and services producing SLI- and KPI-based metrics, the corresponding data must be effectively captured, processed, and presented to key stakeholders for this to be acted on. This is often a cultural challenge as much as it is a technical challenge.

Managing for performance

Amundsen stated that the organization should aim to create a "dashboard culture." Monitoring top-line metrics, user traffic, and the continuous delivery process for "insight" into how a system is performing are vitally important. As stated by Dr. Nicole Forsgren and colleagues in Accelerate, the four key metrics that are correlated with high performing organizations are lead time, deployment frequency, mean time to restore (MTTR), and change failure

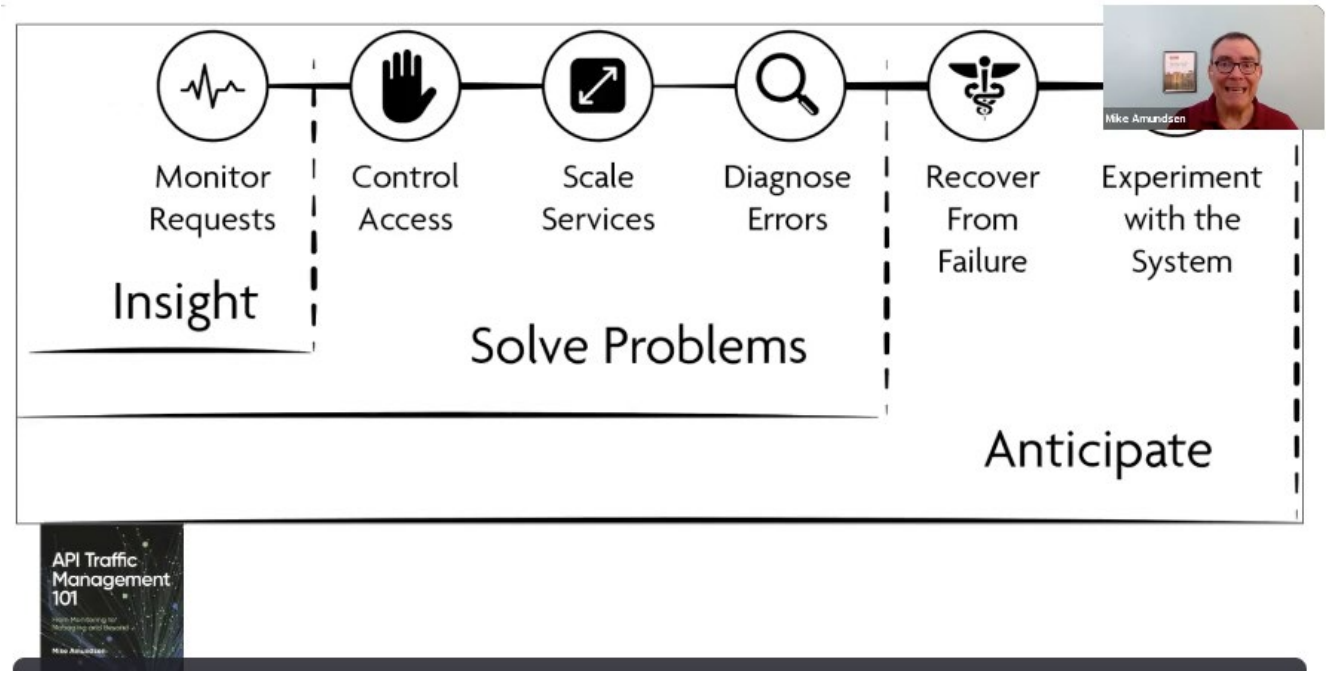
percentage. Engineers should be able to use internal tooling to effectively solve problems, such as controlling access to functionality, scaling services, and diagnosing errors.

The addition of security mitigation is often in conflict with performance—for example, adding layers of security verification can slow responses—but this is a delicate balancing act. Engineers should be able to understand systems, their threat models, and whether there are any indicators of current problems—e.g. an increased number of HTTP 503 error codes being generated, or a gateway being overloaded with traffic.

The elastic nature of cloud infrastructure typically allows almost unlimited scaling, albeit with potentially unlimited cost. However, the culture of managing for performance requires the ability to easily understand how current systems are being used. For example,engineers need to know whether scaling specific components is an appropriate response to seeing increased loads at any given point in time.

Diagnosing errors in a cloud-based microservices system typically requires an effective user experience (or developer experience) in addition to requiring a collection of tooling to collect, process, and display metrics, logs, and traces. Dashboards and tooling should show top-level metrics related to a customer's experience, and also allow engineers to test hypotheses for issues by drilling-down to see individual service-to-service metrics. Being able to answer ad hoc questions of an observability or monitoring solution is currently an area of active research and product development.

Once the ability to gain insight and solve problems has been obtained, the next stage of managing for performance is the ability to anticipate. This is the capability to automatically recover from failure—building antifragile systems—and the ability to experiment with the system, such as examining fault-tolerance properties via the use of chaos engineering.



Summary

Amundsen concluded the presentation by reminding the audience of the IDC research report; the IT industry should prepare for API call volumes to increase and the requirements for transaction processing time to decrease. Supporting these new requirements will demand organization and system-wide changes.

Software developers will have to learn about redesigning services and reengineering data, and instrumenting services for both increased visibility into operational and business metrics. Platform teams will have to consider rethinking networks, monitoring infrastructure, and managing traffic. Application operation and SRE teams will need to work across the engineering organization to enable the effective identification and resolution of problems, and also anticipate issues and enable experimentation.

A more detailed exploration into the topics discussed here can be found in Mike Amundsen's recent O'Reilly book API Traffic Management 101.

TL;DR

- The "IDC MaturityScape: Digital Transformation Platforms 1.0" report states that 71% of organizations expect to see the volume of API calls increase in the next 2 years. 59% of organizations expect the latency of a typical API request to be under 20 milliseconds, and 93% expect a latency under 50 milliseconds.
- According to the same report, 90% of new applications being built will feature a microservice architecture, and 35% of all production applications will be "cloud native."
- To meet the increasing demand placed on API-based systems, there are three areas for consideration: architecting for performance, monitoring for performance, and managing for performance.
- Good architectural practices include avoiding a simple "lift and shift" to the cloud, embracing asynchronous processing, and reengineering data and networks.
- Understanding a system and the resulting performance, and being able to identify bottlenecks, is critical to meeting performance targets. Services must also be monitored, both from an operational perspective (SLIs), and also a business perspective (KPIs).

Load Testing APIs and Websites with Gatling: It's Never Too Late to Get Started

by **Guillaume Corré**, Tech Lead, Gatling Corp

You open your software engineering logbook and start writing—"Research and Development Team's log. Never could we have foreseen such a failure with our application and frankly, I don't get it. We had it all; it was a masterpiece! Tests were green, metrics were good, users were happy, and yet when the traffic arrived in droves, we still failed."

Pausing to take a deep breath, you continue—"We thought we were prepared; we thought we had the hang of running a popular site. How did we do? Not even close. We initially thought that a link on TechCrunch wouldn't bring that much traffic. We also thought we could handle the spike in load after our TV spot ran. What was the point of all this testing if the application stopped responding right after launch, and we couldn't fix this regardless of the number of servers we threw at it, trying to salvage the situation as we could."

After a long pause, "It's already late, time goes by. There is so much left to tell you, so many things I wish I knew, so many mistakes I wish I never made." Unstoppable, you write, "If only there was something we could have done to make it work..."

And then, lucidity striking back at you—you wonder: why do we even do load testing in the first place? Is it to validate performance after a long stretch of development or is it really to get useful feedback from our application and increase its scaling capabilities and performance? In both cases validation takes place, but in the latter, getting feedback is at the center of the process.

You see, this isn't really about load testing per se, the goal is focused on making sure your application won't crash when it goes live. You don't want to be writing "the cathedral" of load testing and end up with little time to improve performance overall. Often, working on improvements is where you want to spend most if not all of your time. Load testing is just a means to an end and nothing else.

If you are afraid because your deadline is tomorrow and you are looking for a quick win, then welcome, this is the article for you.

Writing the simplest possible simulation

In this article, we will be writing a bit of code, Scala code to be more precise as we will be using Gatling. While the code might look scary, I can assure it is not. In fact, Scala can be left aside for the moment and we can think about Gatling as its own language:

```
import io.gatling.core.Predef._
import io.gatling.http.Predef._

class SimplestPossibleSimulation extends
Simulation {

  val baseHttpProtocol =
    http.baseUrl("https://computer-
database.gatling.io")

  val scn = scenario("simplest")
    .exec(
      http("Home")
        .get("/")
    )

  setUp(
    scn.inject(atOnceUsers(1))
  ).protocols(baseHttpProtocol)
}
```

Let's break this down into smaller parts. As Gatling is built using Scala, itself running on the Java Virtual Machine, there are some similarities to expect. The first one is that the code always starts with package imports:

```
import io.gatling.core.Predef._
import io.gatling.http.Predef._
```

They contain all the Gatling language definitions and you'll use these all the time, along with other imports, depending on the need. Then, you create enough to encapsulate the simulation:

```
class SimplestPossibleSimulation extends
Simulation {}
```

`SimplestPossibleSimulation` is the name of the simulation and Simulation is a construct defined by Gatling that you "extend" and will contain all the simulation code, in three parts: (1) we need to define which protocol we want to use and which parameters we want it to have:

```
val baseHttpProtocol =
  http.baseUrl("https://computer-database.
gatling.io")
```

(2) The code of the scenario itself:

```
val scn = scenario("simplest")
  .exec(
    http("Home")
      .get("/")
  )
```

Notice I used the term scenario even though we only spoke about simulations until now. While they are often conflated, they have both really distinct meanings:

- A **scenario** describes the journey a single user performs on the application, navigating from page to page, endpoint to endpoint, and so on, depending on the type of application
- A **simulation** is the definition of a complete test with populations (e.g.: admins and users) assigned to their scenario
- When you launch a simulation, the result is called a run

This terminology helps make sense of the final section, (3) setting up the simulation:

```
setUp(
  scn.inject(atOnceUsers(1))
).protocols(baseHttpProtocol)
```

Since a simulation is roughly a collection of scenarios, it is the sum of all the scenarios configured with their own injection profile (which we will ignore for now). The protocol on which is based all the requests done in a scenario can be either shared, if chained after setUp, or defined per scenario, if we had multiple ones, scn1 and scn2, as such:

```
setUp(
  scn1.inject(atOnceUsers(1)).
protocols(baseHttpProtocol1),
  scn2.inject(atOnceUsers(1)).
protocols(baseHttpProtocol2)
)
```

Notice how everything is chained, separated by a comma: scn1 configured with such injection profile and such protocol, etc.

When you run a Gatling simulation, it will launch all the scenarios configured inside at the same time, which is what we'll do right now.

Running a simulation

One way to run a Gatling simulation is by using the Gatling Open-Source bundle. It is a self-contained project with a folder to drop simulation files inside and a script to run them.

Warning!

All installations require a proper Java installation, JDK 8 minimum, see the Gatling installation docs for all the details.

After unzipping the bundle, you can create a file named `plestPossibleSimulation.scala` inside the `user-files/simulations` folder. Then, from the command line, at the root of the bundle folder, type:

```
./bin/gatling.sh
```

Or, on Windows:

```
.\bin\gatling.bat
```

This will scan simulations inside the previous folder and prompt you, asking which one you want to run. As the bundle contains some examples, you'll have more than one to choose from.

While this is the easiest way to run a Gatling simulation, it doesn't work well with source repositories. The preferred way is to use a build tool, such as Maven, or SBT. If this is the solution you would like to try, you can clone our demo repositories using git:

- Gatling Maven Plugin Demo
- Gatling's SBT plugin demo

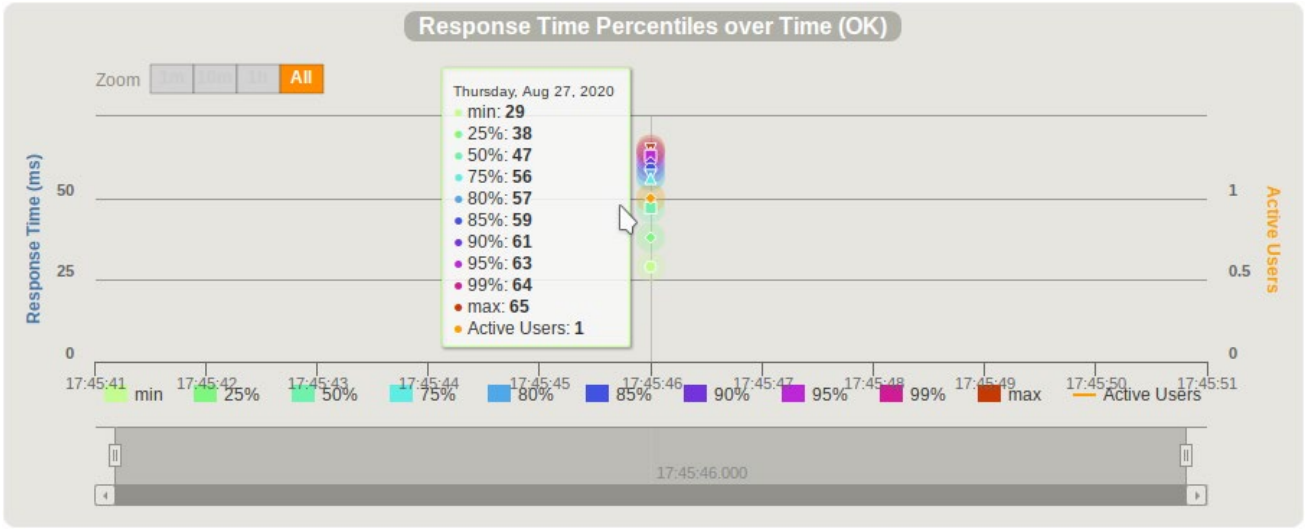
Or, if you want to follow along with this article, you can clone the following repository which was made for the occasion:

- Gatling Maven and SBT project

After running the test, you notice there is a URL at the end of the output, which you open and stumble upon:

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	2	2	0	0%	2	29	47	56	63	65	65	47	18
Home	1	1	0	0%	1	65	65	65	65	65	65	65	0
Home Redirect 1	1	1	0	0%	1	29	29	29	29	29	29	29	0



You will notice there is a request named “Home Redirect 1" which we didn't make ourselves. You might be thinking it's probably Gatling following a redirect automatically and separating the response times from all the different queries. But it does look kind of underwhelming, and you would be right.

Configuring the Injection Profile

Looking back, when configuring our scenario we did the following:

```
scn.inject(atOnceUsers(1))
```

Which is great for debugging purposes, but not really thrilling in relation to load testing. The thing is, there is no right or wrong way to write an injection profile, but first things first.

An injection profile is a chain of rules, executed in the provided order, that describes at which rate you want your users to start their own scenario. While you might not know the exact profile you must write, you almost always have an idea of the expected behavior. This is because you are typically either anticipating a lot of incoming users at a specific point in time, or you are expecting more users to come as your business grows.

Questions to think about include: do you expect users to arrive all at the same time? This would be the case if you intend to offer a flash sale or your website will appear on TV soon; and do you have an idea of the pattern your users will follow? It could be that users arrive at specific hours, are spread across the day, or appear only within working hours, etc. This knowledge will give you an angle of attack, which we call a type of test. I will present three of them.

Stress testing

When we think about load testing, often we think about “stress testing,” which it turns out is only a single type of test; “Flash sales” is the underlying meaning.

The idea is simple: lots of users, the smallest amount of time possible. atOnceUsers is perfect for that, with some caveats:

```
scn.inject(
  atOnceUsers(10000)
)
```

It is difficult to say how many users you can launch at the same time without being too demanding on hardware. There are a lot of possible limitations: CPU, memory, bandwidth, number of connections the Linux kernel can open, number of available sockets on the machine, etc. Is it easy to strain hardware by putting too high a value and ending up with nonsensical results?

This is where you could need multiple machines to run your test. To give an example, a Linux kernel, if optimized properly, can easily open 5k connections every second, 10k being already too much.

Splitting 10 thousand users over the span of a minute would still be considered a stress test, because of how strenuous the time constraint is:

```
scn.inject(
  rampUsers(10000) during 1.minute
)
```

Soak test

If the time span gets too long, the user journey will start feeling more familiar. Think “users per day,” “users per week," and so on. However, imitating how users arrive along a day is not the purpose of a soak test, just the consequence.

When soak testing, what you want to test is the system behavior under a long stretch of time: how does the CPU behave? Can we see any memory leaks? How do the disks behave? The network?

A way of doing this is to model users arriving over a long period of time:

```
scn.inject(
  rampUsers(10000000) during 10.hours  //
~277 users per sec
)
```

This will feel like doing "users per day." Still, if modeling your analytics is your goal, then computing the number of users per second that your ramp-up is doing and reducing the duration would give you results faster:

```
scn.inject(
  constantUsersPerSec(300) during
10.minutes
)
```

Speaking about this, `rampUsers` over `constantUserPerSec` is just a matter of taste. The former gives you an easy overview of the total users arriving while the latter is more about throughput. However, thinking about throughput makes it easier to do "ramping up," i.e. progressively arriving at the final destination:

```
scn.inject(
  rampUsersPerSec(1) to 300 during
10.minutes,
  constantUsersPerSec(300) during 2.hours
)
```

Capacity test

Finally, you could simply be testing how much throughput your system can handle. In which case a capacity test is the way to go. Combining methods from the previous test, the idea is to level the throughput from some arbitrary time, increase the load, level again, and continue until everything goes down and we get a limit. Something like:

```
scn.inject(
  constantUsersPerSec(10) during 5.minutes,
  rampUsersPerSec(10) to 20 during
30.seconds,
  constantUsersPerSec(20) during 5.minutes,
  rampUsersPerSec(20) to 30 during
30.seconds,
  ...
)
```

You could say doing this 20 times could be a bit cumbersome…but as the base of Gatling is code, you could either make a loop that generates the previous injection profile, or use our DSL dedicated to capacity testing:

```
scn.inject(
  incrementUsersPerSec(10)
    .times(20)
    .eachLevelLasting(5.minutes)
    .separatedByRampsLasting(30.seconds) //
optional
    .startingFrom(10) // users per sec too!
)
```

See on the next page in Figure 1 how to model this graphically. And that's it!

Where to start with loading testing?

If it is your first time load testing, whether you already know the target user behavior or not, you should start with a capacity test. Stress testing is useful but analyzing the metrics is really tricky under such a load. Since everything is failing at the same time, it makes the task difficult, even impossible. Capacity testing offers the luxury to go slowly to failure, which is more comfortable for the first analysis.

To get started, just run a capacity test that makes your application crash as soon as possible. You only need to add complexity to the scenario when everything seems to run smoothly.

Then, you need to look at the metrics. (See Figure 2).

What do all of these results even mean?

The above chart shows response time percentiles. When load testing, we could be tempted to use averages to analyze the global response time and that would be error-prone. If an average can give you a quick overview of what happened in a run, it will hide under the rug all the things you actually want to look at. This is where percentiles come in handy.

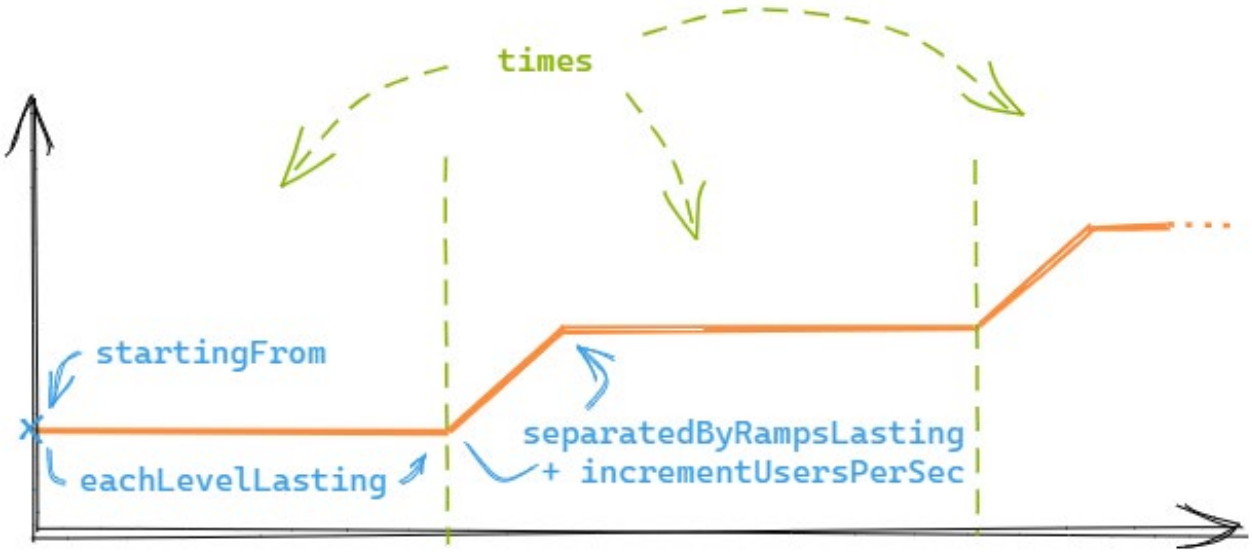


Figure 1

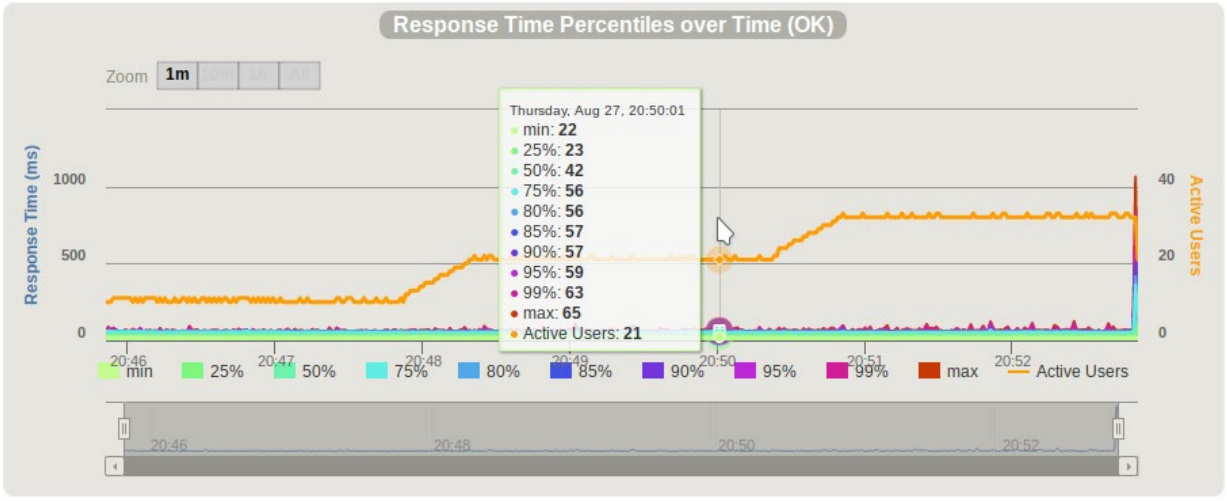


Figure 2

Think of it this way: if the average response time is some amount of milliseconds, how does the experience feel in the worst case for 1% of your user base? Better or worse? How does it feel for 0.1% of your users? And so on, getting closer and closer to zero. The higher the amount of users and requests, the closer you'll need to get to zero in order to study extreme behaviors. To give you an example, if you had 3 million users performing a single request on your application, and 0.01% of them timed-out, that would be **30 thousand users that weren't able to access your application.**

Percentiles are usually used, which correspond to thinking about this the other way around. The 1% worse case for users is turned into "how does the experience feel at best for 99% of the users," 0.1% is turned into 99.9%, etc. A way to model the computation is to sort all the response times in ascending order and mark spots:

- 0% of the responses time is the minimum, marking the lowest value
- 100% is the maximum
- 50% is the median

From here, you go to 99% and as close to 100% as possible, by adding nines, depending on how many users you have.

In the previous chart, we had a 99 percentile of 63 milliseconds, but is that good or not? With the maximum being 65, it would seem so. Would it be better if it were10 milliseconds, though?

Most of the time metrics are contextual and don't have any meaning by themselves. Broadly speaking, a 10ms response time on localhost isn't an achievement, and it would be impossible from Paris to Australia due to the speed of light constraint. We have to ask, "in what conditions was the run actually performed?" This will help us greatly deduce whether or not the run was actually that good. These conditions include:

- What is the type of server the application is running?
- Where is it located?
- What is the application doing?
- Is it under a network?
- Does it have TLS?
- What is the scenario doing?
- How do you expect the application to behave?
- Does everything run on the same cloud provider in the same data center?
- Are there any kinds of latency to be expected? Think mobile (3G), long distance.
- Etc.

This is the most important part. If you know the running conditions of a test, you can do wonders. You can (and should) test locally, with everything running on your computer, as long as you understand what it boils down to: no inference can be made on how it will run in production, but that allows you to do regression testing. Just compare the metrics between multiple runs, and it will tell you whether you made the performance better or worse.

You don't need a full-scale testing environment to do that. If you know your end-users are mobile users, testing with all machines located in a single data center could lead to a disaster. However, you don't need to be 100% realistic either. Just keep in mind what it implies and what you can deduce from the testing conditions: a data center being a huge local network, results can't be compared to mobile networks, and observed results would in fact be way better than reality.

Specifying the need

Not only should you be aware of the conditions under which the tests are run, but you should also decide beforehand what makes the test a success or a failure. To do that, we define criteria, as such:

1. Mean response time under 250ms
2. 2000 active users
3. Less than 1% failed requests

There is a catch though. Out of these three acceptance criteria, only one is actually useful to describe a system under load, can you guess which and what they can be replaced with?

(1) "Mean response time under 250ms": It should come naturally from the previous section that while average isn't bad, it isn't sufficient to describe user behavior, it should always be seconded with percentiles:

- ~~Mean response time under 250ms~~
- 99 percentile under 250ms
- Max under 1000ms

(2) "2000 active users": This one is more tricky. Nowadays we are flooded with analytics showing us "active users" and such, so it should be tempting to define acceptance criteria using this measurement. That is an issue though. The only way to model an amount of active users directly is by creating a closed model. Here, you will end up with a queue of users, and this is where the issue lies. Just think about it. If your application were to slow down and users are on a queue, they would start piling up in the queue, but only a small amount, (the maximum amount allowed in the application), would be "active" and performing requests. The amount of active users would stay the same, but users would be waiting outside as they would be in a shop.

In real life, if a website goes down, people will continue to refresh the page until something comes up, which will further worsen the performance of the website until people give up and you lose them. You shouldn't model active users unless it is your use case—you should instead target an amount of active users. It can be difficult, but doable. It will depend on: the duration of the scenario, including response time, pauses, etc., the network, the injection profile, etc.

This is what you could measure instead:

- ~~2000 active users~~
- Between 100 and 200 new users per second
- More than 100 requests per second

(3) "Less than 1% failed requests" was in fact the only criterion that properly represents a system under load between the three. However, it is not to be taken as a rule of thumb. Depending on the use case, 1% may be too high. Think of an e-commerce site, you might allow some pages to fail here and there, but having a failure at the end of the conversion funnel right before buying would be fatal to your business model. You would have this specific request with a failure criterion at 0% and the rest to either a higher percentage or no criterion at all.

All of this leads to a specification based on the Given-When-Then model and how to think about load testing in general, with everything we learned so far:

- **Given**: Injection Profile
- **When**: Scenario
- **Then**: Acceptance Criteria

Using the previous examples, it can look like this:

- **Given**: a load of 200 users per second
- **When**: users visit the home page
- **Then**:
  - We got at least 100 requests/seconds
  - 99 percentile under 250ms
  - And less than 1% failed requests

Finally, the Given-When-Then model can be fully integrated as a Gatling simulation code:

```
import io.gatling.core.Predef._
import io.gatling.http.Predef._

import scala.concurrent.duration._

class FullySpecifiedSimulation extends
Simulation {

 val baseHttpProtocol =
   http.baseUrl("https://computer-database.
gatling.io")

 // When
 val scn = scenario("simplest")
   .exec(
     http("Home")
       .get("/")
   )

 // Given
 setUp(
   scn.inject(
     rampUsersPerSec(1) to 200 during
1.minute,
     constantUsersPerSec(200) during
9.minutes
   )
 ).protocols(baseHttpProtocol) // Then
   .assertions(
     global.requestsPerSec.gte(100),
     global.responseTime.percentile(99).
lt(250),
     global.failedRequests.percent.lte(1)
   )
}
```

The Scenario (**When**) part didn't change (yet), but you will need the Injection Profile (**Given**) at the same place, and a new part, called assertions, as **Then**. What assertions does is computing metrics from within all the simulation, and will fail the "build" if they are under/over the requirements. Using a Maven project, for example, you'll see:

```
[INFO] ------------------------------------
------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------
------------------------
[INFO] Total time:  07:07 min
[INFO] Finished at: 2020-08-
27T20:52:52+02:00
[INFO] ------------------------------------
------------------------
```

And BUILD FAILURE otherwise. This is handy when using a CI tool such as Jenkins, for which we provide a Gatling Jenkins plugin. Using it, you can configure the running of simulations right into your CI, and get notified if your acceptance criteria are failed and by how much.

Note that in the previous example we used the keyword global as a starting point, but you could be as precise as having a single request having no failures and ignore all the others:

```
.assertions(
  details("Checkout").failedRequests.
percent.is(0)
)
```

You can find more examples in our assertions documentation.

Defining a test protocol

If you believe that a specific variable, for example, the network, is the culprit behind the poor performance, you will need a proper testing protocol to test your assumptions. Each variable should be tested against a "witness." If you suspect "something" is the cause, test it with and without, then compare. Make a baseline and make small variations, then test against this.

It boils down to ALWAYS TESTING YOUR ASSUMPTIONS,

Additional tooling will help catch the essentials that Gatling can't. As all metrics given by Gatling are from the point of view of the user, you'll need to make sure you are equipped with system and application monitoring on the other side of the spectrum. System monitoring is also very useful to have on the Gatling side. It will help you see if you are using too many resources on the machine on a huge test with information such as CPU, memory, network connections, and their states, networking issues, etc. There are many popular metrics collection solutions out there, such as Prometheus combined with Node Exporter. These can be used alongside Grafana combined with an appropriate dashboard. Let it run and collect the metrics all the time. Check when running a test on the time window of the test.

Equipped with such tools, the test protocol will boil down to:

1. Ensure system and application monitoring is enabled
2. Run a load test (capacity, for example)
3. Analyze and deduce a limit
4. Tune
5. Try again

Going beyond

Closing your logbook, you realize you weren't that far off succeeding after all. Rather than building a full-fledged load testing Cathedral, you decide to go one small step at a time and understanding that knowledge is key. The sooner you have the information you need, the better.

From now, when everything seems to work fine, you will have the possibility to add complexity to the scenario, approaching more and more the way your users actually perform actions on your application. Your most helpful resources will be the Gatling OSS documentation, along with:

- The Quickstart, so you can learn how to record this user journey without writing much code right at the beginning
- The Advanced Tutorial to go deeper
- The Cheat-Sheet that lists all the Gatling language keywords and their usage

Four Case Studies for Implementing Real-Time APIs

by **Karthik Krishnaswamy** Director | Product Marketing at F5

API calls now make up 83% of all web traffic. The age of the API has arrived, and companies should be well past the point of just having enthusiasm for developing APIs — they need them to survive in digital business. But in the digital era, it's easy for your customers and partners to switch services. Don't like your bank? Opening a new account in another bank is as simple as downloading an app. APIs have made it easy for us to consume services across all industries.

That means competitive advantage is no longer won by simply having the APIs; the key to gaining ground is based on the performance and the reliability of those APIs. According to our research at NGINX, you need to be able to process an API call in 30ms or less in order to deliver real-time experiences.

Here are four case studies of companies that created an API structure capable of delivering the real-time speed needed to win business in this highly competitive landscape.

Automate API management at scale for microservices: Greenfield online bank

Objective: Deliver real-time performance and reliability at scale to customers accessing applications from non-smart mobile devices.

Problem: Microservices-based applications delivering poor

performance and struggling with added latency when processing internal API calls for transactions.

Key takeaway: External services can consist of multiple, internal API calls among microservices; slow-performing API infrastructure can degrade service levels.

Background and challenge

This green-field bank was launched in 2016 with the goal of providing banking services to poor and rural areas in South Africa. In order to compete with the incumbent banks, they decided to build a modern digital banking application from the ground up using containers and microservices to provide digital banking services to unbanked or underbanked households.

They built a distributed architecture based on this microservices reference architecture. This provided the bank with the flexibility needed to deliver new services quickly enough to match the expectations of today's digital consumers, while limiting downtime. The bank chose this reference architecture as it was prescriptive and gave them a roadmap to start small – initially handing ingress traffic to a modest Kubernetes cluster – and grow to hundreds or even thousands of microservices connected via a service mesh.

However, pivoting to a microservices architecture introduced a lot of complexity around both API scalability and increased inter-service communication (east-west traffic), which developers needed to allow microservices to communicate with each other.

This posed a significant challenge to the bank as most of their customers were not accessing their services from smart devices but using non-smart mobile phones. As a result, the bank was not able to rely on customer computing power. Instead, a simple mobile phone would communicate with the bank via SMS, which would kick off a series of internal APIs calls to process the transaction and deliver the result back to the customer via SMS.

How real-time API management helped

Unreliable or slow performance can directly impact or even prevent the adoption of new digital services, making it difficult for a business to maximize the potential of new products and expand its offerings. Thus, it is not only crucial that an API processes calls at acceptable speeds, but it is equally important to have an API infrastructure in place that is able to route traffic to resources correctly, authenticate users, secure APIs, prioritize calls, provide proper bandwidth, and cache API responses.

Most traditional APIM solutions were made to handle traffic between servers in the data center and the client applications accessing those APIs externally (north-south traffic). They also need constant connectivity between the control plane and data plane, which requires using third-party modules, scripts, and local databases. Processing a single request creates significant overhead — and it only gets more complex when dealing with the east-west traffic associated with a distributed application.

Considering that a single transaction or request could require multiple internal API calls, the bank found it extremely difficult to deliver good user experiences to their customers. For example, the bank's existing API management solution was adding anywhere from 100 to 500ms of transaction latency to each call.

The bank had also established CI/CD tooling and frameworks to automate microservices development and deployment. They had a Site Reliability (SRE) team tasked with overseeing the entire system and needed a solution that could integrate easily with their CI/CD pipeline infrastructure.

Deploying an API management solution that decouples the data and control plane introduced less latency and offered high-performance API traffic mediation for both external service and inter-service communication. Runtime connectivity to the control plane was no longer needed

for the data plane to process and route calls, minimizing complexity. As a result, the bank was able to process API calls up to three times faster. In addition, the new API management solution integrated directly with the bank's existing microservices deployment and CI/CD pipeline, as well as offering monitoring for SRE teams to help eliminate human error, downtime, and reduce operational complexity.

Connect microservices API traffic - Leading Asian-Pacific telecommunications provider

Objective: Process 600 million API calls per month across at least 800 different internal APIs.

Problem: The existing API infrastructure was too expensive and slow to use for internal API traffic, resulting in performance degradation.

Key takeaway: Processing internal API calls within the corporate firewall instead of routing it through a cloud outside the corporate network improves performance significantly.

Background and challenge

A leading telecom organization from the Asia-Pacific region embraced an API-first development philosophy as part of their digital transformation initiative. This drove an explosion of internal APIs needed to process 600 million API calls per month across 800 different internal APIs. The current infrastructure was not well-equipped to process this new API traffic, causing a degradation of performance and developer and DevOps team to invest in non-standard solutions.

The telecom company was already using a solution for API management. However, it was better suited for handling north-south traffic from external APIs rather than east-west traffic between internal services. The company decided to seek a new solution that would allow them to reduce latency, while also providing capabilities to empower their DevOps teams with self-service API management.

How real-time API management helped

The telecom organization's existing API management solution relied on deployment models where the API management and gateways were hosted in the public cloud, meaning that traffic must be looped out to the cloud first for every interaction. In this case, sending traffic out to the public cloud was not only costly, it was much slower — adding several seconds of latency.

The telecommunications organization opted to segment external and internal API management, implementing a higher-performing API management solution to manage and deploy internal APIs and that logically sat "behind" the existing deployment on the enterprise perimeter. This allowed them to process internal API calls within the corporate network, rather than being forced to route them out into the public cloud, resulting in a 70% reduction in latency with API calls being processed at 20ms or less.

DevOps teams were also able to easily create, publish, and monitor APIs, helping to increase application and microservices release velocity by integrating API management tasks directly into their CI/CD pipeline. By automating routine tasks using APIs, such as API definition and gateway configuration, the organization was able to achieve significant savings in time and effort.

Process large volume of credit card transactions in real-time - Large US-based credit card company

Objective: Process billions of API calls in real-time — sub 70ms latency.

Problem: The existing API management solution added 500ms of latency for every API call, resulting in direct revenue loss for the company.

Key takeaways: Performance trumps feature richness in API management solutions when revenue is impacted.

Background and challenge

A leading credit card company was struggling with a transaction latency problem. When paying with a credit card, most point-of-

sale (POS) systems will time out after a set limit expires. Cards will need to be run through again, but transactions will automatically fail to avoid any duplicate transactions being charged to the card.

At the same time, the organization was moving to Open Banking standards, which provide API specifications that enable sharing of customer-permissioned data and analytics with third-party developers and firms to build applications and services—increasing the volume of API calls into the hundreds of billions.

As a result, the company started looking for a solution that would be able to manage and scale to handle billions of API calls as fast as possible — the goal was set to sub 70ms latency per API call. This was deemed as the threshold before customer experience would be impacted for POS transactions.

How real-time API management helped

The company's existing API solution was adding 500ms of latency for every API call, causing a nominal amount of transactions to fail. But even a small fraction of billions of transactions is a significant number, especially when they result in a loss of revenue.

It's common for customers to try paying again with another credit card when a transaction fails. If the second card they use is not issued by the same company, it could potentially mean millions in lost dollars—all due to timed-out API calls.

The company pioneered a real-time API reference architecture to ensure API calls were processed in as close to real-time as possible. For example, they deployed clusters of two or more high availability API gateways to improve the reliability and resiliency of their APIs. The company also chose to enable dynamic authentication by pre-provisioning authentication information (using API keys and JSON Web Tokens, or JWT), which made authentication almost instantaneous. In addition, they chose to delegate authorization to the business-logic layer of their backend so that their API gateways were only responsible for handling authentication, resulting in faster response times to calls.

By following these best practices, the company was able to achieve response times that were consistently less than 10ms, exceeding the performance requirements by 85%. This delivered tangible, direct savings—helping to not only recover lost transactions but enabling them to process even more transactions than before. The company concluded that these performance gains were more critical to their business outcomes than some of the additional features the incumbent solution had, which included a richer developer portal, API design tools, and API transformation features of their previous solution.

The improved transaction latency and reliability had the added benefit of helping to create and solidify new revenue streams. As part of their open banking efforts, the company is now able to expose their core transactional engine to ISVs and developers and win more business as a result of the speed and reliability advantages they can demonstrate over their competitors.

Securely process billions of transactions using a single API management solution – US-based financial services company

Objective: A lightweight, cost-effective solution that can process and route API calls for REST APIs, SOAP APIs, and externally accessed services that meets strict federal financial compliance requirements.

Problem: The company had three existing solutions to manage each of the different types of traffic, which required configuring specific gateways multiple times in order to make changes and quickly achieve scale to serve the needs of internal and external consumers.

Key takeaway: High API transaction throughput (calls per second) is essential for rapid adoption

Background and challenge

Recognizing that the future of financial services would rely heavily on software development, the financial services company began to focus on transforming into a technology company in 2014 by investing in RESTful APIs. This involved placing heavy emphasis on engineering and development, empowering developers to create software that was easy to consume in order to deliver great applications to end users.

In addition to their own offerings that served millions of customers, the company also created an internal development exchange platform, which was eventually made available externally to allow them to integrate with business partners and third parties through APIs. There was extreme pressure to be able to deliver performant APIs and an infrastructure that could support a high volume of transactions every day.

Over the years, the company had also acquired a lot of technical debt, including a legacy service bus and appliances to handle SOAP/XML APIs. As a result, the company was using at least three different solutions to manage API traffic, which made it hard to adapt and respond quickly to changing environments and meet market demands. Teams were forced to constantly configure specific gateways — making updates three times rather than once every time a change was needed.

The company wanted to consolidate, but they were also looking for a solution that would allow them to scale and handle billions of API calls while protecting the high-speed developer exchange platform that was the main core of their business. To meet their scale, flexibility, and performance requirements, the company decided it couldn't rely solely on a packaged solution or service. They would need a combination of API infrastructure software and custom-developed API tooling.

How real-time API management helped

The goal for the company, like any modern technology-focused business, was resiliency, high speed, and low overhead for internal customers that want to avoid impacting functionality. However, handling hundreds of thousands of concurrent API calls without performance degradation can be tricky using traditional application design, which relies on a process-per-connection model to handle requests.

Using open source software (OSS) as the main foundation to support their API gateway infrastructure, the company was able to reduce context switching and the load on resources. This allowed them to leverage an asynchronous, event-driven approach that allows multiple requests to be handled by a single worker process — in other words, they were able to scale to support hundreds of thousands of concurrent connections with very little additional overhead.

The company also struggled to manage and configure multiple disparate solutions, which not only consumed resources but also caused a lot of downtime as they had to be restarted during upgrades. The developer and DevOps teams then highly customized the open source beyond its original capability by adding Lua-based modules and scripts, enabling them to standardize on this OSS gateway to handle all types of traffic, helping to eliminate sprawl and complexity. In addition, the company was able to upgrade without downtime or service interruption by starting a new set of worker processes when a new configuration is detected while continuing to route live traffic to the old processes containing the previous configuration. Once the new configuration is tested and ready, the new gateway immediately starts accepting connections and processing traffic based on the new settings.

The company is now able to handle around 360 billion API calls per month — or about 12 billion calls in a single day with peak traffic of 2 million API calls per second.

Real-time APIs require a real-time API solution

The case studies above serve to demonstrate some of the most common ways we are helping organizations develop their API programs. We would love to hear from you about your real-time API needs and experiences with delivering APIs in real-time. What API management challenges are you facing? How are you dealing with scaling your API infrastructure to meet your modernization needs?

Please leave comments below or, better yet, use our open source API assessment tool to measure your API performance. Learn more on GitHub.

TL;DR

- API calls now make up 83% of all web traffic. Competitive advantage is no longer won by simply having APIs; the key to gaining ground is based on the performance and the reliability of those APIs.
- Modern systems can consist of multiple internal API calls among microservices; slow-performing API infrastructure can degrade service levels throughout an organisation's systems.
- Processing internal API calls within the corporate firewall instead of routing it through a cloud outside the corporate network can improve performance significantly.
- Performance trumps feature richness in API management solutions when revenue is correlated with speed. Teams should recognize this and design system applications and infrastructure accordingly.
- Consolidating traffic management solutions within a system can improve the capability to react to changing requirements, rapidly update configuration, and benchmark overall performance.

SPONSORED ARTICLE

NGINX
Part of F5

A Reference Architecture for Real-Time APIs

As companies seek to compete in the digital era, APIs become a critical IT and business resource. Architecting the right underlying infrastructure ensures not only that your APIs are stable and secure, but also that they qualify as real-time APIs, able to process API calls end-to-end within 30 milliseconds (ms).

API architectures are broadly broken up into two components: the data plane, or API gateway, and the control plane, which includes policy and developer portal servers. A real-time API architecture depends mostly on the API gateway, which acts as a proxy to process API traffic. It's the critical link in the performance chain.

API gateways perform a variety of functions including authenticating API calls, routing requests to the right backends, applying rate limits to prevent overburdening your systems, and handling errors and exceptions. Once you have decided to implement real-time APIs, what are the key characteristics of the API gateway architecture? How do you deploy API gateways?

This article addresses these questions, providing a real-time API reference architecture based on our work with NGINX's largest, most demanding customers. We encompass all aspects of the API management solution but go deeper on the API gateway which is responsible for ensuring real-time performance thresholds are met.

The Real-Time API Reference Architecture

Our reference architecture for real-time APIs has six components:

- **API gateway.** A fast, lightweight data-plane component that processes API traffic. This is the most critical component in the real-time architecture.
- **API policy server**. A decoupled server that configures API gateways, as well as supplying API lifecycle management policies.
- **API developer portal**. A decoupled web server that provides documentation for rapid onboarding for developers who use the API.
- **API security service**. A separate web application firewall (WAF) and fraud detection component which provides security beyond the basic security mechanisms built into the API gateway
- **API identity service**. A separate service that sets authentication and authorization policies for identity and access management and integrates with the API gateway and policy servers.
- **DevOps tooling**. A separate set of tools to integrate API management into CI/CD and developer pipelines.

Please read the full-length version of this article here.

Real-Time APIs in the Context of Apache Kafka

by **Robin Moffatt**, Senior Developer Advocate at Confluent

One of the challenges that we have always faced in building applications, and systems as a whole, is how to exchange information between them efficiently whilst retaining the flexibility to modify the interfaces without undue impact elsewhere. The more specific and streamlined an interface, the likelihood that it is so bespoke that to change it would require a complete rewrite. The inverse also holds; generic integration patterns may be adaptable and widely supported, but at the cost of performance.

Events offer a Goldilocks-style approach in which real-time APIs can be used as the foundation for applications which is flexible yet performant; loosely-coupled yet efficient.

Events can be considered as the building blocks of most other data structures. Generally speaking, they record the fact that something has happened and the point in time at which it occurred. An event can capture this information at various levels of detail: from a simple notification to a rich event describing the full state of what has happened.

From events, we can aggregate up to create state—the kind of state that we know and love from its place in RDBMS and NoSQL stores. As well as being the basis for state, events can also be used to asynchronously trigger actions elsewhere when something happens - the whole basis for event-driven architectures. In this way, we can build consumers to match our requirements—both stateless, and stateful with fault tolerance. Producers can opt to maintain state, but are not required to since consumers can rebuild this themselves from the events that are received.

If you think about the business domain in which you work, you can probably think of many examples of events. They can be human-generated interactions, and they can be machine-generated. They may contain a rich payload, or they may be in essence a notification alone. For example:

- Event: `userLogin`

  – Payload: `zbeeble-brox` logged in at `2020-08-17 16:26:39 BST`

- Event: `CarParked`

  – Payload: Car registration `A42 XYZ` parked at `2020-08-17 16:36:27` in space `X42`

- Event: `orderPlaced`

  – Payload: `Robin` ordered `four tins of baked beans` costing a total of `£2.25` at `2020-08-17 16:35:41 BST`

These events can be used to directly trigger actions elsewhere (such as a service that processes orders in response to them being placed), and they can also be used in aggregate to provide information (such as the current number of occupied spaces in a car park and thus which car parks have availability).

So, if events are the bedrock on which we are going to build our applications and services, we need a technology that supports us in the best way to do this—and this is where Apache Kafka® comes in. Kafka is a scalable event streaming platform that provides

- **Pub/Sub**

  – To publish (write) and subscribe to (read) streams of events, including continuous import/export of your data from other systems.

- **Stateful stream processing**

  – To store streams of events durably and reliably for as long as you want.

- **Storage**

  – To process streams of events as they occur or retrospectively.

Kafka is built around the concept of the log. By taking this simple but powerful concept of a distributed, immutable, append-only log, we can capture and store the events that occur in our businesses and systems in a scalable and efficient way. These events can be made available to multiple users on a subscription basis, as well as processed and aggregated further, either for direct use or for storage in other systems such as RDBMS, data lakes, and NoSQL stores.

In the remainder of this article, I will explore the APIs available in Apache Kafka and demonstrate how it can be used in the systems and applications that you are building.

The Producer and Consumer APIs

The great thing about a system like Kafka is that producers and consumers are decoupled, meaning, amongst other things, that we can produce data without needing a consumer in place first (and because of the decoupling we can do so at scale). An event happens, we send it to Kafka—simple as that. All we need to know is the details of the Kafka

cluster, and the topic (a way of organising data in Kafka, kind of like tables in an RDBMS) to which we want to send the event.

There are clients available for Kafka in many different languages. Here's an example of producing an event to Kafka using Go:

```
package main

import (
    “gopkg.in/confluentinc/confluent-kafka-
go.v1/kafka”
)

func main() {

    topic := “test_topic”
    p, _ := kafka.NewProducer(&kafka.
ConfigMap{
        “bootstrap.servers”:
“localhost:9092”})
    defer p.Close()

    p.Produce(&kafka.Message{
        TopicPartition: kafka.
TopicPartition{Topic: &topic,
            Partition: 0},
        Value: []byte(“Hello world”)}, nil)

}
```

Because Kafka stores events durably, it means that they are available as and when we want to consume them, until such time that we age them out (which is configurable per topic).

Having written the event to the Kafka topic, it's now available for one, or more, consumers, to read. Consumers can behave in a traditional pub/sub manner and receive new events as they arrive, as well as opt to arbitrarily re-consume events from a previous point in time as required by the application. This replay functionality of Kafka, thanks to its durable and scalable storage layer, is a huge advantage for many important use cases in practice such as machine learning and A/B testing, where both historical and live data are needed. It's also a hard requirement in regulated industries, where data must be retained for many years to meet legal compliance. Traditional messaging systems like RabbitMQ, ActiveMQ cannot support such requirements.

```
package main

import (
    “fmt”

    “gopkg.in/confluentinc/confluent-kafka-
go.v1/kafka”
)

func main() {

    topic := “test_topic”

    cm := kafka.ConfigMap{
        “bootstrap.servers”:
“localhost:9092”,
        “go.events.channel.enable”: true,
        “group.id”:
“rmoff_01”}

    c, _ := kafka.NewConsumer(&cm)
    defer c.Close()
    c.Subscribe(topic, nil)

    for {
        select {
        case ev := <-c.Events():
            switch ev.(type) {

            case *kafka.Message:
                km := ev.(*kafka.Message)
                fmt.Printf(“ Message ‘%v’
received from topic ‘%v’\n”, string(km.
Value), string(*km.TopicPartition.Topic))
            }
        }
    }

}
```

When a consumer connects to Kafka, it provides a Consumer Group identifier. The consumer group concept enables two pieces of functionality. The first is that Kafka keeps track of the point in the topic to which the consumer has read events, so that when the consumer reconnects it can continue reading from the point that it got to before. The second is that the consuming application may want to scale its reads across multiple instances

of itself, forming a Consumer Group that allows for processing of your data in parallel. Kafka will then allocate events to each consumer within the group based on the topic partitions available, and it will actively manage the group should members subsequently leave or join (e.g., in case one consumer instance crashed).

This means that multiple services can use the same data, without any interdependency between them. The same data can also be routed to datastores elsewhere using the Kafka Connect API which is discussed below.

The Producer and Consumer APIs are available in libraries for Java, C/C++, Go, Python, Node.js, and many more. But what if your application wants to use HTTP instead of the native Kafka protocol? For this, there is a REST Proxy.

Using a REST API with Apache Kafka

Let's say we're writing an application for a device for a smart car park. A payload for the event recording the fact that a car has just occupied a space might look like this:

```
{
    “name”: “NCP Sheffield”,
    “space”: “A42”,
    “occupied”: true
}
```

We could put this event on a Kafka topic, which would also record the time of the event as part of the event's metadata. Producing data to Kafka using the Confluent REST Proxy is a straightforward REST call:

```
curl -X POST \
     -H “Content-Type: application/vnd.
kafka.json.v2+json” \
     -H “Accept: application/vnd.kafka.
v2+json” \
     --data ‘{“records”:[{“value”:{ “name”:
“NCP Sheffield”, “space”: “A42”, “occupied”:
true }}]}’ \
     “http://localhost:8082/topics/carpark”
```

Any application can consume from this topic, using the native Consumer API that we saw above, or by using a REST call. Just like with the native Consumer API, consumers using the REST API are also members of a Consumer Group, which is termed a subscription. Thus with the REST API you have to declare both your consumer and subscription first:

```
curl -X POST -H “Content-Type: application/
vnd.kafka.v2+json” \
      --data ‘{“name”: “rmoff_consumer”,
“format”: “json”, “auto.offset.reset”:
“earliest”}’ \
      http://localhost:8082/consumers/rmoff_
consumer

curl -X POST -H “Content-Type:
application/vnd.kafka.v2+json” --data
‘{“topics”:[“carpark”]}’ \
 http://localhost:8082/consumers/
rmoff_consumer/instances/rmoff_consumer/
subscription
```

Having done this you can then read the events:

```
curl -X GET -H “Accept: application/vnd.
kafka.json.v2+json” \
      http://localhost:8082/consumers/rmoff_
consumer/instances/rmoff_consumer/records
[
    {
        “topic”: “carpark”,
        “key”: null,
        “value”: {
            “name”: “Sheffield NCP”,
            “space”: “A42”,
            “occupied”: true
        },
        “partition”: 0,
        “offset”: 0
    }
]
```

If there are multiple events to receive, then you'll get them within a batch per call, and if your client wants to check for new events, they will need to make the REST call again.

We've seen how we can get data in and out of Kafka topics. But a lot of the time we want to do more than just straight-forward pub/sub. We want to

take a stream of events and look at the bigger picture—of all the cars coming and going, how many spaces are there free **right now**? Or perhaps we'd like to be able to subscribe to a stream of updates for a particular car park only?

Conditional Notifications, Stream Processing, and Materialised Views

To think of Apache Kafka as pub/sub alone is to think of an iPhone as just a device for making and receiving phone calls. I mean, it's not wrong to describe that as one of its capabilities…but it does so much more than just that. Apache Kafka includes stream processing capabilities through the Kafka Streams API. This is a feature-rich Java client library for doing stateful stream processing on data in Kafka at scale and across multiple machines. Widely used at companies such as Walmart, Ticketmaster, and Bloomberg, Kafka Streams also provides the foundations for ksqlDB.

ksqlDB is an event streaming database purpose-built for stream processing applications. It provides a SQL-based API for querying and processing data in Kafka. ksqlDB's many features include filtering, transforming, and joining data from streams and tables in real-time, creating materialised views by aggregating events, and much more.

To work with the data in ksqlDB we first need to declare a schema:

```
CREATE STREAM CARPARK_EVENTS (NAME
VARCHAR,
                              SPACE
VARCHAR,
                              OCCUPIED
BOOLEAN)
                         WITH (KAFKA_
TOPIC='carpark',
                              VALUE_
FORMAT='JSON');
```

ksqlDB is deployed as a clustered application, and this initial declaration work can be done at startup, or directly by the client, as required. With this done, any client can now subscribe to a stream of changes from the original topic but with a filter applied. For example, to get a notification when a space is released at a particular car park they could run:

```
SELECT TIMESTAMPTOSTRING(ROWTIME,'yyyy-MM-
dd HH:mm:ss') AS EVENT_TS,
       SPACE
  FROM CARPARK_EVENTS
 WHERE NAME='Sheffield NCP'
   AND OCCUPIED=false
  EMIT CHANGES;
```

Unlike the SQL queries that you may be used to, this query is a continuous query (denoted by the `EMIT CHANGES` clause). Continuous queries, known as push queries, will continue to return any new matches to the predicate as the events occur, now and in the future, until they are terminated. ksqlDB also supports pull queries (which we explore below), and these behave just like a query against a regular RDBMS, returning values for a lookup at a point in time. ksqlDB thus supports both the worlds of streaming and static state, which in practice most applications will also need to do based on the actions being performed.

ksqlDB includes a comprehensive REST API, the call against which for the above SQL would look like this using `curl`:

```
curl --http2 'http://localhost:8088/query-
stream' \
     --data-raw '{"sql":"SELECT TIMESTAMPTO
STRING(ROWTIME,'\''yyyy-MM-dd HH:mm:ss'\'')
AS EVENT_TS, SPACE FROM CARPARK_EVENTS
WHERE NAME='\''Sheffield NCP'\'' and
OCCUPIED=false EMIT CHANGES;"}'
```

This call results in a streaming response from the server, with a header and then when any matching events from the source topic are received these are sent to the client:

```
{“queryId”:”383894a7-05ee-4ec8-bb3b-c5ad39811539”,”columnNames”:[“EVENT_
TS”,”SPACE”],”columnTypes”:[“STRING”,”STRING”]}
…
[“2020-08-05 16:02:33”,”A42”]
…
…
…
[“2020-08-05 16:07:31”,”D72”]
…
```

We can use ksqlDB to define and populate new streams of data too. By prepending a `SELECT` statement with `CREATE STREAM streamname AS` we can route the output of the continuous query to a Kafka topic. Thus we can use ksqlDB to perform transformations, joins, filtering, and more on the events that we send to Kafka. ksqlDB supports the concept of a table as a first-class object type, and we could use this to enrich the car park events that we're receiving with information about the car park itself:

```
CREATE STREAM CARPARKS AS
    SELECT E.NAME AS NAME, E.SPACE,
           R.LOCATION, R.CAPACITY,
           E.OCCUPIED,
           CASE
               WHEN OCCUPIED=TRUE THEN 1
               ELSE -1
           END AS OCCUPIED_IND
    FROM   CARPARK_EVENTS E
           INNER JOIN
           CARPARK_REFERENCE R
           ON E.NAME = R.NAME;
```

You'll notice we've also used a `CASE` statement to apply logic to the data enabling us to create a running count of available spaces. The above `CREATE STREAM` populates a Kafka topic that looks like this:

```
+---------------+-------+----------+---------------------------+----------+------------
--+
|NAME           |SPACE  |OCCUPIED  |LOCATION                   |CAPACITY  |OCCUPIED_IND
|
+---------------+-------+----------+---------------------------+----------+------------
--+
|Sheffield NCP  |E48    |true      |{LAT=53.4265964, LON=-1.8426|1000      |1
|
|               |       |          |386}                       |          |
|
```

Finally, let's see how we can create a stateful aggregation in ksqlDB and query it from a client. To create the materialised view, you run SQL that includes aggregate functions:

```
CREATE TABLE CARPARK_SPACES AS
    SELECT NAME,
           SUM(OCCUPIED_IND) AS OCCUPIED_SPACES
        FROM CARPARKS
        GROUP BY NAME;
```

This state is maintained across the distributed ksqlDB nodes and can be queried directly using the REST API:

```
curl --http2 'http://localhost:8088/query-stream' \
     --data-raw '{"sql":"SELECT OCCUPIED_SPACES FROM CARPARK_SPACES WHERE
NAME='\''Birmingham NCP'\'';"}'
```

Unlike the streaming response that we saw above, queries against the state (known as "pull queries", as opposed to "push queries") return immediately and then exit:

```
{"queryId":null,"columnNames":["OCCUPIED_SPACES"],"columnTypes":["INTEGER"]}
[30]
```

If the application wants to get the latest figure, they can reissue the query, and the value may or may not have changed

```
curl --http2 'http://localhost:8088/query-stream' \
     --data-raw '{"sql":"SELECT OCCUPIED_SPACES FROM CARPARK_SPACES WHERE
NAME='\''Birmingham NCP'\'';"}'
{"queryId":null,"columnNames":["OCCUPIED_SPACES"],"columnTypes":["INTEGER"]}
[29]
```

There is also a Java client for ksqlDB, and community-authored Python and Go clients.

Integration with other systems

One of the benefits of using Apache Kafka as a highly-scalable and persistent broker for your asynchronous messaging is that the same data you exchange between your applications can also drive stream processing (as we saw above) and also be fed directly into dependent systems.

Continuing from the example of an application that sends an event every time a car parks or leaves a space, it's likely that we'll want to use this information elsewhere, such as:

- analytics to look at parking behaviours and trends
- machine learning to predict capacity requirements
- data feeds to third-party vendors

Using Apache Kafka's Connect API you can define streaming integrations with systems both in and out of Kafka. For example, to stream the data from Kafka to S3 in real-time you could run:

```
curl -i -X PUT -H "Accept:application/json" \
    -H  "Content-Type:application/json" http://localhost:8083/connectors/sink-s3/config \
    -d ' {
        "connector.class": "io.confluent.connect.s3.S3SinkConnector",
        "topics": "carpark",
        "s3.bucket.name": "rmoff-carparks",
        "s3.region": "us-west-2",
        "flush.size": "1024",
        "storage.class": "io.confluent.connect.s3.storage.S3Storage",
        "format.class": "io.confluent.connect.s3.format.json.JsonFormat"
```

```
        }’
```

Now the same data that is driving the notifications to your application, and building the state that your application can query directly, is also streaming to S3. Each use is decoupled from the other. If we subsequently want to stream the same data to another target such as Snowflake, we just add another Kafka Connect configuration; the other consumers are entirely unaffected. Kafka Connect can also stream data into Kafka. For example, the CARPARK_REFERENCE table that we use in ksqlDB above could be streamed using change data capture (CDC) from a database that acts as the system of record for this data.

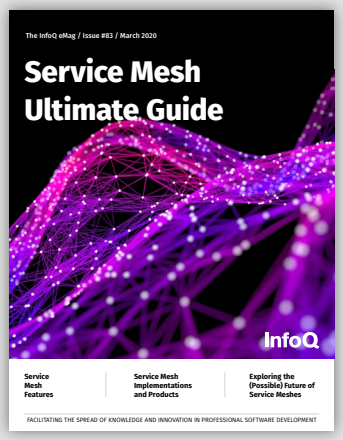
Conclusion

Apache Kafka offers a scalable event streaming platform with which you can build applications around the powerful concept of events. By using events as the basis for connecting your applications and services, you benefit in many ways, including loose coupling, service autonomy, elasticity, flexible evolvability, and resilience.

You can use the APIs of Kafka and its surrounding ecosystem, including ksqlDB, for both subscription-based consumption as well as key/value lookups against materialised views, without the need for additional data stores. The APIs are available as native clients as well as over REST.

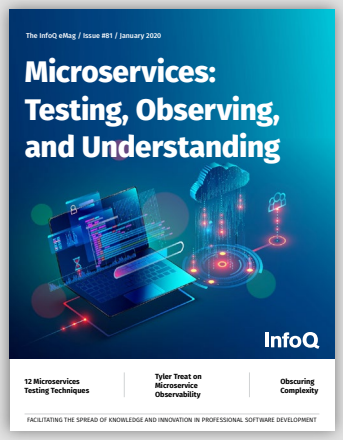
To learn more about Apache Kafka visit developer.confluent.io. Confluent Platform is a distribution of Apache Kafka that includes all the components discussed in this article. It's available on-premises or as a managed service called Confluent Cloud. You can find the code samples for this article and a Docker Compose to run it yourself on GitHub. If you would like to learn more about building event-driven systems around Kafka, then be sure to read Ben Stopford's excellent book Designing Event-Driven Systems.

Curious about previous issues?



This eMag aims to answer pertinent questions for software architects and technical leaders, such as: what is a service mesh?, do I need a service mesh?, and how do I evaluate the different service mesh offerings?

To tame complexity and its effects, organizations need a structured, multi-pronged, human-focused approach, that: makes operations work sustainable, centers decisions around customer experience, uses continuous testing, and includes chaos engineering and system observability. In this eMag, we cover all of these topics to help you tame the complexity in your system.



This eMag takes a deep dive into the techniques and culture changes required to successfully test, observe, and understand microservices.

InfoQ