API and Microservice Management
Technical Whitepaper Part 1

Abstract

API management and microservice management have very different goals but, when understood, can be strong allies. By understanding the differences and some of the key use-cases, one can better understand not just where the lines are, but how they complement each other.

This whitepaper describes a technical strategy for a deeper integration between API Connect and the Istio Service Mesh. This whitepaper is split into three parts: Context, Journey, and our Perspective.

Key Words:

API Connect: IBM's market leading API management solution. A complete, modern and intuitive API lifecycle platform to create, securely expose and manage APIs across clouds to power digital applications.

Istio: An open platform that provides a uniform way to connect, manage, and secure microservices. Istio supports managing traffic flows between microservices, enforcing access policies, and aggregating telemetry data, all without requiring changes to the microservice code.

We begin by discussing the differences and the complementary aspects between API management and microservice management, and how we can further build on these unique features to deliver value to our customers.

The Context

To understand where we are going, we first need to step back and understand the context upon which our future vision will be built. There are distinctive capabilities that microservice and API management excel at. When these capabilities are brought together in the right way, the whole is truly greater than the sum of its parts.

Differences between API management and microservice management

Simply put, API management is focused on how an API is shared and lives through a lifecycle. The relationship between the API creator and the API consumer is a key point that cannot be overlooked - this is a crucial difference between microservice management and API management. The definition of consumer can differ by the use-case, but could be as simple as across different teams in smaller companies, to crossing lines of business (LOBs) or even company bounds. In this paper, an API creator is the developer or squad that creates an API, or the underlying services that the API calls.

The further the distance between the API's creator and the API's consumer, the greater the need for API management. Again, distance can be a construct of either actual space or distance in an organization. If you sit next to the API's creator, you can ask simple questions to understand how to utilize and access the API. If you cannot communicate directly with the creator, you need a standardized way to describe how to use the API. The needs and risks are very different within a microservice mesh, where the consumer is another microservice, and outside, where other apps, orgs, and even companies are leveraging an API. Let's dig into the differences between API and microservice management's view of security, management, socialization, and analytics.

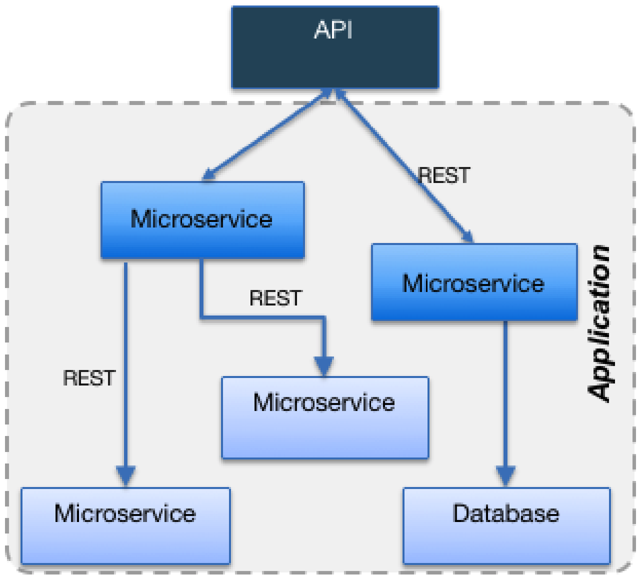


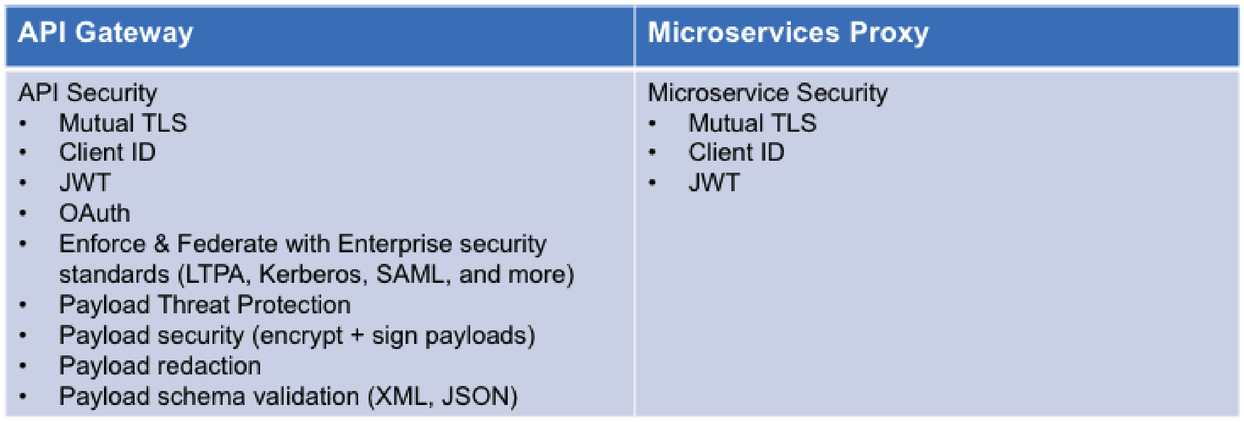
Figure 1: APIs expose a subset of microservices across business boundaries

Security

API security centers on inter-app communication (across firewalls), where microservice security is focused on intra-mesh security. The business risk of exposing backend information is at times so high that modernization efforts are halted. In some industries, like finance and healthcare, the costs of a potential breach can cripple a company's ability to make useful apps for their customers. This risk is mitigated by ensuring every API call goes through a "hardened" API Gateway. There are many industry standards for validating requests, including the use of an API Key and Secret (to validate an API consumer and her applications) and user-level authentication schemes like OAUTH or JWT (ensuring the App *user* is who they say they are). Additionally, payloads can be inspected for threats and structure prior leaving the API Gateway.

Microservices, on the other hand, are identified based on the identity of the microservice, which is usually a DNS hostname. The most common approach to securing microservices is mutual TLS, where certificates can be used to provide the microservice identity.

You should minimize the security complexity between microservices, because it will impact the overall performance and scalability of your application. The best practice is to use a security model which can identify the microservice without requiring an external outbound call to a security policy server. Good examples of securing microservices are those based on client Id, JSON Web Tokens (JWT), or mutual TLS, where validation can be done in-place.



API Gateway	Microservices Proxy
API Security	Microservice Security
• Mutual TLS	• Mutual TLS
• Client ID	• Client ID
• JWT	• JWT
• OAuth	
• Enforce & Federate with Enterprise security standards (LTPA, Kerberos, SAML, and more)	
• Payload Threat Protection	
• Payload security (encrypt + sign payloads)	
• Payload redaction	
• Payload schema validation (XML, JSON)	

Figure 2: Security requirements between an API Gateway and microservice proxy management

The word "management" has many meanings. API management focuses on how the API can be changed and consumed over time, and how subscriptions to the API are controlled. It focuses on making consumption as risk free as possible. For microservice management, it is all about how microservices interact with one another and change with each other over time.

Let's examine the concept of change and the concept of time and the impacts it can have on an API consumer. When an API consumer embeds an API into their app, they do so with a basic understanding of how it is supposed to work. Given an input, a certain output is expected. This "contract" is essential for how the App works going forward.

Now, let's say that a fundamental change is happening to the API: the API creator wants to change the underlying data model for the output. If they do this without communication to the API consumer, they will break Apps. End users will have a negative experience. The API takes on a lifecycle, just like any App. Change needs to be communicated, and the old method needs to be deprecated *prior* to total retirement. There needs to be a process and a timeline for transition, if the creator and consumer relationship is going to be built on trust.

Traffic management (Rate Limiting)

There are fundamental differences between API and microservice traffic management. API Traffic management is focused on how an API is consumed, whereas microservice traffic management is about how microservices interact with each other with a focus on protecting those assets.   Let's start with the API use-case.

The focus of API traffic management is on the consumer. Some of the limits are put in place as broad protections of backend systems, but the actual rate-limit is more of a business rule. For example, giving an App Developer a rate limit of 1,000 API calls per day reduces the risk that they will be a bad actor, so approval can be given more easily. Again, it is about reducing the barriers to consuming APIs across organizational (management) boundaries.

Managing traffic between microservices (such as request routing and rate limiting) have traditionally been performed at the API/application-level. The shift towards cloud/infrastructure platforms have given rise to enhanced traffic management functionality built into the platform, creating a clear boundary between what is delivered between applications and the infrastructure. These platforms standardize the runtime operations using a declarative approach, so you can write policies to enforce runtime behavior without any code.

For example, let's explore the circuit breaker pattern. This pattern helps prevent failure for your entire application when a single service or component is unresponsive. Netflix Hystrix is a popular library used within Java applications to provide circuit breaker functionality. The challenge with using a shared library is that it gets embedded in code and becomes difficult to manage when changes need to be made; instead, using an out-of-process proxy (i.e., sidecar) allows your microservice to add circuit breaker capability without modifying your application. You should design your API and applications to *embrace circuit breaker functions* delivered as part of a microservice management platform.

Rate limiting is a critical part of any API solution; it is also a capability that is needed within a microservice management platform. This leads to a question -- where is the right place to use this capability? The answer is at *both* the API and microservice management platform, but for different reasons. API rate limiting requirements are different than microservices. For example, API rate limiting is often performed based on a consumer with business context (e.g., more than 10 tps is charged and less than 10 tps is free) and can involve a sophisticated set of runtime criteria, such as total API calls per month. In contrast, microservices rate limiting is based on an acceptable runtime call volume between two microservices.

It's important that APIs and applications be designed to take advantage of and embrace what the platform provides when it makes sense -- saving development costs, and enabling you to build a more highly-available and resilient application.

Subscription management

How you manage subscribers is an important differentiator between API and microservice management. Subscription management is another pure attribute and strength of API management. Although microservice management utilizes roles and allows for limited access, few organizations want to take on the risk of adding external app developers to their infrastructure. Just like the security frameworks above, the greater the privacy needs for backend information, the greater the governance for who can access the endpoints.

This can range from internal risk-management needs to regulatory laws. From European privacy laws to reducing risk of breach, an organization must employ business rules for allowing access and monitoring use and have auditable logs of API calls and actions taken inside of the API management platform. Additionally, how organizations interact with each other can vary greatly as well. One vertical (for example, Marketing) might have very different business rules than another vertical (like Operations). Even if they are on the same microservice management mesh, they need the ability to set rules for who can access what, and when. Innovation will be slowed and halted unless these considerations are made upfront with the solution.

Socialization

Another important attribute of API management is how easily an organization can socialize its APIs to internal and (if they choose) external developers. This is accomplished with a self-service developer portal. The ability for a developer to quickly subscribe, learn, and understand an API will ensure it can be utilized more quickly. This understanding is accomplished through the API documentation. A developer needs to know the API endpoint, what method to use, what should be sent (and in what structure), and what is expected to be returned. Everything from error handling to payload size needs to be accounted for. The quicker the adoption loop, the faster APIs will be utilized (and *more* of the APIs will be used). This is the strength of API management. Microservice Manangement does not have a socialization strategy. The mesh owner can give access to the mesh, but there are several governance and risk issues that arise.

Beyond socialization, many are beginning to look at APIs as their products and monetizing them. Monetization strategies require quick discovery, easy subscription, tracking, and billing -- all of which API management provides in spades.

Analytics

Infrastructure administrators will unanimously tell you that they can solve problems faster if they have the right level of runtime visibility into an application. In traditional (monolithic) applications, there was often a single application or service log file that would provide information about the runtime. Now that (formerly monolithic) applications are being designed as a set of microservices, the number of overall components increase and the ability to understand their behaviours and interaction becomes even more important.

Analytics becomes a key component of the operational success of any microservices-based application. Fortunately, cloud/container platforms have been designed with operational efficiency as a top priority. For a microservice management platform, they provide centralized logging, graphical dashboards, and end-to-end transaction tracing (i.e., distributed tracing) as a function of the platform. Their focus is to provide **operational visibility** within the microservices.

API management platforms also provide a set of analytics capabilities. Their main focus is on providing the API product owner "business-user" detail around the **success of their APIs**. This includes dashboards showing which API product is popular and the list of API consumers who are the heaviest users of the platform. The analytics data captured within an API management platform could potentially be used to provide operational visibility (for the API) but its value is smaller when compared to the visibility available within a microservice management platform.

For example, if you want to troubleshoot an issue with your application, the analytics functionality built into the platform will provide you a comprehensive view of the interaction of all the microservices, including a trace of the transaction and the context within each hop. This is the best place to look to troubleshoot any issues. While API analytics could potentially be used to troubleshoot operational issues, it's not the best choice when you already have an analytics component available as part of the runtime, which provides the complete end-to-end picture. It is important to understand the best tool for the job so you can quickly troubleshoot issues and monitor your environment in real-time.

Recap:

The lines around API management and microservice management are well-drawn:

1. API Security is concerned with external access to a resource. The greater the organizational or physical distance between the API Creator and the Consumer, the greater the need for API management.
2. Communicating changes to an API requires a view of consumption lifecycle management. Many organizations want to create new ecosystems around their APIs and to do so, they need to develop trust that their APIs are going go through similar lifecycles as apps: Staging, Published, Replacement (non-breaking), Deprecation (if breaking), and finally Retirement.
3. There is difference between burst-limiting and rate-limiting. Although burst-limiting can be handled by microservice management, Rate-limiting is a business construct that can span meshes, backends, and even companies. Let microservice management protect the backend (where applicable) and have API management focus on the business goals of the API.
4. Adding API subscribers is not as simple as just clicking "add" or creating a key. Governance and internal business rules (e.g., who can add who, who can make changes to an API, who can publish or retire, who can approve API consumers) are rules that microservice management should not answer for external users.
5. The ability to allow developers, either internal or external, to self-subscribe to APIs, get updates, and understand how APIs are used is a critical part of an API strategy.
6. The analytics needs for microservice developers and the infrastructure team and API managers are different. API Analytics are looking at the business case -- who is using what, how is their experience, how can you make it better. Microservice analytics focus on performance of particular microservices. In other words, the "consumer" is explicit for APIs but implicit for microservice management.

The Journey of API Connect and Istio

Evolution of the Service Gateway from centralized to decentralized

Looking back to WebServices and the use of Service Gateways and ESBs, it was very common to see them implemented in a centralized manner (with "centralized" meaning homogeneous clusters running the same software versions and configurations, located centrally in a topology, and operated by a dedicated team independent from the application teams or lines of business).

There were many downsides to this architecture:

- On-ramping new LOBs or applications required a more rigorous testing and release process to minimize the impacts to the existing workload.
- Self-service was likely impossible and each application team was competing for the attention of the team managing this centralized infrastructure

Fast-forward to today with the API Gateway and API Connect where we recommend a decentralized and self-service approach. With this philosophy in mind we don't expect the API Gateway to just sit at the edge of the service mesh, but instead be sprinkled throughout the mesh and facilitate inter-mesh communication.

Now, this does not mean that the API Gateway should be positioned between every microservice, since this would likely be overkill for most use cases and impede the performance and scalability of the greater application. The onus is on the enterprise to define the right granularity of boundaries where OAuth and API-level security are required. Furthermore, we also envision that the API Gateway be a first-class citizen of the mesh and to take full advantage of all the features that Istio brings in a coherent manner.

To achieve this, our DataPower API Gateway is engineered to be compatible with Istio Sidecar injection technique, giving the mesh operator the ability to route traffic to the API Gateway when necessary and route around it when not, but more importantly, to have direct access to all service mesh concepts like canary testing, circuit breaking, and so on.

A little about Istio

Some Background about a key component of Istio: the mixer and pilot

Mixer

One of the core components of Istio is the mixer.

Mixer provides a generic intermediation layer between application code and infrastructure back-ends. Its design moves policy decisions out of the app layer and into configuration, under operator control. Instead of having application code integrate with specific backends, the app code instead does a simple integration with Mixer, and Mixer then takes responsibility for interfacing with the backend systems. Mixer is designed to change the boundaries between layers to reduce systemic complexity, eliminating policy logic from service code and giving control to operators instead.

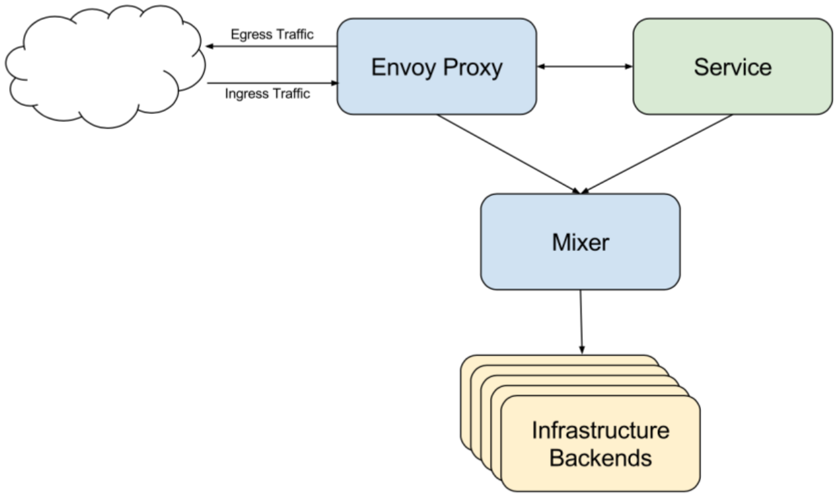


Figure 3: Traffic flow using the core Istio components: The Proxy and the Mixer

Mixer provides three core features:

- **Precondition Checking**. Enables callers to verify many preconditions before responding to an incoming request from a service consumer. Preconditions can include whether the service consumer is properly authenticated, is on the service's whitelist, passes ACL checks, and more.
- **Quota Management**. Enables services to allocate and free quotas on several dimensions. Quotas are used as a relatively simple resource management tool to provide some fairness between service consumers when contending for limited resources. Rate limits are examples of quotas.
- **Telemetry Reporting**. Enables services to report logging and monitoring. In the future, it will also enable tracing and billing streams intended for both the service operator as well as for service consumers.

These mechanisms are applied based on a set of attributes that are materialized for every request into Mixer. Within Istio, the attributes are generated by a sidecar proxy per request, but with the addition of API Connect, attributes are generated using a combination of the Istio sidecar and the API Connect API Gateway.

Pilot

Pilot is the core component used for traffic management in Istio, and it manages and configures all the Istio sidecar (Envoy) instances deployed in a particular Istio service mesh. It lets you specify what rules you want to use to route traffic between microservices, and configure failure recovery features such as timeouts, retries and circuit breakers.

It also maintains a canonical model of all the services in the mesh and uses this to let Istio know about the other instances in the mesh via its discovery service. Each Istio sidecar instance maintains load balancing information based on the information it gets from Pilot and periodic health-checks of other instances in its load-balancing pool, allowing it to intelligently distribute traffic between destination instances while following its specified routing rules.

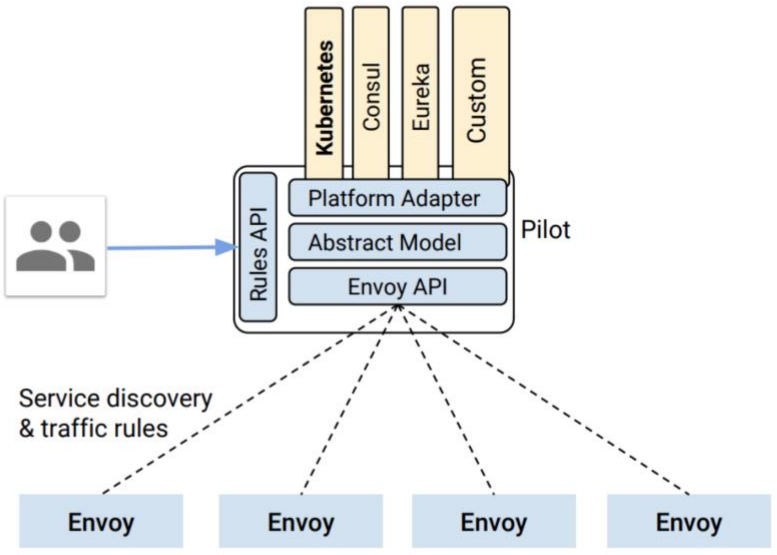


Figure 4: Pilot communicates the traffic rules with the side-car envoy proxies

Conclusion:

Istio improvements with APIm added:

- Ability to share quickly
- Leverage dev portal
- Leverage common language (REST/Open API Spec)
- Be able to try backend quickly in an App
- Leverage subscription management
- Leverage user management
- Users don't need to know backend systems (REST based)
- Quickly add Istio to API Strategy without losing Istio strengths
- Governance is added (global policies, etc)
- Ability to add capabilities to the App Lifecycle, by easily directing “Dev”

APIm improvements with Istio added

- Ability to create new API plans that prioritize users
- Ability to dynamically route API traffic based on plans
- Ability to dynamically route API traffic based on App Lifecycle
- Leverage Istio's ops friendly deployment models (A/B testing, etc)

API and Microservice management each have strengths and weaknesses. By combining the two capabilities, the sum can truly be more powerful and capable than the parts. Istio is on the cutting edge of microservice management with its deep integration with Kubernetes and containers and API Connect is on the cutting edge of API management with its new API Gateway and cloud native architecture.   Contact a sales rep or check out our blogs for the latest and greatest demos and integrations as we move forward in making a true integration.

Part 2 is coming soon and will include how API Connect will integrate with Istio.

Contributors:

Ozair Sheikh, Offering Management, IBM
Tony Ffrench, Gateway Architect
Cesar Botti, APIC Developer, IBM
Chad Petty, APIC Developer, IBM
Jim Laredo, Distinguished Engineer, IBM
Robert Thelen, Offering Management, IBM