API management

Overview

API management is the process of creating and publishing web application programming interfaces (APIs), enforcing their usage policies, controlling access, nurturing the subscriber community, collecting and analyzing usage statistics, and reporting on performance. API Management components provide mechanisms and tools to support developer and subscriber community (Wikipedia - API Management)

Why API management?

A single point of entry for all connected systems and services CMS Wire - Why API Management. See the below informaation for the featres of API management.

Questions

1. OpenAPI : How does OpenAPI work within the API Management ecosystem?
    - I assume the API Management solution is the service catalog for APIs (e.g, the branded API catalog with API specfication, documentation, code samples, etc)

Gartner References

- Gartner - Successfully Implement API Management
- Gartner API Stategy Maturity
- Gartner Ensure API Management includes Cloud and Microservices
- Gartner Magic Quadrant - Full Lifecycle API Management

API Management vs Service Message

- Cloud Native Foundation - API Management vs Service Mesh
- CNF - Cheat Sheat API-M vs Service Mesh Cheat Sheet

History

1. **2003**: EIP : Enterprise Integration Patterns (EIP - 2003). Apache CAMel and Apache ServiceMix (ESB). Many vendors have products in the ESB space (Oracle Fusion / Oracle SOA Suite, IBM Websphere, …). The SOA timeframe (Service Oriented Architecture) which replaced the 3-tier architecture. Note, EIP has many patterns include "data flow" which ESB's provided and more recently products like Apache Nifi
2. **2010**: API Management : Came around 2010 (I'm guessing before microservices and containers). API Management (and **API Gateways**) - runtime features (connect, secure and govern API traffice), plus, non-functionals (creating, testing, documenting, monetizing(billing), monitoring, user personas, ..)
3. **2017**: Service Mesh : A pattern for monoliths or microservices on any platform (VMs, containers, kubernetes). APIM has a role in service meshs (high similarity in functionality). With the service mesh pattern, we are outsourcing the network management of any inbound or outbound request made by any service (not just the ones that we build but also third-party ones that we deploy) to an out-of-process application (the proxy) that will manage every inbound and outbound network request for us, and because it

lives outside of the service, it is by default portable and agnostic in order to support any service written in any language or framework. CNF Difference Between Service Mesh and API Gateway. See the cheat sheet "Most likely, the organization will have both of these use cases, and therefore an API gateway and service mesh will be used simultaneously."

Vendors

Vendors include Mulesoft, Azura API Management, Axway, Mulesoft…

- Amazon : [Amazon API] Gateway(https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html)
- Google : APIGee
- Google API Endpoints
- Microsoft : API Management

Amazon API Gateway

Amazon API Gateway

Features:

- support stateful and stateless - APIs
- flexible authentication mechanisms
- developer portal for publishing - APIs
- Canary Release (for safely rolling - out API)
- Logging for API usage and changes - (AWS Cloudtrail)
- CloudWatch : Logging
- Cloud Formation : template driven - API creation.
- Firewall : protection against - common threats (Web Application Firewall)

Accessing:

- Management console
- SDK
- CLI

Azura APIM

[Microsoft APIM - 2017[(https://docs.microsoft.com/en-us/azure/api-management/api-management-key-concepts)
Features:

- API keys
- DOS Attack Throttling, Advanced Security - Policies - JWT Token Validatoin

Benefits:

- Secure Mobile Infrastructure : gate access - with API keys, prevent DOS attackes, - advanced security with JWT token - validation.
- Partner Integration Ease: Fast partner - onboarding with developer portal, buidling - API facade to decouple internal - implementations that are not released.
- Internal API Program: Centralized location - for communication about API availability - and changes, gate access based on - organization accounts, foundation of - secure channel from API gateway to backend.

Components

- API Gateway: accepts API calls, routes to backend. Verifies API keys, JWT toekns, certificates, and other credentials, enforce policy (quotas, rate limits), transform API, cache resonses, logs for analytics (metadata)
- Admin Portal (Azure Portal)- define API schema, package APIs into products, policies (quotas, transformations, ..), analytics, manage users
- Developer Portal : read API documentation, try API through console, create account, get API keys, view usage analystics

See Microsoft's article on

- The API Gateway Pattern versus the Direct client-to-microservices communication.
- Azure Application Gateway - 2020 - Microsoft is not too clear in documenting the difference between their API-M and their Application Gateway Microsoft Blog Q&A

Google API Endpoints

Google API Endpoints features:
API Proxy; why use it:

- manage your APIs
- protect your APIs
- Fast
- World Class monitoring

Features:

- API keys
- user authenitcation
- Automated deployment
- Integration (Google Cloud Endpoints Framework web framework for AppEngine or add OpenAPI specfication)
  - API management via Google Estensible Service Proxy - ESP. Supports both gRPC and [OpenAPI]() endpoints.
- Logging and monitoring

Wikipedia - API management

Wikipedia API Management
Features:

- gateway: API front-end, receives requests, enforces throtting & security policies, passess requests to the back-end, and then passes response back to requestor
  - transformation engine: orchestrate and modify request and responses on the fly
  - collect analytics data
  - provide caching
  - authentication, authorization, security, audit, regulatory compliance
  - "reverse proxy"
- publishing tools: tools to define APIs, (e.g. OpenAPIor RAML), generate API documentation, govern API usage via access/usage policies, *test* and *debug* API (including security testing and automated test generation/test suites), deploy APIs to production/staging/test, coordinate API lifecycle
- developer API store: community site (usually branded by the API provider) - documentation, tutorials, sample code, SDK, interactive API console, …
- reporting and analytics: monitor API usage and load (hits, transactions, data objects returned, compute time, ..)

- monetization : billing for use

API Gateway

A reverse proxy (e.g. nginx) with features:

- load balancing
- health checks
- API versioning
- routing
- request authentication & authorization
- data transformation, anlaytics, logging
- SSL termination

Vendors Kong, Tyk, AWS API Gateway, Azura API Management, Google Cloud Endpoings

Benefits (from Gateway vs Mesh vs MQ)

- powerful
- low complexity
- understood by IT professionals (web-services)
- layer of defense on public internet
- offload repetitive tasks (user authentication and data validation)

Cons:

- Centralized, horizontally scalable (but central registration for new API and configuration)
- single-team maintenance (centralized control)
- [see Service Mesh](# Service Mesh)

Microgateway

- IBM Strongloop: https://strongloop.com/projects/#mg. IBM OpenAPI-to-GraphQL
- APIGee Microgateway: https://github.com/apigee-internal/microgateway Official Docs Apigee Microgateway

Umbrella / API Umbrella

API Umbrella

- API keys: Track API usage and control access to your APIs with API keys. API Umbrella provides an API key signup form that an be embedded on any website, or internal APIs you can leverage to create API keys programmatically.

- Rate Limiting: Control how many requests each user can make to your APIs to prevent abuse or define usage tiers. Flexible rate limits can be defined ranging from per-second to per-day limits. Different limits can be defined for different APIs or for specific users.

- Analytics: Understand how your API is being used with rich analytics about API requests. View high level summary data, or drill down into the specifics with a flexible analytics querying interface in the admin tool.

- Caching: API Umbrella integrates a standard HTTP caching layer in front of your APIs. Accelerating your APIs and offloading work from your API servers is as easy as setting standard Cache-Control headers.

- Unifying APIs: Provide a single, public entry point to all your APIs and microservices regardless of where your APIs might live behind the scenes or how many APIs you might have. API Umbrella can define how your public endpoints get routed to your API servers.

Other features

- Consumer API Key : If you have multiple APIs, API Umbrella can simplify access for API consumers, with a single API key that can be used across different APIs. And by shifting common functionality, like API keys, rate limiting, and analytics outside of any individual API, API producers don't need to implement any of those details over and over again.

- Admin Web interface: An administrative web tool is available to manage all aspects of API Umbrella, including API routing configuration, user management, and viewing analytics.

- Admin REST APIs : programatically administer the system

- Multitenancy

Netflix zuul

- Netlflix Zuul Github
- Netflix Zuul Medium
- DZone Netflix Zuul - 2016

Microservice architecture

Microservices benefits

Main benefits (rationale for microservices)
Rapid Development: Develop and deploy single services independently. Development & operations focus on the service's repsonsiblity (not the entire systems or business application)
Scalability: Using kubernetes each service can scale independently without impacting other services.

Other major benefits:

- Flexible: each service has flexibility (can use its own language, framework - polygot)
- Independent: services can have their own development cycle, delivery dates and priorities
- Full-Stack Teams: development of microservices suited for cross-functional teams
- Fault Isolation: improved fault isolation
- Stack Flexibility: eliminates long-term committment to a technology stack (debatable)

To use microservices effectively you must adopt and embrace DevOps:

- use Automation, that brings:
- reduce costs and effort
- brings operational efficiency

Microservice Challenges:

There are challenges to using Microservices. These can be overcome with experience.

- resliency : many instances of a given microservice (10's, 100's, 1000's) any of which might fail
- load balancing / auto-scaling : with many endpoints able to fulfil a request routing and scaling get more complicated (effective routing and scaling can save costs in large deployments)
- service discovery : finding a service endpoint and establishing a communication channel gets more complicated with more endpoints
- tracing and monitoring : a single transaction might execute on many microservices. Observing/tracing a transaction is more complicated. When a service fails, identify the cause is difficult.
- versioning : updating an API while allowing older versions to be available
- inter-service communication : A microservice must use APIs to fetch data (versus simple service queries)
- testing microservice dependencies without a tool

Many of the above challenges relate to the distributed nature introduced by the use of microservices (many endpoints involved in a transaction, and many endpoints providing the service). Microservices generally lead to more Ops (operations) overhead. In the DevOps model this is sustainable as the same team providing development provides the operations (**DevOps**)

BMC Microservices - Challenges to Avoid

Design
: Increased complexity with microservices. Microservices create a distributed system, and the challenges of a distributed system must be understood and planned for. Design considerations on the size of microservices, optimal boundaries, connection points, and a framework to integrate services. Each microservice should define its resonsibilities.

Security
: Data in microservices is distributed. It can be hard to maintain and comply with the CIA (confidentiality, integrity and availability) and **privacy** of user data. For example, respecting the authentication and authroisation model for a complex access model across all microservices can be difficult. Designing, and configuring access controls and administering secured authentication to individual microservices is a technical challenge. This also increases the attack surface for the people, processess and technology responsible for securing access.

Testing
: The standalone nature of microservices increases testing complexity. One for testing a microservice without access to the other services (tooling requirement to test), and secondly for testing complex interactions between services (testing integration services, interdependencies, etc).

Increased Operational Compliexity
: Each microservice can decide to use and manage it (*not necessarily true*)

- traditional monitoring: A request that traverses several micorservices can be a convulted/complex path. Debugging issues without proper observability tools (and upfront design to provide observability) is a challenge.
- scalability: Successfully scaling microservices is a skill and a challenge
- optimizing and scaling require coordination: A sudden spike in usage of a business application may required coordinated scaling of multiple microservices.
- fault tolerance for all services: One microservice failure can impact an entire system. Application avaibility can be reduced by poor fault-tolerance of underlying framework and microservices.

Communication
: The microservices must be able to communicate with each other (discover and communicate). This adds latency and reduced speed/response to transactions on applications.