GlobalLogic

WHITE PAPER

Microservice Architecture: API Gateway Considerations

Sanjay Gadge, Principal Architect
Vijaya Kotwani, Senior Engineer

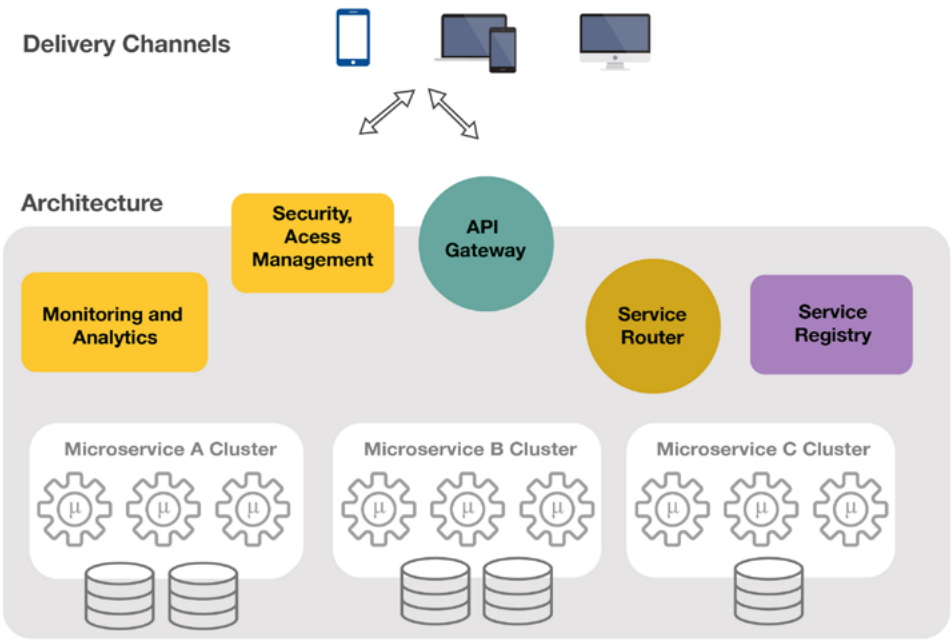
Table of Contents

Introduction

In today's extremely competitive business environment, customers are more demanding than ever and will abandon a business that is too slow to respond. This has put an onus on IT to deliver solutions that provide a holistic and uniform experience to the customer, across all business channels. Microservice architecture has the potential to address this business challenge; it is all about achieving speed and safety at scale and it provides the flexibility to pick and choose technology for implementing a solution. This approach positions IT as a business partner rather than in a traditional support role.

Microservice architecture consists of a suite of independently deployable, small, modular, and compassable (composable) services. Each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal. It aligns with the business to deal with changes in agile fashion, matches business changes with agile response, and delivers solutions in a decentralized manner. In addition to modular services, the API gateway and other elements are integral parts of microservice architecture (see figure below).

Figure 1.
In addition to modular services, the API gateway and other elements are integral parts of microservice architecture.



The What and Why of API Gateways

Basically, the API Gateway is a reverse proxy to microservices and acts as a single-entry point into the system. It is similar to a Facade pattern from object-oriented design and similar to the notion of an "Anti-Corruption Layer" in Domain Driven Design. It makes the processes of API design, implementation, and management considerably simpler and more consistent. The gateway helps to address some of the key concerns, including:

- How to deal with features such as security, throttling, caching and monitoring at one place
- How to avoid chatty communication between clients and microservices
- How to satisfy the needs of heterogeneous clients
- How to route requests to backend microservices
- How to discover working microservice instances
- How to discover when a microservice instance is not running

With a single-entry point into the system, it becomes easy (and manageable) to enforce runtime governance such as common security requirements, common design decisions (e.g. every consumer of the service should have X-Correlation-ID header), and real-time policies such as monitoring, auditing and measuring API usage, and throttling. The gateway abstracts microservices from their consumers, which provides flexibility to freely add or remove microservice instances to adapt to the load/demand of microservices. There may be an instance when different consumers of the service require particular data and/or have it in a special format. For example:

- An e-commerce mobile app shows product information and availability of product on the product details screen, but the desktop website version of the same e-commerce site shows recommendations, in addition to product information and availability.

- There are mobile apps where one understands XML payload and others understand JSON.

The API Gateway is the best place to address these transformation requirements, which can be accomplished by providing application-specific APIs for the same business feature at the gateway level. A one-size-fits-all approach would make it hard to extend functionality, as the degree of diversity increases.

The gateway also helps by recording data for analysis and auditing purposes, load balancing, caching, and static response handling. The diagram below shows how the gateway typically fits into the overall microservice architecture.

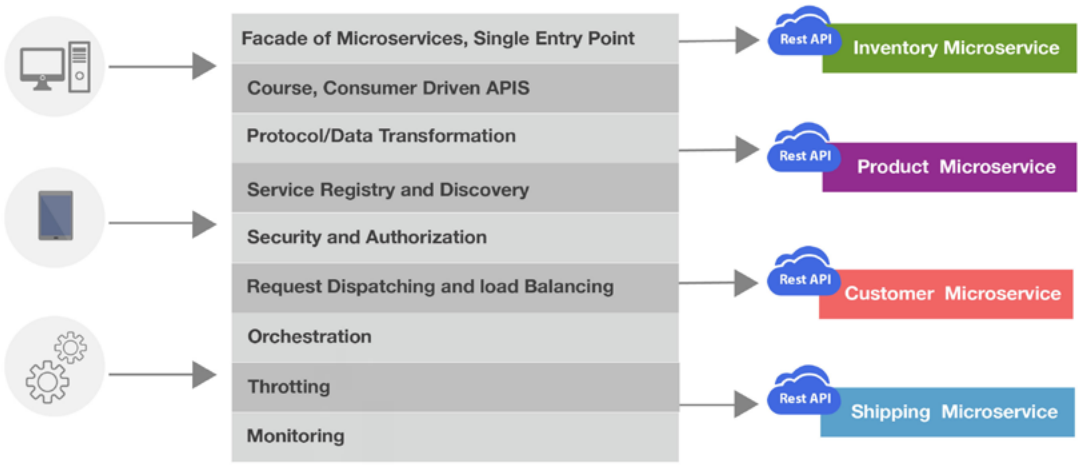


Figure 2. Demonstration of how a gateway typically fits into the overall microservice architecture

Security

Security is an important requirement of any enterprise solution. At a certain point in the architecture, the best options available for authentication, authorization, threat protection, message protection, etc. must be chosen.

Authentication and Authorization

Federated identity is the preferred solution for implementing authentication and authorization in microservice architecture. Each microservice does not necessarily need to obtain and store users' credentials in order to authenticate them. Instead, microservices can use an identity management system that is already storing a user's identity to authenticate the user. This approach allows the decoupling of the authentication and authorization functions. It also makes it easier to centralize these two functions, to avoid a situation where every service must manage a set of credentials for every user.

There are three major protocols for federated identity: OpenID, SAML, and OAuth. The figure below shows the security architecture using OAuth2.0.

Every API request is authenticated at the gateway layer. On behalf of the end user, the application client first grabs an access token from the authentication server by presenting credentials. This access token is then passed along with the API request in Authorization HTTP header. The API Gateway then validates the access token with the authorization server. The JWT token, which contains the claims for the user, is then passed to backend microservices. Backend microservices then use the information inside the JWT token for authorization purpose. The same JWT token is passed along when one microservice communicates with another microservice.

The flow above has the potential to be a "confused deputy problem," as every microservice is relying on the API Gateway for authentication. Ideally, every microservice should authenticate the token as received from its caller (gateway or microservice). There is a trade-off between security and performance. The above mentioned architecture leans toward performance, as there are other mechanisms to mitigate the security risk.

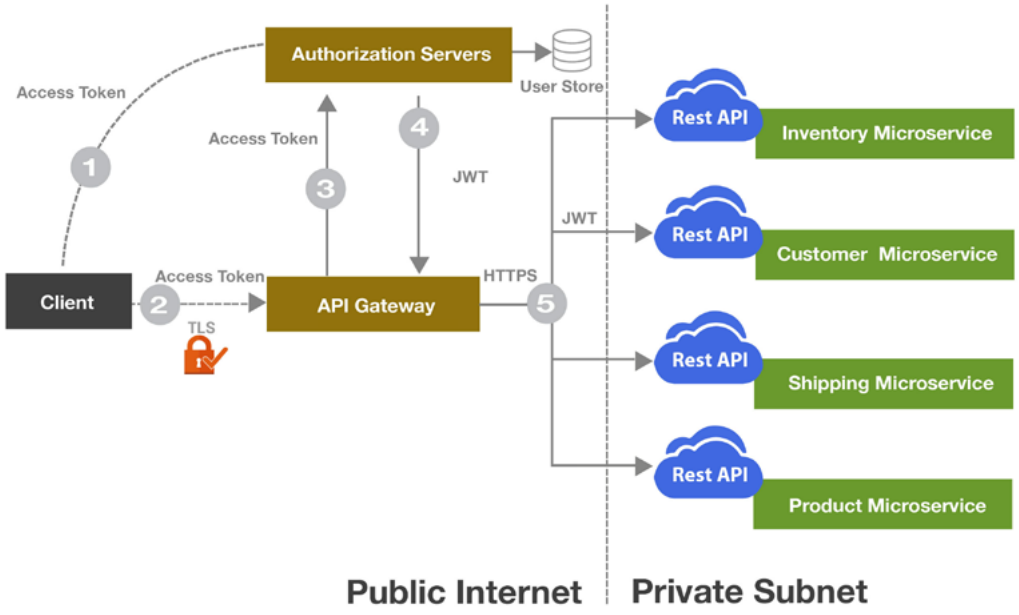


Figure 3. Security architecture using OAuth2.0

The microservice architecture is part of the overall IT infrastructure for an enterprise. If the enterprise IT is cloud focused, then you should use a well-known cloud based authorization server, like Azure Active Directory or AWS IAM, which can also be integrated with on premise identity stores, like active directory. An open-source server like IdentityServer4 makes it possible to implement your own authorization server and integrate with existing identity stores.

Threat Protection from DDoS

Most of the API Gateway provides (either integral or add-on packages) features that can handle DDoS attacks, by regulating and controlling the traffic as it proceeds to backend microservices. Consider configuring the following traffic regulating parameters for the API Gateway:

- Limiting the rate of requests: Maximum number of requests an API can access within a given time frame, based on rate limiting approach. Some of the approaches are Authenticated User, Request Origin, Authenticated User, and Request Origin. For example, you might decide that an API may be accessed only once a day, by an authenticated user from a specific mobile application.

- Limiting the number of connections: Maximum number of connections that can be opened by a single client for an API

- Closing slow connections: Time span, after which a connection will be closed from a client that is writing data too infrequently, which can represent an attempt to keep connections open as long as possible

- Black list / White list IP addresses, if you can definitely identify valid and invalid end users of your APIs

- Limiting the connections to backend microservices

- Blocking requests:
    - o that seem to target a specific API
    - o that have a User-Agent header, set to a value that does not correspond to normal client traffic
    - o that have a referrer header, set to a value that can be associated with an attack
    - o that have other headers with values that can be associated with an attack

Secure Communication

It is always desirable to have SSL/TLS compliant endpoints at the API Gateway, as well as at the microservices layer, to safeguard against man-in-middle attacks, and bi-directional encryption of message data to protect against tampering.

If you are dealing with a number of certificates, then managing those becomes a huge administrative burden. There are solutions like letsencrypt.org, an AWS certificate manager, which makes it possible to transparently issue or revoke certificates.

Deployment Considerations

To strengthen security further, you should host all microservices on private subnet and whitelist the gateway IP at the microservices layer. If it is not possible to have microservices on private subnet, then you should validate each token with the authorization server per service call, however, this will impact performance.

Service Registry and Service Discovery

Ease in scaling is one of the key advantage of microservice architecture. You will keep adding or removing microservice instances to adapt to incoming traffic. In addition, your teams may be adding new microservices or refactoring existing microservices into multiple (especially when moving from monolith to microservices). Service instances have dynamically assigned network locations.

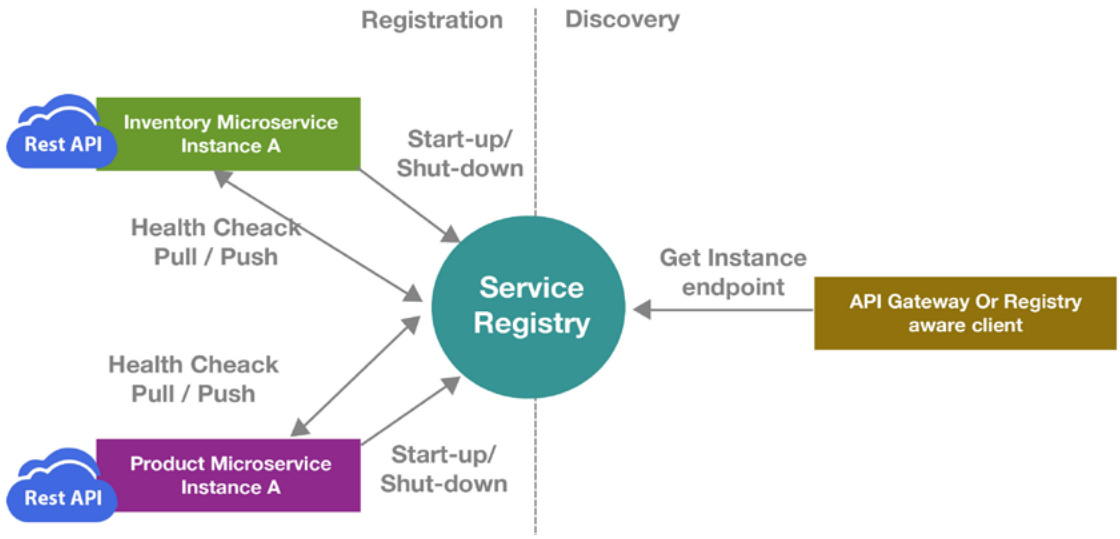
Service Registry

The service registry helps to keep track of these instances. It is a database containing the network locations of the service instances. Every service instance registers itself on start-up and de-registers on shutdown. The API Gateway uses this information during service discovery. The figure below shows service registration and discovery.

There are two models for checking the status of a service: pull model and push model. Although some of the registries support the pull model, it is not recommended, as it puts an additional load on the registry. Moreover, it is the service's responsibility to update the registry about its availability/ unavailability to serve the request.

As a critical component, the service registry needs to be highly available. You should plan for a cluster of registries that uses replication to maintain consistency. Never try to cache the network locations obtained from the registry at the registry user (API Gateway or registry aware client). That information eventually becomes out-of-date, causing clients to become unable to discover service instances.

Figure 4.
Service registration and discovery



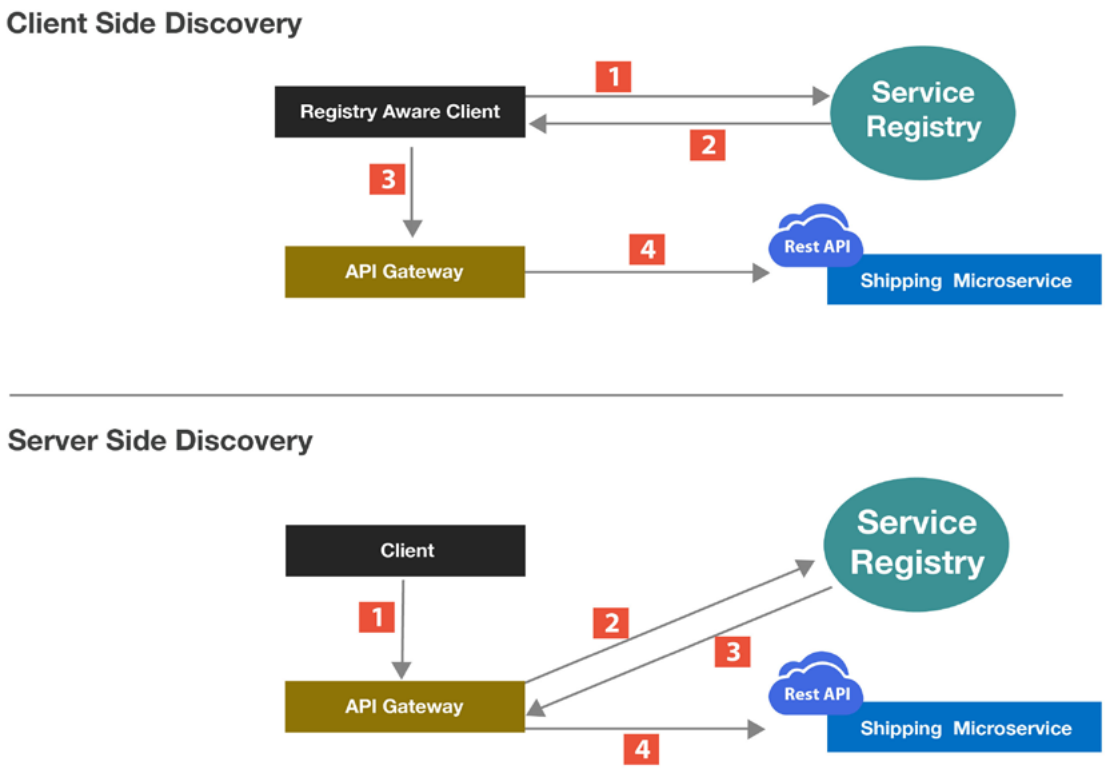
Service Discovery

The number and location of service instances is dynamic. Consequently, your client code needs to use a more elaborate service discovery mechanism. There are two main service discovery patterns: client-side discovery and server-side discovery, as shown in the figure below.

Server-side discovery is preferred for various reasons:

- It removes the discovery burden from the client so it can focus on business functions.
- If you have multiple clients, then you need to code and maintain the discovery code for each client.
- It reduces number of calls over the internet.

Figure 5.
There are two main service discovery patterns: client-side discovery and server-side discovery.



Orchestration

It is often necessary to orchestrate across multiple fine-grained microservices to accomplish a business use case. As shown in the figure below, there are two options for implementing the orchestration: using the API Gateway as the orchestration layer or coding orchestration in a separate microservice.

You should avoid orchestration at the gateway layer. It violates the single responsibility principle and, in the case of scaling the API, you will have to scale the gateway along with orchestrated microservices. Some of the API Gateways have little to no capability for orchestration.

Although it is discouraged to use orchestration at the gateway layer, if you still want to use it for whatever reason, then there should not be any business logic involved while orchestrating.

Transformation

Often microservices have to deal with different clients on the front end. They have different needs, from both protocol (SOAP, REST, JSON,and XML) and data perspective. Similarly, when you are moving from monolith to microservices, backend services may understand different protocols (SOAP, REST, AMQP etc.).

The API Gateway provides a place for data transformation, where messages can be translated between backend, API, and app formats and protocols. The gateway provides a central data transformation point through which all traffic is translated for:

- Requested payload transformations
- Header transformations
- Protocol transformations

Develop your transformation adapters as reusable components. Also, do not try to code all different clients' needs in a single API. You should think of creating client-specific APIs with pluggable transformation logic.

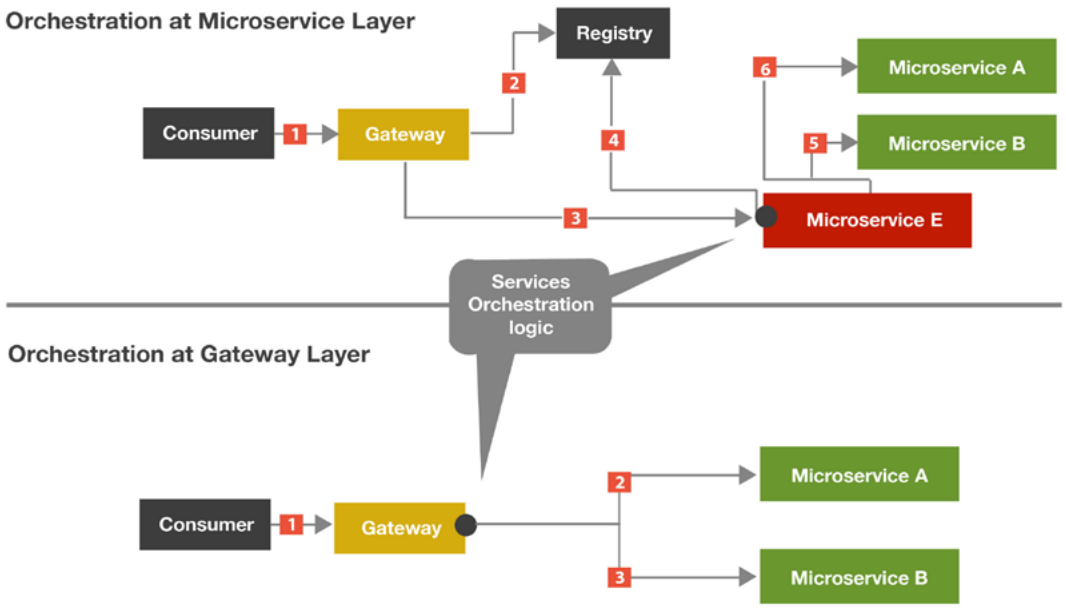


Figure 6. There are two options for implementing orchestration: using the API Gateway as the orchestration layer or coding orchestration in a separate microservice.

Monitoring

Being a single entry point into the system, all of the traffic in and out of the system passes through the API Gateway, so monitoring the gateway is critical. This provides an opportunity to capture the information about data flow, which becomes an input for IT administration and IT policies.

Health Monitoring

Health monitoring is done to make sure the gateway is up and running. For health monitoring, it is recommended to capture:

- System status and health (CPU, Memory, Thread usage)
- Network connectivity
- Security alerts
- Backups and recovery
- Maintenance of logs

Traffic and Data Monitoring

Analyzing the traffic data will help you to take proactive measures, to ensure smooth working of the software systems and shape up the IT policies. You should consider monitoring the following basic metrics:

- Number of requests per API
- Request categorization by criteria (for example, remote-host)
- Performance statistics
- Successful and exception messages
- Blocked messages breaching gateway policies

You should also regularly analyze the traffic over long range period to be able to identify predictable traffic levels and be ready for any surge.

Load Balancing and Scaling

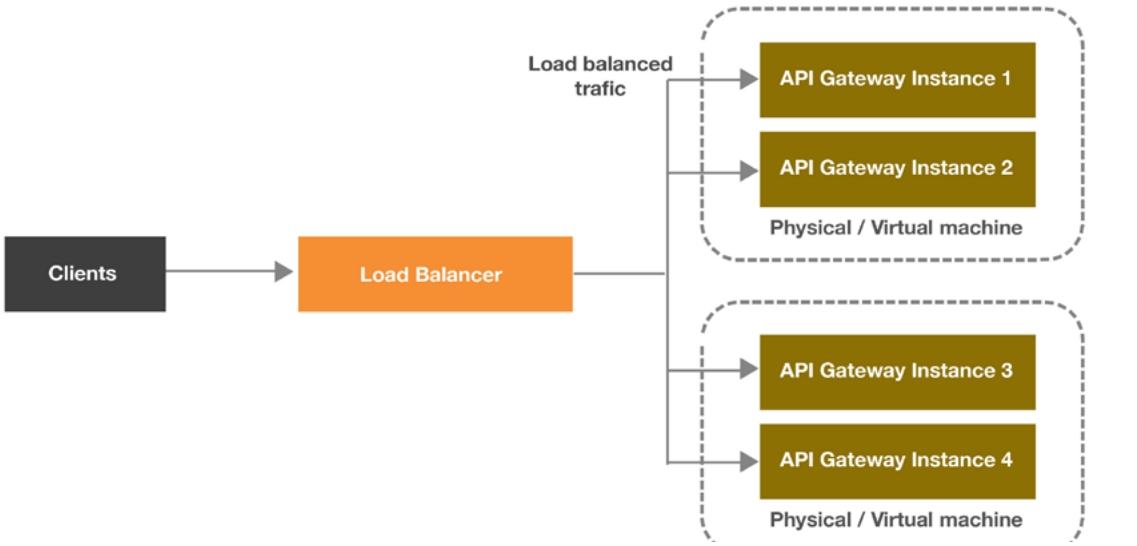
Traffic and data analysis helps in understanding/estimating the load on the system. In turn, this helps in scaling the gateway and underlying services accordingly. The gateway can scale both horizontally and vertically. An API Gateway being load balanced runs the same configuration to virtualize the same APIs, and executes the same policies. If multiple API Gateway groups are deployed, load balancing should be across groups.

The API Gateway does not impose any special requirements on load balancers. Loads are balanced on a number of characteristics including the response time or system load. The execution of API Gateway policies is stateless, and the route through which a message takes on a particular system has no bearing on its processing. Some items like caches and counters, which are held on a distributed cache, are updated on a per-message basis. This helps the API Gateway to operate successfully in both sticky and non-sticky modes.

The distributed state poses a certain restriction in terms of active/active and active/passive clustering. For example, in case the counter and cache state is important, the system should be designed to make sure at least one API Gateway is active at all times. This means that for a resilient high-availability system, you should employ a minimum of at least two active API gateways at any given time, extra in passive mode.

The API Gateway also ensures zero downtime by implementing configuration deployment in a rolling fashion. This means, while each API Gateway instance in the cluster or group takes a few seconds to update its configuration, it stops serving new requests, but all existing in-flight requests are honored. Meanwhile, the rest of the cluster or group can still receive new requests. The load balancer ensures that requests are pushed to the nodes that are still receiving requests.

Figure 7.
Load balancing and scaling



High Availability and Failover

Being a critical component and ONLY entry point in microservice architecture, you should deploy the API gateway in High Availability (HA) mode. API Gateway instances are usually deployed behind standard load balancers, which periodically query the state of the API Gateway. If a problem occurs, the load balancer redirects traffic to the hot stand-by instance.

Events/ alerts are configured for getting notifications in case any issue occurs. If an event or alert is triggered, the issue can be identified using API Gateway analytics and the active API Gateway can then be repaired.

API Gateway instances are stateless by nature. No session data is created, and therefore there is no need to replicate the session state across API Gateways. However, API Gateways can maintain cached data, which can be replicated using a peer-to-peer relationship, across a cluster of API Gateways.

High Availability can be maintained using either of these options: Active/Active, Active/Standby, or Active/Active Systems. These are described as follows:

- Active/Standby: System is turned off
- Active/Passive: System is operational but not containing state
- Active/Active: System is fully operational and with current system state

HA and Failover Guidelines

- To achieve maximum availability, API Gateway should be used in Active/ Active for each production API Gateway

- Proper traffic analysis, to limit traffic to backend services, to protect against message flooding. This is particularly important with legacy systems that have been recently service-enabled. Legacy systems may not have been designed for the traffic patterns to which they are now subjected.

- Monitor the network infrastructure carefully, to identify any issues early on. You can do this using API Gateway Manager and API Gateway Analytics. Interfaces are also provided to standard monitoring tools, such as syslog and Simple Network Management Protocol (SNMP).

Governance

As the number of APIs keep on increasing, it is essential to establish policies and monitoring. The policies can broadly be categorized as design-time governance and runtime governance. The policies are highly influenced by IT (business) objectives and goals.

Select and configure API Gateway policies that help you implement runtime governance, such as:

Tracking the life cycle of APIs:

- Handling routing, blocking, and processing
- Understanding the API utilization and raising the alerts/ alarms in case usage crosses the threshold
- Traffic throttling, smoothing, and load balancing
- Rate limiting per-API usage
- API versioning
- Schema versioning for input/ output request parameters

Design time governance includes:

- Defining the standards for API definitions (example: Swagger)
- Keyword tags used to categorize APIs
- Conformance to REST API design guidelines

Conclusion

The API Gateway is the essential component of microservices architecture. It helps to:

- Decouple consumers of the services from backend services
- Implement policies in one place
- Achieve reusability
- Monitor the entire technology platform performance
- Enable easy scaling of services

GlobalLogic

1741 Technology Dr.
San Jose, CA 95110

+1.408.273.8900
info@globallogic.com
www.globallogic.com

References

- Pattern: API Gateway / Backend for Front-End (Microservices.io)
- Building Microservices: Using an API Gateway (NGINX)
- Enabling Microservice Architecture with Middleware (WSO2 Blog)
- Welcome to IdentityServer4 (IdentityServer4)
- Oracle® Fusion Middleware Part 1. API Gateway administration (Oracle)