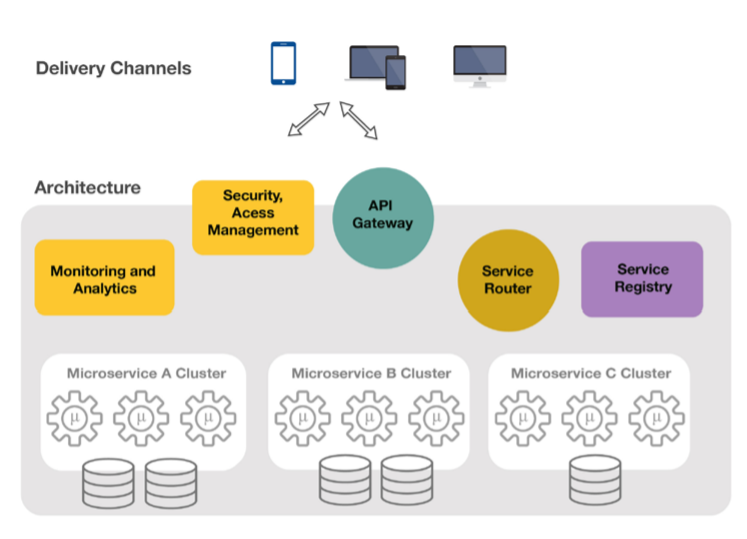
**API Gateway Pattern**

**1 Description**

The API Gateway is a reverse proxy to API / services 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.

It protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service, validates and sanitizes requests, and passes requests and data between them.



**1.1 Pattern Information**

|  |  |
| --- | --- |
| **Tag** | **Description** |
| Pattern Family | API |
| Pattern ID | API Gateway |
| Pattern Status | Draft |
| Version | 0.1 |
| Author(s) | Jamie Gunn |
| Sources | <http://microservices.io/patterns/apigateway.html>  <https://docs.microsoft.com/en-us/azure/architecture/patterns/gatekeeper>  <https://www.globallogic.com/wp-content/uploads/2017/08/Microservice-Architecture-API-Gateway-Considerations.pdf> |

**1.2 Context**

Applications expose their functionality to clients by accepting and processing requests. Applications expose endpoints clients connect to, and typically include the code to handle the requests from clients. This code performs authentication and validation, some or all request processing, and is likely to accesses storage and other services on behalf of the client.

In general, an API Gateway will perform the following Non-Functional capability. These are known as **Offloading**.

* Authentication
* Authorization
* Sessions
* Cookie Management
* Cache
* Throttling / Rate Limiting
* Monitoring
* Logging

The gateway will also provide **routing, aggregation,** and **transformation**.

**1.3 Problem**

In a typical enterprise, API’s need to handle a significant amount of cross-cutting concerns – however, these API’s are not always implemented in a consistent manner, platform, programming language, etc. This causes unique implementations of security, logging, monitoring, etc.

Additionally, organizations are continually looking for ways in which to allow their assets to be exposed as a revenue enhancing source. API Gateway is a common way in which these assets can be protected in a consistent manner while also allowing for quality monitoring, logging, throttling, etc.

**1.4 Forces**

Forces For:

* **Controlled validation.** The API Gateway validates all requests, and rejects those that don't meet validation requirements.
* **Granularity.**The granularity of APIs provided by services is often different than what a client needs. Services typically provide fine-grained APIs, which means that clients need to interact with multiple services. For example, as described above, a client needing the details for a product needs to fetch data from numerous services.
* **Client Dependent Data.** Different clients need different data. For example, the desktop browser version of a product details page desktop is typically more elaborate than the mobile version.
* **Performance.**Network performance is different for different types of clients. For example, a mobile network is typically much slower and has much higher latency than a non-mobile network. And, of course, any WAN is much slower than a LAN. This means that a native mobile client uses a network that has very difference performance characteristics than a LAN used by a server-side web application. The server-side web application can make multiple requests to backend services without impacting the user experience where as a mobile client can only make a few.
* **Abstraction.** Partitioning into services can change over time and should be hidden from clients
* **Protocols.** Services might use a diverse set of protocols, some of which might not be web friendly

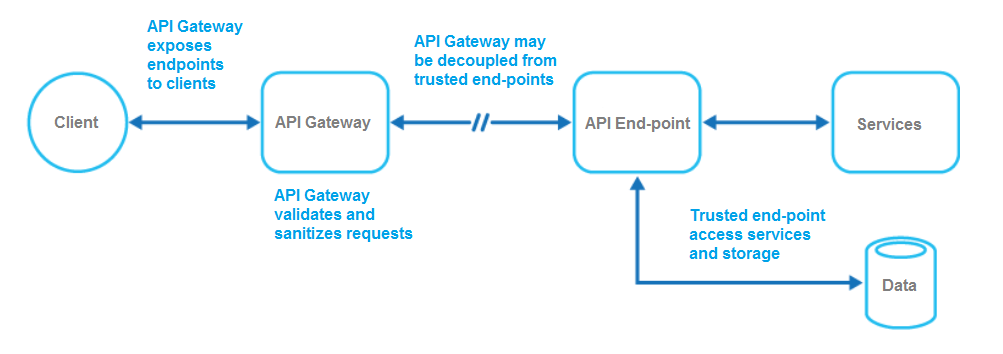
Forces Against:

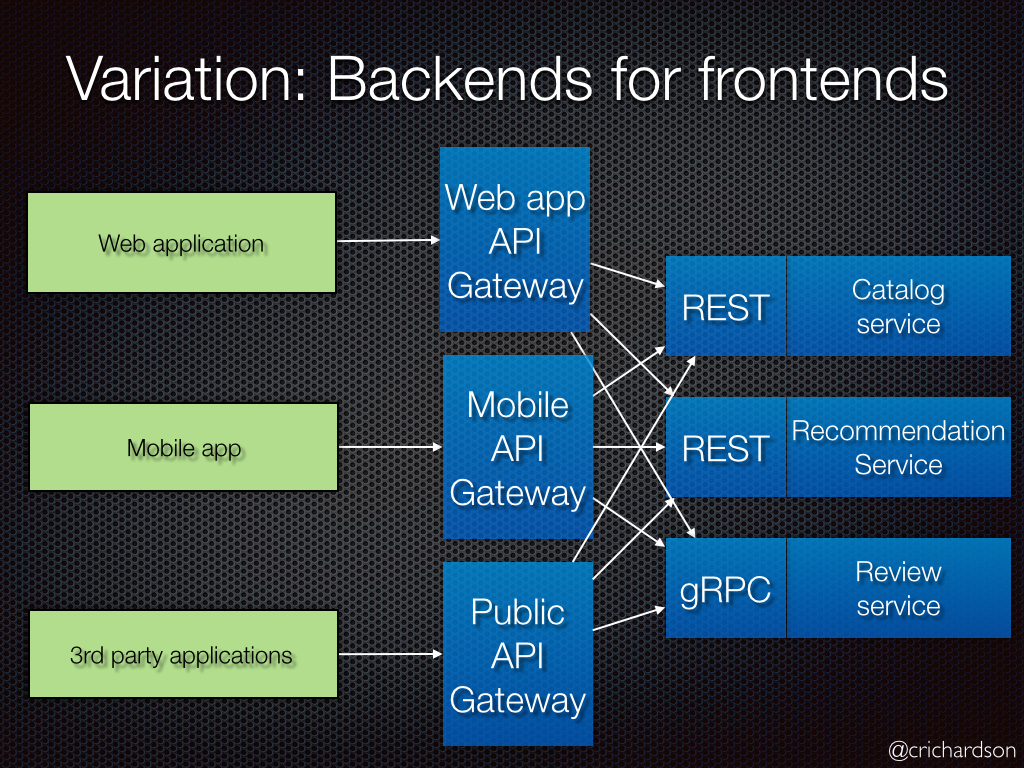
* **Complexity.**Increased complexity - the API gateway is yet another moving part that must be developed, deployed and managed
* **Latency.**Increased response time due to the additional network hop through the API gateway - however, for most applications the cost of an extra roundtrip is insignificant.
* **Long Running Tasks.**Synchronous Long Running Tasks.  The issue is that remote procedure calls are blocking calls. The calling application is waiting for a response and it is blocked from doing work until there is a response. We can mitigate this impact by using sync-over-async invocation methods.
* **Tight Coupling.**Can become tightly coupled in a poor orchestration process as it introduces a dependency between the two applications - if the application that publishes the interface is unavailable, then the calling application can become unavailable. To prevent this we must use the principles of software resiliency and apply patterns such as a timeout and circuit breaker to continue operating in a degraded fashion.
* **SPOF.**The instance could be a single point of failure. To minimize the impact of a failure, consider deploying additional instances and using an auto-scaling mechanism to ensure capacity to maintain availability.
* **Batching.** API Gateway is not tuned to batching capability
* **Messaging.** API Gateway is not tuned to messaging capability

**2.0 Solution**

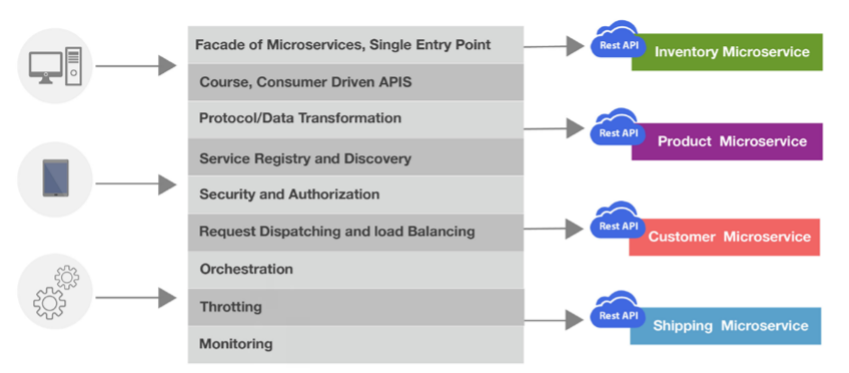
**2.1 Solution Description**

To minimize the risk of clients gaining access to sensitive information and services, decouple hosts or tasks that expose public endpoints from the code that processes requests and accesses storage. You can achieve this by using a façade or a dedicated task that interacts with clients and then hands off the request—perhaps through a decoupled interface—to the hosts or tasks that'll handle the request. The figure provides a high-level overview of this pattern.





(<http://microservices.io/patterns/apigateway.html>)



(<https://www.globallogic.com/wp-content/uploads/2017/08/Microservice-Architecture-API-Gateway-Considerations.pdf>)

The API Gateway pattern can be used to simply protect storage, or it can be used as a more comprehensive façade to protect all of the functions of the application. The important factors are:

* **Controlled validation.** The API Gateway validates all requests, and rejects those that don't meet validation requirements.
* **Limited risk and exposure.** The API Gateway doesn't have access to the credentials or keys used by the API Endpoint to access storage and services. If the API Gateway is compromised, the attacker doesn't get access to these credentials or keys.
* **Appropriate security.** The API Gateway runs in a limited privilege mode, while the rest of the application runs in the full trust mode required to access storage and services. If the API Gateway is compromised, it can't directly access the application services or data.
* **Simplicity.** Simplify the development of services by removing the need to distribute and maintain supporting resources, such as web server certificates and configuration for secure websites. Simpler configuration results in easier management and scalability and makes service upgrades simpler.
* **Specialize.** Allow dedicated teams to implement features that require specialized expertise, such as security. This allows your core team to focus on the application functionality, leaving these specialized but cross-cutting concerns to the relevant experts.
* **Consistency.** Provide some consistency for request and response logging and monitoring. Even if a service is not correctly instrumented, the gateway can be configured to ensure a minimum level of monitoring and logging.

This pattern acts like a firewall in a typical network topography. It allows the API Gateway to examine requests and make a decision about whether to pass the request on to the API Endpoint that performs the required tasks. This decision typically requires the API Gateway to validate and sanitize the request content before passing it on to the API Endpoint.

Components of the gateway include:

1. Routing
2. Offloading
3. Aggregation
4. Transform

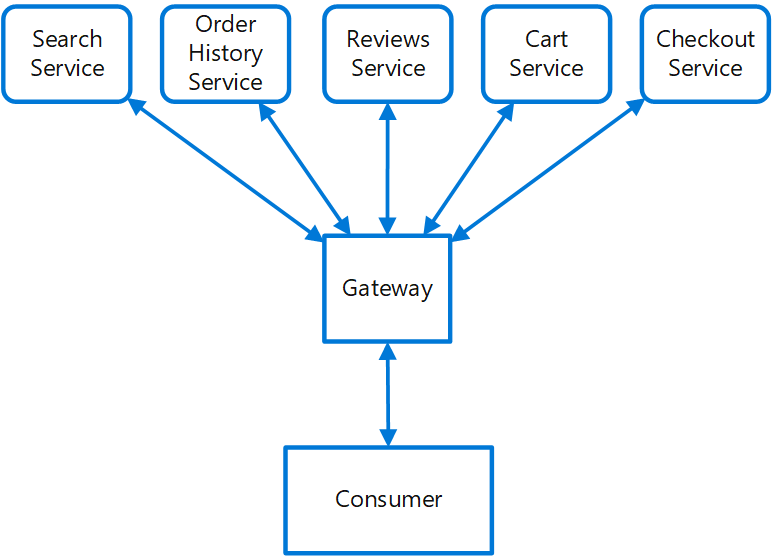
**Routing**

A gateway is placed in front of a set of applications, services, or deployments. It will use application Layer 7 routing to route the request to the appropriate instances.

With this pattern, the client application only needs to know about and communicate with a single endpoint. If a service is consolidated or decomposed, the client does not necessarily require updating. It can continue making requests to the gateway, and only the routing changes.

A gateway also lets you abstract backend services from the clients, allowing you to keep client calls simple while enabling changes in the backend services behind the gateway. Client calls can be routed to whatever service or services need to handle the expected client behavior, allowing you to add, split, and reorganize services behind the gateway without changing the client.

Routing is the only strategy that is available if the implementation is using an API Gateway and wishes control the access, logging, directory mapping above and beyond what the OS will supply.

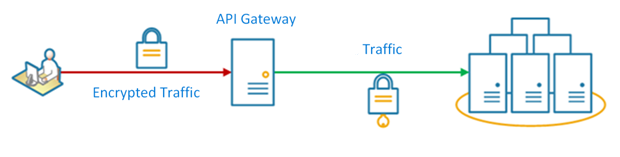


**Offloading**

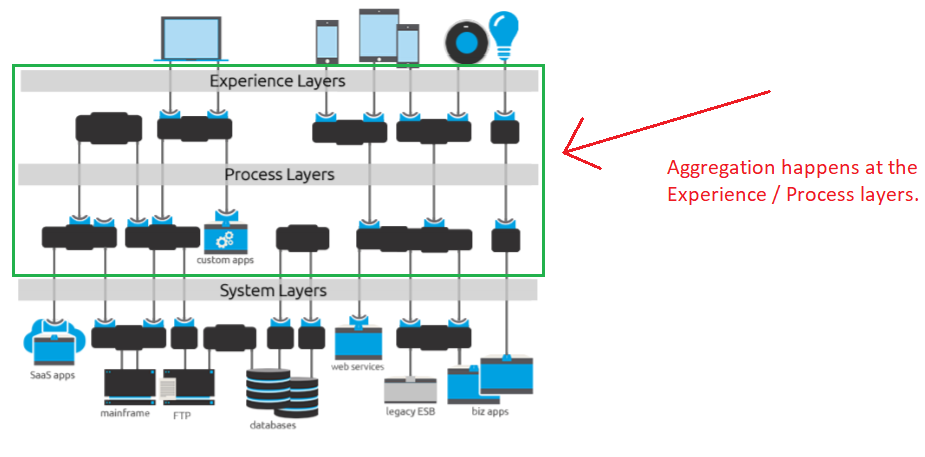
Common services such as authentication, authorization, logging, monitoring, or [throttling](https://docs.microsoft.com/en-us/azure/architecture/patterns/throttling) can be difficult to implement and manage across a large number of deployments. It may be better to consolidate this type of functionality, in order to reduce overhead and the chance of errors.

Recommend to offload features into an API gateway, particularly cross-cutting concerns such as certificate management, authentication, SSL termination, monitoring, protocol translation, or throttling.

The following diagram shows an API gateway that terminates inbound SSL connections. It requests data on behalf of the original requestor from any HTTP server upstream of the API gateway. The gateway will then provide all the cross-cutting concerns that are needed to provide security, monitoring, etc.



**Aggregation**

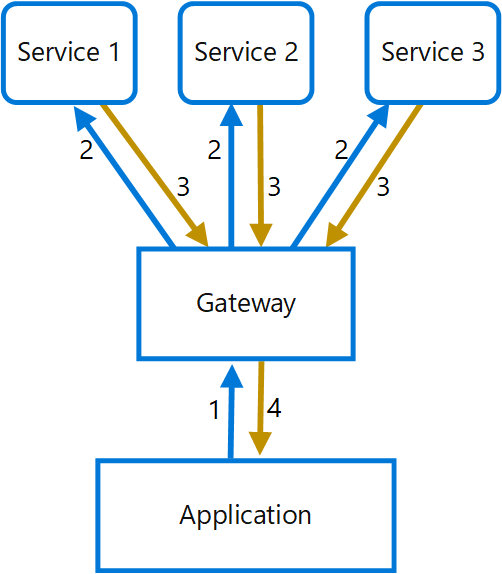


(<https://www.mulesoft.com/sites/default/files/resource-assets/API-led-connectivity-new-soa-updated.pdf>)

Use a gateway to reduce chattiness between the client and the services. The gateway receives client requests, dispatches requests to the various backend systems, and then aggregates the results and sends them back to the requesting client.

This pattern can reduce the number of requests that the application makes to backend services, and improve application performance over high-latency networks.

In the following diagram, the application sends a request to the gateway (1). The request contains a package of additional requests. The gateway decomposes these and processes each request by sending it to the relevant service (2). Each service returns a response to the gateway (3). The gateway combines the responses from each service and sends the response to the application (4). The application makes a single request and receives only a single response from the gateway.



The following should be considered:

**NOTE: We discourage placement of Business Logic in the Gateway. Business Logic is typically logic that incurs “if then else” or “sum” or “pick the first item in the list” type logic.**

* The gateway should not introduce service coupling across the backend services.
* The gateway should be located near the backend services to reduce latency as much as possible.
* The gateway service may introduce a single point of failure. Ensure the gateway is properly designed to meet your application's availability requirements.
* The gateway may introduce a bottleneck. Ensure the gateway has adequate performance to handle load and can be scaled to meet your anticipated growth.
* Perform load testing against the gateway to ensure you don't introduce cascading failures for services.
* Implement a resilient design, using techniques such as [bulkheads](https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead), [circuit breaking](https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker), [retry](https://docs.microsoft.com/en-us/azure/architecture/patterns/retry), and timeouts.
* If one or more service calls takes too long, it may be acceptable to timeout and return a partial set of data. Consider how your application will handle this scenario.
* Use asynchronous I/O to ensure that a delay at the backend doesn't cause performance issues in the application.
* Implement distributed tracing using correlation IDs to track each individual call.
* Monitor request metrics and response sizes.
* Consider returning cached data as a failover strategy to handle failures.
* Instead of building aggregation into the gateway, consider placing an aggregation service behind the gateway. Request aggregation will likely have different resource requirements than other services in the gateway and may impact the gateway's routing and offloading functionality.

**Transform**

Transformation is the process of re-shaping or changing of a protocol, character set encoding, http header, data type, or adjusting the return payload per the client that is making the request.

* API virtualization and mediation
* Bi-Directional transformation (for example, REST-to-SOAP, XML-to-JSON, and HTTP-to-JMS)
* Wide range of protocols, data formats, and standards

**2.2 Applying the Pattern**

When applying the messaging pattern, the following should be considered:

* Applications that handle sensitive information, expose services that must have a high degree of protection from malicious attacks, or perform mission-critical operations that shouldn't be disrupted.
* Distributed applications where it's necessary to perform request validation separately from the main tasks, or to centralize this validation to simplify maintenance and administration.
* Development of processes to manage security profiles, migrate API’s onto the Gateway, etc.
* Security (Supports, oAuth, SAML, JWT)
* Service Registration / Discovery

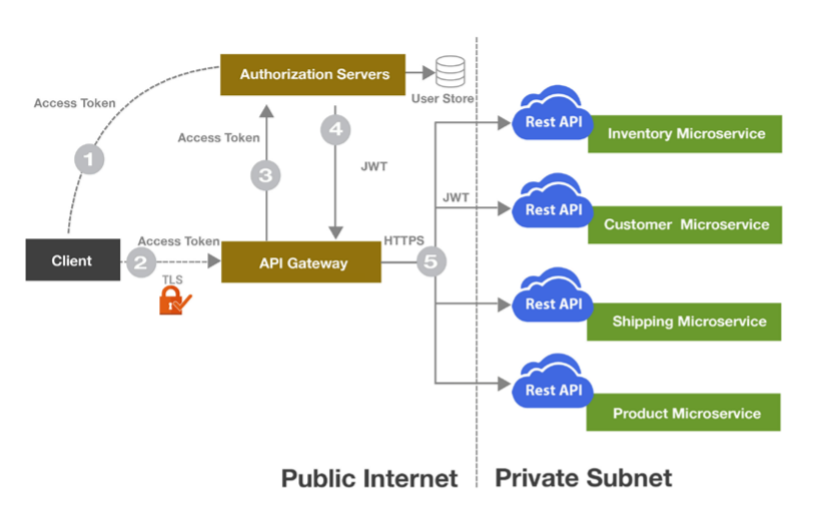
**We recommend that no business logic is placed in the gateway.**

**NOTE:** Services behind the Gateway should not be performing the above functionality unless required for an ***INCREMENTAL*** capability.

The gateway will have the following offloaded to it.

* + Authentication
  + Authorization
  + Sessions
  + Cookie Management
  + Cache
  + Throttling / Rate Limiting
  + Monitoring
  + Logging

The following is an example of an authentication / authorization flow.



**2.3 Benefits**

* It makes the processes of API design, implementation, and management considerably simpler and more consistent.
* Handles the non-functional “ilities” so that each service does not need implement, such as Health Monitoring, Throttling, Security, Logging, Alerting, Rate-limiting
* Insulates the clients from how the application is partitioned into services
* Insulates the clients from the problem of determining the locations of service instances
* Provides the optimal API for each client
* Reduces the number of requests/roundtrips. For example, the API gateway enables clients to retrieve data from multiple services with a single round-trip. Fewer requests also means less overhead and improves the user experience. An API gateway is essential for mobile applications.
* Simplifies the client by moving logic for calling multiple services from the client to API gateway
* Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally
* Consistently provide cross cutting concern capability to new development.
* Quickly secure and provide cross cutting concern capability to existing API’s or services.
* An application deployment has a shared concern such as SSL certificates or encryption.
* A feature that is common across application deployments that may have different resource requirements, such as memory resources, storage capacity or network connections.
* You wish to move the responsibility for issues such as network security, throttling, or other network boundary concerns to a more specialized team.

**2.4 Examples**

* Mulesoft API gateway
* [Netflix API gateway](http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html)

**3.0 Architectural Guidance**

**3.1 Rationale**

API Gateway is our preferred application integration pattern for request/response (synchronous) web based applications, SaaS based integrations, and external partner integrations and in combination with messaging / enterprise service bus.

Common alternatives to leveraging an API Gateway include:

* Exposing individual API Endpoints and having them implement their own solutions to the cross-cutting concerns.

Additional alternative patterns to API Gateway for application integration are:

* Shared Data
* Flat File Transfer
* Messaging

The **Shared Data** pattern is often the simplest to apply since it doesn’t require the movement of data. However, this introduces tight coupling between applications in the form of shared data. A change in the shared data requires changes to all applications that use it. The use of this pattern is discouraged.

**Flat File Transfer** of data is not ideal for a variety of reasons. First, flat files are often created by bulk or batch processes and we have a strong preference for near real time integration over batch. Secondly, flat files must be processed by a single worker so the scalability and performance of the application becomes compromised.

However, there are times when we must communicate with business partners or applications that don’t support other forms of integration. In these scenarios the use of flat file transfer is permissible.

There are also situations where the payload to be sent via messaging is too large for messaging infrastructures. In that case a flat file may be sent, or the Claim Check messaging pattern can be used.

**Messaging.** This is a valid pattern for distributed applications to communicate with one another. However, if there is a need for real-time confirmation of the success or failure of an operation, messaging breaks down.  Sometimes messages can be sent more than once and it can be very difficult to implement idempotency, which requires very careful design of the message processing code.

**In 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

**3.2 Related Patterns**

* [Gateway Aggregation](https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation)
* [Gateway Offloading](https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-offloading)
* [Gateway Routing](https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-routing)
* [Backends for Frontends pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends)
* The [Microservice architecture pattern](http://microservices.io/patterns/microservices.html) creates the need for this pattern.
* The API gateway must use either the [Client-side Discovery pattern](http://microservices.io/patterns/client-side-discovery.html) or [Server-side Discovery pattern](http://microservices.io/patterns/server-side-discovery.html) to route requests to available service instances.
* The API Gateway may authenticate the user and pass an [Access Token](http://microservices.io/patterns/security/access-token.html) containing information about the user to the services
* An API Gateway will use a [Circuit Breaker](http://microservices.io/patterns/reliability/circuit-breaker.html) to invoke services
* An API gateway often implements the [API Composition pattern](http://microservices.io/patterns/data/api-composition.html)

**3.3 Known Uses**

* WOFCO has implemented the Mulesoft API Gateway