***What is the API gateway pattern?***

*The API gateway pattern is a common architectural pattern in which an API gateway sits between the client and a collection of microservices. The API gateway acts as a single-entry point for clients to access the microservices, allowing clients to interact with the microservices as if they were a single service.*

*The API gateway can perform tasks such as authentication, rate limiting, and caching to improve the performance and security of the microservices. This pattern is often used in microservice-based architectures to help manage and route requests to the various microservices.*

***When to use the API gateway pattern***

*There are several situations in which the API gateway pattern can be useful:*

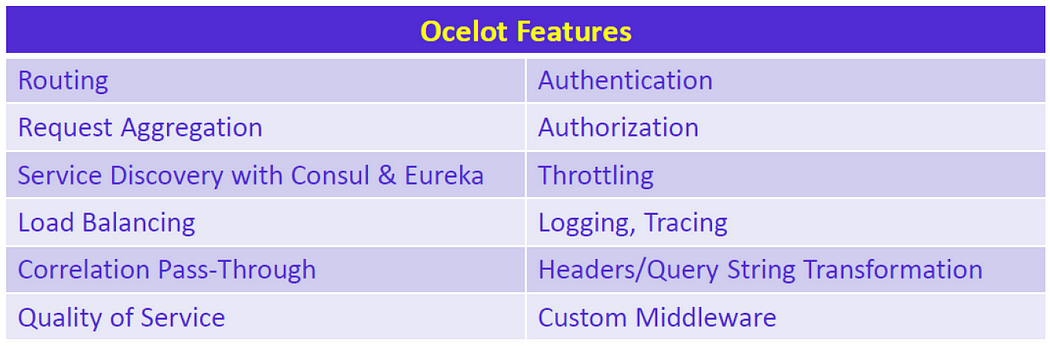
* *When you have a microservice architecture: If you have a system that is built using microservices, an API gateway can be used to route requests from clients to the appropriate backend service.*
* *When you need to expose your backend services to external clients: An API gateway can be used to expose your backend services to external clients over the internet, while still keeping the backend services protected and secure.*
* *When you need to add additional functionality: An API gateway can provide additional functionality, such as request routing, load balancing, caching, and authentication.*
* *When you need to hide the complexity of your backend services: An API gateway can hide the complexity of the backend services from the client, making the client’s code simpler and easier to maintain.*
* *The API gateway pattern is recommended if you want to design and build complex large microservices-based applications with multiple client applications.*

***Common API gateway design patterns***

1. *Centralized edge gateway - A centralized edge gateway is an API gateway design pattern in which all incoming requests are routed through a single API gateway, which sits at the edge of the system and routes requests to the appropriate backend service. This pattern is simple to implement and can provide a single entry point for all requests, making it easy to add security and other functionality. The API gateway can also provide additional functionality, such as request routing, load balancing, caching, and authentication. It provides a single, stable interface for clients to access the backend services, which can be beneficial for maintaining and scaling the system. It can also provide additional security by acting as a firewall, blocking malicious requests and only allowing requests that meet certain criteria.*
2. *Two-tier gateway - A two-tier gateway is an API gateway design pattern in which a client-facing gateway sits at the edge of the system and routes requests to a second gateway, which is responsible for routing requests to the appropriate backend service. This pattern can be useful if you want to separate the client-facing gateway from the backend gateway for security or scalability reasons. One advantage of the two-tier gateway pattern is that it allows you to separate the client-facing gateway from the backend gateway, which can be beneficial for security or scalability reasons.*
3. *Microgateway - A microgateway is an API gateway design pattern in which each microservice has its own dedicated API gateway, which routes requests to the appropriate service. This pattern can be useful if you want to give each service more control over its own traffic, but it can also be more complex to manage. It allows each service to have its own dedicated gateway, which can be beneficial for security or scalability reasons. However, the microgateway pattern can also be more complex to manage than other patterns.*
4. *Per-pod gateways - Per-pod gateways is an API gateway design pattern in which each pod (a group of one or more containers) has its own dedicated API gateway, which routes requests to the appropriate service. This pattern can be useful if you want to give each pod more control over its own traffic.* *It allows each pod to have its own dedicated gateway, which can be beneficial for security or scalability reasons. For example, you could use a different type of gateway or a different set of hardware for each pod, depending on your needs.*

***Best practices for implementing the API gateway pattern -***

* *Keep the API gateway lightweight and focused on routing and managing requests. Avoid adding business logic to the gateway, as this can make it difficult to maintain and scale.*
* *Use the API gateway to perform tasks such as authentication, rate limiting, and caching to improve the performance and security of the microservices.*
* *Use a decentralized approach to routing, where each microservice is responsible for its own routing. This allows each microservice to evolve and scale independently.*
* *Use a declarative routing approach, where the routes are defined in a configuration file rather than hard-coded in the gateway.*
* *Use a circuit breaker pattern to prevent cascading failures in the microservices. This allows the API gateway to automatically failover to a backup service.*
* *Monitor the performance of the API gateway and the microservices to identify potential issues and areas for improvement. Use this information to fine-tune the configuration of the gateway and the microservices to improve their performance and reliability.*



*Advantages of API gateway pattern –*

* *It encloses the whole internal structure of web applications.*
* *It never calls a particular service. For example, client interaction with API gateway.*
* *It helps in the simplification of code of the client-side.*

*Disadvantages of API gateway pattern –*

* *The main drawback of the gateway pattern is a single-point failure. If the gateway fails due to some issue, the backend becomes unresponsive. As a remedy, you can use multiple fine-grained gateways for client requirements (BFF) or think of a scalable gateway solution. Another issue is additional development and maintenance costs. If the gateway is not scaled out properly, it may become a bottleneck for the system as well.*
* *It is an important component for every web application means the web application services will be shown only if the API is up-to-date means updated.*
* *It becomes very important for each process for being lightweight because otherwise their time complexity will get increased because their developer has to wait in the process of updating API.*