POLITECNICO DI TORINO

MASTER'S DEGREE COURSE

in computer engineering

MASTER'S DEGREE THESIS

Deploying a scalable API management platform in an enterprise Kubernetes-based environment

Supervisor:
Prof. Risso Fulvio Giovanni Ottavio

Candidate:
Ignazio Di Natali

October 2020

ABSTRACT

APIs are the basis of the digital evolution, a revolution that has taken place thanks to the development of Information and Communication Technologies (ICT) which have led to heavy socio-economic changes and which has completely revolutionized the way to communicate.

APIs are interfaces that allow access to data and / or functionality, more generally to digital assets. Modern applications are no longer independent and one-sided but versatile blocks capable of exploiting shared features, services, and data accessible through the APIs.

In the Enterprise environment, the APIs, which were initially used only within the company itself to allow communication between the various technologies and platforms, have now become a business tool, exposing the APIs externally and making them accessible to the developer communities. Fundamental at this point is the way to expose the APIs that offer digital assets which need documentation to be understood and easily used by consumers.

The thesis aims not only to study the concept of API and its life cycle but to manage the inevitable proliferation of API in an enterprise context such as Avio Aero - at GE Aviation Business in order to optimize business processes, reduce costs and allow new applications to be implemented faster. The aim is to create an API management platform applied to a business process capable of enabling / facilitating real-time communication in a controlled and safe manner, offering the necessary information to developers who want to exploit the APIs and providing a monitoring and analysis system in real-time regarding the use of the APIs.

Table of contents

ACKNOWLEDGMENTS

First, I would like to thank my academic supervisor, Dr. Fulvio Risso, who allowed the realization of this wonderful internship experience, for his availability.

Special thanks to my company tutor, Dr. Quirico Davis, actively present during the internship at Avio Aero.

A special thanks to my family who supported me throughout my university career by helping me achieve this goal.

1 Introduction

This first chapter aims to provide a general overview of what APIs are, their use and the benefits that can be derived from a solution based on this type of interface. It is possible, in fact, to see that APIs have played an important role in the technical field thanks to their ability to make data and functionality easily accessible.

1.1 What is an API

Application Programming Interface (API) are interfaces that provide access to services, which can be simple data or functionality. It is possible to consider APIs as access doors and based on the needs for which they were created, they can have different characteristics in terms of security, accessibility, access priority and type of data provided, to name a few. The APIs offer a level of abstraction between user and services that allows the application developer (consumer) to ignore the code behind an API, thus receiving a complete, independent service to be incorporated in its development. It also represents a form of contract between the user and suppliers in which the terms of service, the protocols to be used, the type of security, the types and format of input / output data are defined.

The concept of API is an old concept, however, when the API topic is now dealt with, reference is made to the REST API introduced in Roy Fielding's dissertation "Architectural Style and the Design of Network-based Software Architectures" [1] which defines the interconnection of systems on the internet through the well-known Hypertext Transfer Protocol (HTTP).

eBay was one of the first to publish its first APIs, making its services available through its developer portal, thus encouraging innovation. In fact, after eBay many others followed such as Netflix, Amazon, and the more modern Facebook, thus contributing to the birth of digital ecosystems and creating a new market formed by APIs (products) and developers (buyers). In recent decades, there has been a huge increase in public APIs developed and accessible via the web.

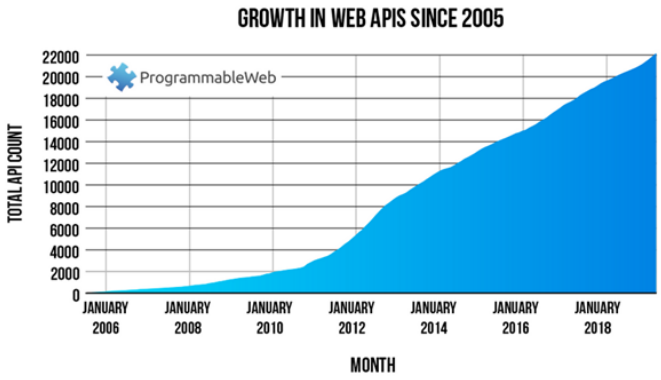


Figure 1 - Rise of the public APIs [2]

The graph above shows how the number of public APIs, according to programmableweb.com, a well-known information site on the web, is continuously growing with an average increase of around 2000 public APIs per year, demonstrating how the APIs have been and continue to be fundamental from the technological point of view.

1.2 API management

Starting from the service-oriented architecture (SOA) up to the creation of the microservices, it is possible to see a change in the development of the applications themselves which have become small, autonomous, independent. This aspect of technological development has brought numerous advantages in terms of agile development allowing new applications to release updated services quickly. The various microservices can be used through the APIs, and as the microservices increase, it is possible to note a consequent increase in the number of APIs and each of these must be managed appropriately to ensure maximum efficiency.

When referring to API management, the discipline is not just about managing individual APIs, that is, those processes that go from creation to disposal. The concept of API management, above all at the company level, is much broader and refers to that set, not only of processes, but also of tools that allow, in addition to the creation of the API itself, the discovery and their use, how they are coordinated and monitored. The only creation of the API is a secondary aspect given the huge number of public APIs, accessible for free on the Web. The use of APIs at the enterprise level has

brought advantages from various points of view such as, for example, the reduction of development costs, thanks to the reusability of the APIs, regardless of whether they are public or developed by the company itself. Furthermore, the use of APIs has changed the way applications are delivered to the end user, moving from a platform that provides a specific service (monolith), to accessing different services through a single access device. APIs are therefore seen as the link between the various services capable of creating new digital ecosystems and are essential components for the development of a company.

Numerous companies begin to use this type of technology to offer end users quickly usable information. However, it is possible to note how, the uncontrolled use and without an appropriate governance of the APIs, can lead to problems of strong dependence between the systems and services. Certain situations can lead to problems such as poor visibility of APIs use and security problems, considering the mess between communications which, in the presence of a fault, would make it difficult to discover the cause. API management has become of fundamental importance and was born as a discipline to define governance in communications between systems and to provide tools for API control, management, and use.

1.3 Benefits of using API

After a description of the importance of APIs and how they have contributed to a considerable development from a technological point of view, a description follows on what are the advantages and benefits that can derive from their use.

1.3.1 API as a means of innovation

In an era marked by digital innovation, companies feel the need to follow this trend to remain competitive. However, for many of these, pursuing innovation is not at all easy given the old, often inflexible, IT infrastructure. The way in which applications are now developed depends mainly on the customer's needs and for this reason they must be agile and fast, easy to maintain and update.

The attention to data and the customer has allowed the birth of agile and flexible realities, an innovation process that is defined as "Disruptive innovation" [3],

that is the ability of small companies with limited resources to become competitive and often surpass companies already stated. However, it must be borne in mind that the basis of any development depends on the data made available, most of which is found in the back-end systems. APIs, by nature, make sure that this information is easily accessible, thus allowing the creation of new services that meet customer needs in a process of continuous innovation.

1.3.2 Reuse the same capabilities

One of the characteristics of the API is the ability to be reused for different purposes, obviously offering the same service. The reuse of the same functionality shows numerous advantages from various points of view: First of all, it is possible to avoid rewriting the same code, moreover since the same API can be used by different consumers for different purposes it reduces development costs. Finally, accessing a feature through a single-entry point offers advantages both from a security and an analytical point of view, by making access and use statistics to improve its use.

1.3.3 Regulatory compliance

Regulatory compliance is a set of rules that organizations must comply with. In the IT field, for example, reference is made to the GDPR (General Data Protection Regulation) [4] which has precise rules regarding the security and respect of personal data. The GDPR is a European regulation which defines the guidelines for the management of sensitive data of European citizens.

Another important regulation introduced by the European Union on data management is the PSD2 (Payment Services Directive 2) [5] which has rules to secure online payments and access to your bank details using third-part providers.

The two regulations shown are just a few examples that demonstrate the importance of data management and protection. The APIs, acting as a data access interface, lend themselves well to this type of regulation as they guarantee, in a relatively simple way, secure access to information by enabling

authentication and authorization systems, access control and notification of abusive or inappropriate data use.

1.3.4 API monetization

Since APIs provide access to data and functionality, these can represent a possible source of income not only directly but also indirectly. On the one hand, access to a specific service can only take place following a monetary transaction. The payment to use the APIs can be applied in different ways: a single payment for an unlimited use, a constant payment, that is a subscription (monthly, annual, etc.), payment based on the use of the API and therefore based on the number of times the API are accessed. The cost also varies according to the type of data provided. This type of gain is direct. The API can also be a means of indirect earnings, through marketing and brand disclosure operations through the services that the API exhibits and that will be used by other applications. An example of this type of revenue is the disclosure of your brand through advertisements embedded in applications. The application is used as a means of spreading services and products. This type of process occurs as a result of an agreement with the application owner.

For this reason, an API creation strategy should be accompanied by a monetization strategy capable of generating both direct and indirect revenues.

2 API management platform

Architectures based on Enterprise Resource Planning (ERP), that is systems that incorporate most of the business process (monolith), are highly inefficient from various points of view: costs, efficiency, agility. The birth of microservices, small application units which perform a single task and are independent from the rest of the system, has meant that the information is now federated, distributed on various platforms thanks to the use of the cloud computing paradigm (based on Software as a Service, Platform as a Service, Infrastructure as a Service) [6] that is the set of infrastructures and resources that can be configured in an efficient way capable of offering services to the customer through the use of the internet.

As stated in the previous chapters, APIs are considered the basis of the interaction between different services provided by companies through different digital channels. The growing use of APIs has prompted organizations to create management platforms with the aim of:

- release appropriate software development kits (SDKs) to support the entire API lifecycle.
- create in-depth documentation on the APIs used to be easily understood by consumers.
- secure access to resources through authentication and authorization mechanisms, management of keys and tokens to prevent any threats.
- control and management of incoming and outgoing traffic, including limitations and concessions based on customers to guarantee the service level agreement (SLA).
- monitoring of both generic and specific calls by service and by consumer to detect trends, anomalies, and faults.

The accomplishment of these criteria can be achieved through three fundamental elements:

- API gateway
- Life cycle management
- Developer portal

2.1 API gateway

The API gateway is the core of communications between applications and back-ends since it exposes the APIs thus allowing them to be invoked. Specifically, it has the task of managing incoming calls and forwarding them to the related back-ends which provide certain services. During this forwarding process, it can perform tasks relating to calls, both outgoing and incoming. These tasks can be functions to all calls or to specific calls. The main functionalities are explained below.

2.1.1 Security

The services offered using APIs often expose sensitive data that need a certain level of protection. In addition to the private APIs, used only internally, there are thousands of public APIs that allow access to data and functionality and therefore need protection as a source of possible attack on the company that provides them. APIs, even if public, should be used only after registration processes by consumers who use them for their own developments. This is a first approach that allows the identification of those who use the APIs. The API gateway has the task of making these services secure by exploiting efficient security mechanisms that guarantee an appropriate use of resources and that try to avoid threats. Security mechanisms have the following tasks:

Authentication and authorization: authentication is the process that allows to know the identity of the person, application or, more generally, of the caller who wants to access a service or a particular type of data. In the authentication process, the caller generally provides credentials, as proof of his identity. The login credentials can be username and password, token or certified according to the authentication protocol that is used. Once the caller has been identified, the authorization phase begins, that is the process by which it is possible to understand which data or functionality can be accessed. Generally during the authentication phase, if the caller has been validated, he receives a resource access token to be inserted in the header when making any HTTP request. The token allows to understand the role of the caller thus establishing which services it can access or not. Tokens have a duration that establishes the validity time,

after which, the token can no longer be used. In these cases, the caller who wants to access the same resource needs a new token accessible through a new authentication or through automatic refresh systems.

Avoid DDoS: A Distributed Denial of Service (DDoS) [7] is an attack that aims to saturate resources. Through large-scale callers, the goal is to keep the host busy so that it cannot provide services, generally via ping flooding or ICMP echo request, which are considered as a high priority for the kernel and which therefore processes them first of other requests. Given the function of the API gateway, considered as an entry point to numerous resources, it is the main candidate for this type of attack. The countermeasures for this attack are only palliative, however an API gateway must have mechanisms capable of recognizing and stopping threatening traffic flows avoiding processing them and wasting resources useful for the functioning of the system.

Script injection detection: Some types of attacks could result from the content of the request payload. The format contained in the payload could be, for example, SQL or an XML. Below is an example showing the real danger of these types of attacks**.**

Suppose that our application allows searching for a name within a database of university students and that the name sent as a request field is used for searching within an SQL query. Using the API (/students/{nameOfTheStudent}) we would have

select * from Students where student = "+queryparam.nameOfTheStudent +";

So, suppose we pass a string like:

"/students/Ignazio; drop table Students"

It can note that without due payload checks the consequences could be disastrous. The API gateway has the task of intercepting certain contents within the payload and providing data validation mechanisms to avoid these types of attacks.

Data encryption: making secure access to sensitive data avoids the risk of unauthorized access from the outside to the inside. However, it is necessary to consider the case in which this data is requested from the outside and therefore

the API gateway must be forwarded out. The API gateway must in fact be able to hide sensitive data so, even if intercepted, they cannot be understood. The use of encryption methods becomes essential to avoid possible intrusions and readings of data during communication. One of the main defence methods is the use of SSL / TLS which makes use of certificates and asymmetric keys allowing both the validation of the identity of the user making the request, and the encryption and decryption of the data.

The characteristics described above are just some of the features that the API gateway should be able to implement to guarantee an acceptable level of security. Obviously, not all APIs need the same level of security and this represents an element of discussion during the first phase of API development.

2.1.2 Network traffic management

Not all APIs expose the same data type or the same type of service. The use of the services may be different in relation to various factors such as: the business value that an API offers, the type (or class) of consumer who can access the API, the cost that must be paid for access the API, the number of times it can be accessed, the place where the call is made and the time of the day.

APIs can be associated with a monetary share of consumption. In fact, there are application that can make a different call number based on the type of subscription that application made for that specific API. Policies regarding the number of accesses are established by the API provider. For example, a consumer who pays the maximum fee will have unlimited access unlike a consumer who made a free subscription and will have a limited number of accesses. A good API management platform should be able to manage the diversity of accesses related, in this case, to policies.

An API management platform must also be able to detect peak calls. The services behind it may not support a huge amount of calls. For this reason, a call peak detection system that can lead to a throttling or slowing down of requests and allow the specific system or service to continue to function correctly. This

policy allows to stop DoS attacks previously explained. Generally, an upper limit is set on the number of requests that a service can accept.

Finally, during the hours of heavy network traffic, a system for detecting classes of consumers that can lead to management in terms of traffic priority. Some calls, in fact, have higher priorities than others and therefore have a different processing time.

2.1.3 Caching system

When an API is contacted, the gateway, by default, forwards the request to the underlying service. There are, however, some cases where this forwarding process is avoided to optimize the call-answer process. Answers to the caller are not always different. In fact, in the presence of this type of scenario, the service responsible for generating the response is not contacted. The API gateway must have a caching mechanism capable of saving static responses within its memory avoiding forwarding requests to the recipient. This greatly reduces the latency between the call and the subsequent answer. The cache is small memory and for this reason cached responses have a limited residence time.

2.1.4 Routing and orchestration

The presence of different services accessible through APIs, sometimes consisting of multiple instances to make the system scalable, requires the presence of an API gateway that intelligently routes calls.

The correspondence between the call and the respective service generally takes place via URL path. Often, the path present within the API call does not coincide with the service path. A gateway, in the context of an API management platform, should be able to have a mapping system between the incoming path and the true path of the backend service. This would allow a modification of the path during the execution of the call.

Another important aspect related to routing is the capacity that a gateway should have in balancing management. To improve performance, having an appropriate call distribution mechanism becomes essential to not overload the system.

Balancing is managed through appropriate algorithms that can be different depending on the chosen platform.

Finally, in a world made of microservices, it can find scenarios in which a response is composed of the combination of multiple calls to different services. Having a single unified response, on the one hand improves the user experience, on the other it is possible to gain advantages in terms of performance.

Starting from a single request, the API gateway must be able to manage and coordinate the various calls to the respective services, to incorporate the various responses into a single body and return the response to the caller. Avoiding making several calls, allows to take advantage of user-side performance.

2.1.5 Translation

As seen above, one of the main purposes of an API management platform is to make the services offered easily accessible. Sometimes, these services are not always modern and with easy-to-use interfaces. The goal of an API gateway should be to allow application developers to easily access information using cutting-edge mechanisms and interfaces that are easy to implement, understand and use. For this reason, the need, by an API gateway, to have translation mechanisms regarding the interfaces so as to avoid, on the one hand, modifying the back-end services, which are often difficult to update, and on the other to facilitate consumers during their development, implementing already known and easily usable systems and interfaces. In the context of translation mechanisms, reference is mainly made to protocol and format translation

About protocol transactions, the back-end services use, for example, the SOAP protocol, still widely used, but unsuitable in the development of modern applications. Translation systems between different formats, such as the one between SOAP and REST would allow consumers to easily use the services exposed by the APIs.

Format translation, on the other hand, is about change of the data format that the system uses. A data format widely used for its simplicity is, for example, JSON. However, not all back-end systems support this type of format, using,

for example, XML. An API management platform should therefore support mechanisms that allow modification of protocols and type of data used.

This type of feature promotes the development of APIs by following the API-first method, that is, considering the user experience and in this case the developer experience.

2.2 API life cycle management

This section aims to deal in more detail with the API life cycle and all the processes that can lead to a correct management of the API from the design phase to the disposal. The various processes that will be treated are part of an iterative cycle that leads to a continuous updating and improvement of the API.

The processes that will be described are always preceded by a strategic phase and definition of the objectives that the company wants to achieve through the API. Within this context, everything starts from the need to release a new service to internal and / or external customers. Meetings are organized in which the reasons that can lead to API-based development and therefore the usefulness are discussed, what are the benefits and the earnings, the potential customers and the reasons that can push customers to use their solution, etc.

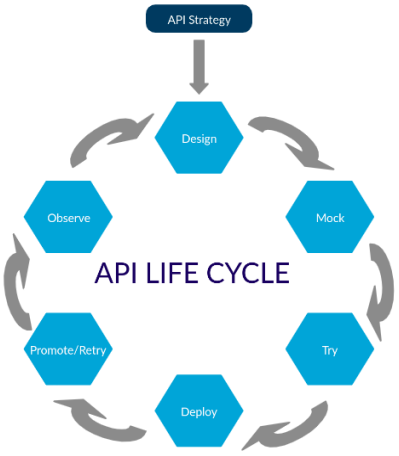


Figure 2 - processes that make up the life cycle of an API

As can be seen from the figure above, the processes are iterative and begin only after a careful strategic phase and definition of the objectives that are to be achieved. Below is a detailed description of the various stages of API life cycle management.

Design: The objective of this first phase is to make the discussed idea concrete. It is essential to have clear the main objectives and concepts discussed during the previous phase and therefore to have clear what are the functional and non-functional requirements. During this process, implementation decisions are made, which describe the components necessary for development and their capabilities, the architectural style to use, the expected performance, the necessary security policies, to name a few. After making the best choices regarding the first phase of development, the next step is to create a practical document that contains all this information. The document must not only describe the decisions that will be taken, but must also explain the discarded alternatives and justify them, so to understand the reasons that led to making certain choices rather than others, specifying the advantages and disadvantages of the different choices.

During the design process, the concept of reuse should not be overlooked. The existence of API with similar functionality must be considered to avoid implementing everything from scratch. An additional document will be created, following the interface description language (IDL) specifications, intended for API consumers, with the description of all the technical specifications and information necessary for use as methods, input and output data format, restrictions, call and answer format examples.

mock and try: After the creation of the document that describes all the specifications, the next step is to create a sort of simulator that has the purpose of testing the operations described in the document. Before starting the effective implementation, it is very important to carry out small tests that can lead to speed up development and avoid continuous maintenance. The mock allows API consumers to use the product and thus have feedback that helps to understand any problems. Furthermore, an additional advantage that the simulation through mock brings, is the fact of being able to make the switch from simulator to real implementation using the API gateway without the consumer notice it.

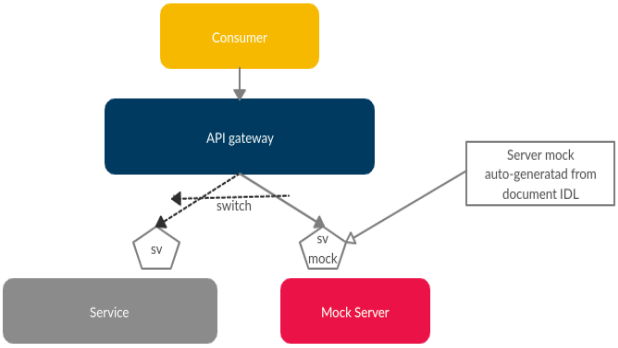


Figure 3 - switching from mock to real service

There are several solutions on the web that allow to create mock servers from some specifications. The most common are Swagger and Apiary. In the figure above it can see a representation of the switch using the API gateway.

Deploy: this phase consists in moving one's work in an environment suitable to be executed and used. This phase does not end with the deployment of the application but is composed of a series of iterative sub-processes, in line with the concept of continuous integration and continuous delivery (CI / CD). The goal is, in fact, to facilitate the continuous updates and releases of applications in an easy and safe way, without compromising the entire system. Before the release there are testing mechanisms with the aim of checking compliance with all established specifications, possible errors, backward compatibility with previous versions. With reference to the API, performance tests can be performed to avoid possible increases in latency and security checks using common attack mechanisms.

However, there are cases where some problems can arise despite having passed all the tests. In this regard, the use of fast rollback mechanisms becomes essential to return to using the previously functioning versions.

Promote/Retire: after the new API versions pass all the tests and after monitoring its operation it is possible that it becomes the default version. At this point, the previous version is initially regarded as deprecated and therefore continues to function and be used. In this phase, the documentation remains available, however, it will be reported as deprecated and usable only for a certain period. After this period, the API will be non-functional, and the related documentation will be removed.

Also, for this scenario there are techniques that support this phase of updates and releases, such as, for example, CanaryRelease in which the first releases can be seen only by a group of users, typically internal employees. If the release proves to be functional, then access for the rest of the users is enabled. Any problems will be solved through rollback mechanisms that allow to return to use the previous version until the next resolution of the problem.

Observe: this phase consists in keeping the functioning of system under control. During the design phase, it is generally necessary to manage many components that communicate with each other and it is appropriate to understand what happens if, for example, one of these components behaves unexpectedly. It is therefore necessary to have the opportunity to understand what is happening inside. It is the developer's task to create or connect to the system components that allow, from the outside, to understand if the various parts are working properly or not and, in the event of a problem, to have systems that allow easy identification the source. Monitoring systems help guarantee the Service Level Agreement (SLA), thus avoiding any delays caused by bottlenecks, increases in latency and possible attacks. Typically, observability is based on three observable levels: the first level concerns the network part and therefore the network transmissions, the second concerns the part of the machines that is the hardware part which, in the case of cloud provider is managed by the provider, finally, the observability that regards the application level. For each of these, the elements to consider when monitoring are logs, metrics, and tracing.

Logs are information regarding the events that applications write within a file and give information about what applications are doing. On the logs it is possible to make aggregations of information and analysis to better understand the reported data.

The metrics give quantitative information of what is happening to our system. It therefore provides information such as the latency in terms of response time of an API, the number of transactions per second, the number of accesses per minute, and so on. The metrics are generally used with alert systems, through the setting of thresholds that allow to identify unexpected operations.

Tracing is essential for distributed architectures because it allow, for example, within a microservice architecture or with many components, to reconstruct a single customer

request through all the modules that have been traversed. Having a clear view of the full path is critical to understanding what is going on.

Generally, information such as metrics and tracing can be extracted directly from the logs. By using a structured format for logs and using filters it is possible to obtain quantitative information. The use, however, of a unique identifier for a request allows to retrieve only the information of that request and see all the modules crossed. The use of visualization systems allows to have a graphic representation and therefore facilitate the understanding of the data described above.

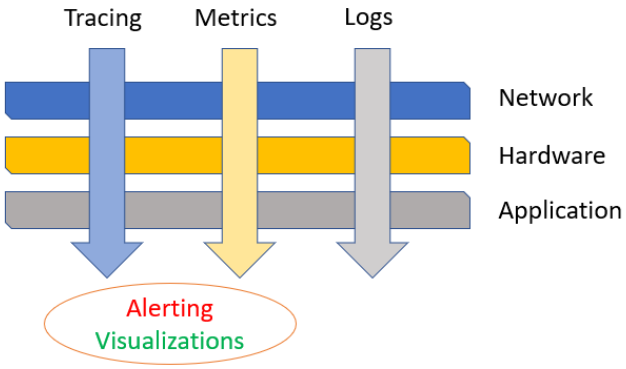


Figure 4 - representation of the components subject to monitoring

2.3 Developer portal

The success of an API does not depend only on good implementation and correct documentation. There is another important feature behind the success of an API: the developer portal, a place where the various APIs are published and made accessible to anyone. It is also a place that helps the developer to properly use the APIs that can be found inside and therefore improve the developer experience (DX). The purpose of this part is to describe in detail the benefits and features of a developer portal.

When a new API is created, in addition to following in detail the phases of the life cycle explained above, it would also be appropriate to follow strategies that can encourage the use of the newly created API. APIs are subject to constant evolution and updates and it becomes essential that consumers continue to be informed. The

developer portal was born from the need to advertise the newly created APIs and notify consumers in the presence of updates, to facilitate easy consumption. Developer portal is a place where application developers can learn and better understand the use of APIs published within. It is in everyone's interest to use the APIs, both for developers, promoting the rapid development of applications, and for API providers, especially in the presence of APIs subject to monetization. So, the best way to spread its use is to advertise APIs through platforms of this type. Inside there are not only documentation and instructions on how to use the API, therefore everything related to SLA, type of data to be used, input and output parameters, type of tariff (where present), but it can find a real place of debate between developers. The peculiar feature of an API developer portal is the possibility of having a simple search system that facilitates the developer during the appropriate API search process, the possibility of sharing and discussing about the API and their use with other developers. On the one hand, in fact, the portal represents a means of communication between developers and the portal provider, on the other it represents a means of communication between developers. It is a sort of social platform where it is possible to leave reviews or make requests on one or more published APIs, thus allowing the administrator to understand the needs of the developers and to resolve any API-related problems or bugs. In addition, it promotes mutual support between developers by sharing their experiences regarding the use of APIs and the resolution of any problems. An additional important feature that a developer portal should have is the possibility to implement internal mechanisms to test the API, real testing platforms that help developers understand the functionality and ways of using an API.

2.3.1 Actors of developer portal

The main actors around a developer portal are substantially three and have distinct objectives and purposes.

First, the owner of the API, typically a company, whose interest is to bring success to their product. The owner of the API has the task of understanding what the business interests of the company are and creating the API that can meet those specific needs. In addition to this, they are interested in the needs of the developers, aware that they will be the ones to use that product. Finally, it is

responsible for the architectural and usage choices, such as protocol and tariffs etc.

The second actor is the portal development team which is responsible for creating and managing the platform. The team is responsible for creating registration systems for an API. They also deal with the creation and subsequent updates of the documentation, the blogs that allow the debate between developers and the creation of the testing mechanisms of their API.

The third and final actor is the API consumer, therefore the application developer who uses the published API. Developers take advantage of all the services that the portal offers, therefore, API, documentation, testing and forums, subscribe their applications to the API of interest and interact with the community in search of clarifications or in support of other developers.

2.3.2 Features

This section covers what are the main features that an API developer portal should have to help consumers understand and use the APIs published within it.

Registration and login functionality: The registration and login processes are important steps for using the API, especially for those subject to a fee. Since everything is focused on the developer, it is good practice to make this process as simple as safe. Subscribing an application to an API allows the provider to enable the API for that specific application and apply the related usage costs to the consumer. Once the subscription has started, the next step is the approval by the provider who will have the task of approving the request and managing a key to be delivered to the consumer. However, there are automatic approval systems. A subsequent notification email will be sent to the API consumer as confirmation of the approval and therefore correct registration. Finally, the API supplier has the task of managing the keys, therefore generation/delivery and revocation.

API documentation: documentation is essential for the understanding and use the APIs. The documentation presents information regarding the APIs interface,

the protocols adopted, examples of requests and answers that help the developer to understand the type of format and the explanation of the fields present. The API documentation provides an important contribution for the developer as it speeds up use by avoiding making trivial errors.

Forum: A means that better helps to use the API, together with the documentation, is the comparison with the community of developers. The presence of social platforms allows to clarify doubts, solve implementation problems, and facilitate development. In fact, every developer can share their experience, leave reviews, and open questions and make reports. The forum is also a useful tool for API suppliers who, thanks to the contribution of consumers, can make changes and improvements to their products.

API test: it is also useful to implement platforms within the portal that allow to test the APIs before using them. They provide clear examples of possible requests and responses.

Consumer information support: the developer portal should finally have a contact section. The developer must be able to contact and communicate, via phone number or email, with the supplier for any problems related to the use of the API. Finally, a section dedicated to frequently asked questions is useful.

3 Technology

3.1 Docker

Containers are environments that arise from the need to efficiently manage concurrency and the coexistence of different and incompatible processes within the same machine. Containers are generated through the use of the "clone()" system call which has the ability to generate processes that are completely isolated, which do not share the memory space and specific operating system tables (file system, process and user ID, etc). This division takes place thanks to the creation of namespaces, that is, logical partitions capable of exploiting the host machine kernel. When the processes are scheduled, the kernel, which generally uses the operating system's global tables, can address them toward separate, virtualized tables that live within the namespace. The namespaces created are located within the host operating system and therefore allow software-based virtualization. Processes that do not use the operating system global tables live in the context of containers. Having many processes within the same system using hardware virtualization, and therefore using a hypervisor that manages the various virtual machines, becomes highly inefficient given the high number of machine cycles required. The Container has all the configurations it needs inside and is easily deployable. It is much lighter than a virtual machine because it requires fewer resources. This ability allows many containers to coexist within the same machine, thus promoting scalability. Further advantages of the containers are portability and the possibility of reuse because they are described within text files (dockerfile), therefore easily distributable and reusable.

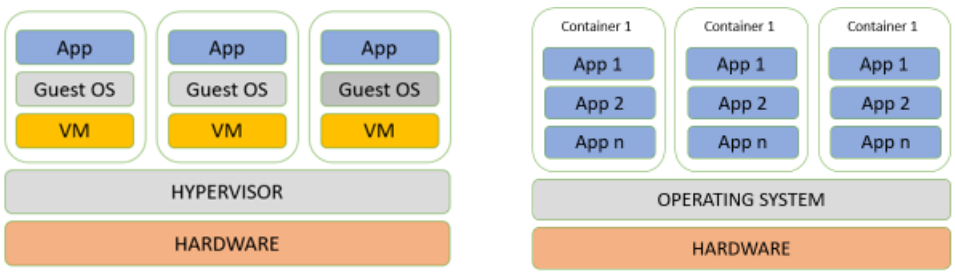


Figure 5 - Difference between Virtual Machine and Container

Docker [8] is an open-source software that allows the creation of containers and consists of four fundamental elements:

- **Docker client**: program that has the task of sending commands to the daemon, a background process accessible through REST interfaces and capable of creating, blocking, removing containers, etc.
- **Docker Image**: file system structure of the process to be executed which represents the basis for containers. From a structural point of view, the image is considered as a set of immutable layers. Starting from a basic configuration (layer) in which the structure of a specific file system is defined, it is possible to make changes or installations by overlapping additional layers that add files within the initial filesystem.
- **Docker container**: docker process running from an image. To the starting image, an additional layer is added with reading and writing skills.
- **Docker engine**: program loaded during the docker installation phase that allows to use all the features that the docker software offers such as the features described above and the interfaces that allow to interact with the docker system from the outside, also allowing to standardize the image and container management.

3.2 Kubernetes

Kubernetes [9] is an open source container orchestrator and manager born in 2014 by Google, widely used by many cloud providers around the world. The use of microservices has brought numerous advantages including the possibility of having different instances allowing to always have the service delivery alive (scalability), the property of isolating the error avoiding the block of the whole application (resilience), the ability to develop microservices using different languages and methods contributing to flexible development. These are just some of the advantages of using microservices.

Kubernetes was born from the need to coordinate the system from an application point of view. It aims to help the developer to deploy their applications and automate the management of the services exposed to the end user. The mere use of dockers makes management difficult, since there are no automatic systems that allow, for

example, automatic load balance between the different instances, or to have an automatic scaling system in relation to needs. Furthermore, since the various instances of the same microservices can run on multiple servers and undergo continuous processes of destruction and creation, the automatic communication and management of the various replicas becomes impossible.

The most important features offered by Kubernetes are:

- Ability to automatically schedule microservices on various server from a deployment point of view.
- ensure the functioning of a defined number of copies of the services chosen by the manager operator. It has the ability, in fact, to recreate one instance of the same service when another goes down.
- Autoscaling functionality: Allows to increase or decrease the instances of the microservices based on the consumption of certain parameters.
- Load balancer: capacity between the various instances of the microservices to avoid overloads that could lead to trouble.
- Ability to create a virtual networking within the system to ensure isolation between microservices.
- Ability to monitor the functioning of the applications, their use, and their status (health check). Kubernetes immediately restarts the service if an error occurs.
- Ability to simulate a DNS which keeps track of where exactly the application to contact is, unknown from the outside.

The basic element adopted by Kubernetes is the cluster. The cluster is a set of one or more nodes, units where applications run inside, and managed by a Master, unit that has the task of controlling and managing the various nodes of a cluster (control plane). The control plan includes further components with specific tasks and functions:

API server: it is the gateway within the cluster whose task is to allow access to the cluster from the outside through the exposure of the API. All requests go through the API server including the requests that are used to configure our system (using the kubectl command). Kubernetes supports application configuration and deployment using declarative mode. After writing the specs inside a script file (yaml file), a REST type request is made using the POST method and passing the configuration script to the API server and Kubernetes will update the cluster status.

etc: stateful archive made available for usable Kubernetes, for example, as a location for backup.

kube-scheduler: component within the control plane that has the task of managing and allocating the newly created Pods (minimum unit in Kubernetes in which the containers run. Explained later). The Pod creation phase is followed by the assignment phase in a node based on appropriate specifications such as availability of hardware resources, compatibility, etc.

kube-controller-manager: has the task of managing the nodes and Pods operating on the cluster.

cloud-controller-manager: system that allows to create a connection between the nodes that are within the cluster and the APIs of the cloud provider

nodes: as mentioned above, are the entities that host the running Pods. The containers inside a Pod are managed by an entity called kublet and communication with the outside takes place thanks to the use of kube-proxy that defines the network rules.

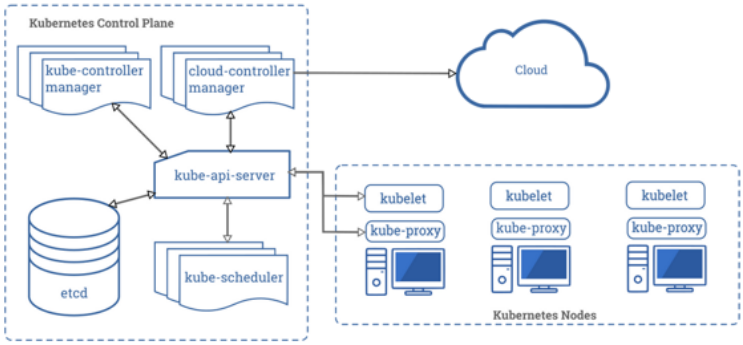


Figure 6 – high-level view of Kubernetes system

The figure, taken from the official documentation of Kubernetes [10], shows a high-level representation of the components described above. After seeing the components that make up a control plane within a cluster, a description will be conducted on what resources that allow to deploy our applications in a cluster and allow Kubernetes to manage them.

- **Pod:** The pod is the smallest element managed within a Kubernetes cluster. It has the property of hosting one or more containers inside. The containers that live inside a pod can be considered as belonging to the same host machine thus sharing the same network and can share the same storage spaces (volumes). The communication within the pod takes place, in fact, using localhost and it is possible to access the containers from the outside using the same IP address.
- **Deployment:** The pods are wrapped in high-level objects: the deployment. It is a component that adds functionality and flexibility to the Pod, like a controller, makes the pod scalable, allows to make simple updates and rollbacks. The figure below shows the layers of the application that runs on the cluster. The container, wrapped in a pod, is in turn enclosed within the deploy object, thus allowing scaling and updating of the pod itself.

Figure 7 - representation of the element wrapped in Kubernetes

- **Service:** Service is one of the most important components of Kubernetes. As explained previously, it can have multiple instances of the same Pod to make the application scalable. To each of the Pods, Kubernetes it associates a unique IP address and a label. The label is the same for all instances of the same Pod. The service allows to expose the Pods instances externally as if it were a single microservice. In this way it is possible to identify the microservice through the label and the Service resource will take care of balancing the requests between the various instances of the Pods in a completely invisible way from the outside.
- **Volume:** The volume in Kubernetes is a memory location that is used to make data relating to a Pod persistent. Pods are generally short-lived and are usually

subject to continuous processes of destruction and creation. In the presence of this scenario, the Pod does not have the ability to maintain the state and without a persistent memory location the data would die together with the Pod itself. The same volume can be shared by multiple Pods. Kubernetes is able to manage a memory location through the use of two main subsystems: persistent volumes (PV) which represents the memory volume available and the persistent volume claim (PVC) which represents the ticket that allows to use the PV.

Each node within the cluster has a set of IP addresses available and the Pods that run on that node are assigned an IP address within that range. Furthermore, since the Pods come and go and consequently the IP addresses always change, the Service resource is created and assigned an immutable name and IP address. Kubernetes guarantees that the name and IP address of the service remains unchanged during the entire life cycle. Communication between the various applications takes place thanks to the help of a DNS cluster. The Service resource, in fact, saves its name and IP address within the DNS Cluster thus making the microservice accessible to all nodes of the cluster by simply using the label of the Service resource. In addition, after the creation of a Service, an endpoint object is created, associated with it, where inside it can find all the IP addresses and ports that have the service label. The service periodically contacts the Kubernetes API server to find out if other pods with that label have been created or removed and updates the endpoint object accordingly.

3.3 Kong

Kong [11] is one of the most popular and used open source API management platforms in the world. It can make the connection of APIs to microservices quick and easy both in multi-cloud and hybrid environments. However, offer high performance by reducing latency and offering great scalability for all microservice applications. In the Enterprise environment, it simplifies the management of the APIs by automating activities and quickly identifying anomalies and failures. Even if the set of features is not extensive, Kong API gateway still offers an essential set of features such as authentication and authorization, traffic control and monitoring, also offering

configuration extension capabilities thanks to the use of default and customizable plug-ins. As described above, Kong's purpose is to orchestrate, manage and organize API calls to the related microservices to have full control of the entire communication structure. All this is made possible through the NGINX-based Kong server, a high-performance server / reverse proxy designed to process not only the API, but also to run the plug-ins, previously configured, which offer additional features to be applied to calls before to be forwarded to the relevant endpoints. To allow access to traffic from outside, Kong listens on predefined ports. The main ones are 8000 and 8443 for HTTP and HTTPS traffic. The ports used internally are 8001 and 8444 also for HTTP and HTTPS traffic which make Kong's administration APIs accessible. Through the administration API it is possible to configure object like Services, Routes, Consumers, Certificates and Plugins.

3.3.1 Load balancing

In this section, the methodologies that Kong uses to effectively balance the load of requests to multiple backend services will be discussed. Two mechanisms are used: DNS Balancer and Ring Balancer.

DNS based load balancing: In this mode, Kong does not internally record the back-end services it can reach and receives information about the services periodically from the DNS server. Balancing is done by the DNS itself in the presence of host names that correspond to multiple IP addresses. Furthermore, the frequency of updating DNS records will depend on the Time-to-Live (TTL) field: small values in this field will result in more frequent updates. As for type A records, in the scenario where hosts are resolvable to multiple IP addresses, DNS uses the round-robin balancing algorithm, given the absence of weights attributed to IP addresses. Instead, for the SRV records which also associate the port number and weight to the IP addresses, the algorithm used for balancing is a weighted round-robin.

Ring balancer: In this case, load balancing is carried out by Kong and through the definition of two entities: Upstream and Target. The new services are registered within Kong which will take care of the management of balancing. The target represents the IP / host address and the back-end port number to

which a specific weight is assigned. It can set both Ipv4 and Ipv6 addresses; the upstream is the entity that represents the set of targets to be exposed and has the task of managing the balance between the various targets. The targets can be added to the upstream simply through calls to the administration API and each upstream can therefore be associated with multiple targets. The algorithm mainly used is weighted round-robin.

3.3.2 Health Check

This section covers how Kong manages the health of its APIs. Kong uses two ways to determine if its APIs are healthy or unhealthy: the first, called active control, by periodically sending HTTP or HTTPS requests, whose responses will determine the status of that target. If the request interval is set to zero, this specification is disabled. The second way, called passive control, through the analysis of the responses generated consequently to requests from clients. The advantage of this method is that it does not generate additional traffic.

Kong uses four counters (variables) to determine whether an API is healthy or not, increased or reset after analysing the responses and based on the type of error that occurred:

- Success (increased if no errors of any kind were detected).
- TCP failure (increased in relation to the presence of connection errors).
- Timeout (increased in relation to the expiration of the timeout).
- HTTP failure (increased if the returned status code is "unhealthy").

When one of these variables reaches the configured threshold, the end point will be marked as unhealthy. Unlike active control, which has the ability to re-enable communication automatically when a target goes from unhealthy to healthy status, passive control does not have this feature and it must be the Kong administrator to make the communication with that particular target. This type of mechanism is also widely used in a cluster environment with the characteristic that the state of an endpoint is not the same for all nodes. In fact, an endpoint can be considered unhealthy for a node that perhaps cannot reach it but is still

reachable for the other nodes. The health status of an endpoint is therefore relative to the single node of the cluster.

3.3.3 Hybrid mode

Since its inception, the saving of the Kong environment configuration, that is, everything related to routes, services, plug-ins, consumers, was saved within the database. With future releases, Kong allows a hybrid structure. On the one hand, there is a control node called Control Plane, on the other there is a data plan called Data Plane. The Control Plane has the task of querying the database, where the current configuration of all services, routes, consumers and plugins is saved and distributing them to the various Data Plane nodes connected to it periodically so that all Data Planes have consistent information about the current configuration. Data Planes, on the other hand, have the task of proxy requests. These maintain a cache in which they save the freshest configuration so that, if the control plane stops working, the Data Planes would still be able to proxy requests. This type of configuration avoids continuous queries to the various instances (DP) of Kong.

3.3.4 Plug-in

As previously reported, one of the main features that allows to expand and customize Kong's features is the use of plug-ins. The plug-ins are configured by the user as an extension of the features already present in Kong. When the API gateway receives a request, the plug-in is executed. The plug-ins are codes written in Lua language, which brings that certain functionality not only to the request before being forwarded but also to the reply, if necessary. An example of functionality could be an extension of the authentication system on requests to certain APIs. It is possible to enable more than one plugin for a specific microservice. In this case, the plugins are executed in cascade, one after the other, according to the order assigned by the administrator, before forwarding the request.

Kong offers a powerful solution to manage APIs. There are two dimensions that determine a good performance by Kong: latency and throughput. Latency is measured in microseconds / milliseconds and is the time that elapses between the sending of a request by the client and the response. Throughput, measured in seconds / minutes, is the ability of the whole system to process requests simultaneously and is a size that depends on the CPU power. Kong allows to increase computing power by scaling horizontally, adding nodes to our cluster; however, it is not always the best solution especially in the presence of a latency sensitive system. The factors that determine the performance of Kong system are:

- number of routes and services (they use a greater effort by the CPU to evaluate the request).
- number of consumers and credentials (Kong stores this information in the cache so that it can be immediately found for communication. If the cache does not have enough space to store this type of information, Kong will store it in the database. This would cause greater latency given by access to the database).
- number and type of plug-ins (high number and type of distinct plugins can cause an increase in latency).
- size of the request and response.

By using Kong in the Kubernetes environment, the Routes, the Services, the Upstream and the Targets are defined through Kubernetes' Ingress, Service and Pod resources. Through the ingress objects it can be define the Kong routes, through the service objects it can be define the Kong services (in which the settings are defined, such as, for example, the protocols used during communication with the microservices) and the Upstream (they have the task of balancing the load between the various targets), finally through the Pods of Kubernetes it can be define the Targets of Kong. The figure below shows a representation of the correspondence between Kong and Kubernetes resources.

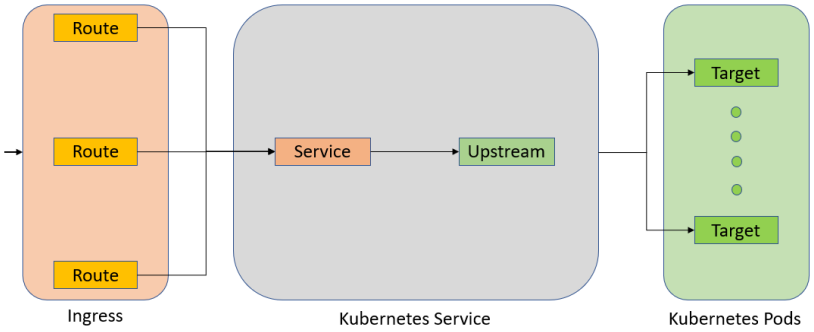


Figure 8 - correspondence of the resources of Kubernetes and Kong

3.4 REST

The REST [1] architectural style, introduced for the first time in 2000 by Roy Fielding in his dissertation entitled "Architectural Styles and the Design of Network-based Software Architectures", represents a way of communication over HTTP between systems interconnected by a network. It promotes the creation of APIs and the transmission of data. REST is an architectural style based on the concept of resource, that is information uniquely identified by a sequence of characters which, in a computer language, is called the Uniform Resource Identifier (URI). HTTP REST requests can be of various types depending on the type of action you want to perform on the resource. REST requests are in fact characterized by methods (HTTP verbs) that define both the type of request and the action. The operations that can be performed on a request via the HTTP verbs can be summarized with the acronym CRUD (create, read, update, delete). Some examples of requests are: HTTP GET to find a resource identified by a URI, HTTP POST and HTTP PUT to create / modify a resource, HTTP DELETE to delete a resource, etc. The REST architectural style focuses on six fundamental constraints:

Client-server

> *"Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve*

> *independently, thus supporting the Internet-scale requirement of multiple organizational domains." [1, p. 78]*

As Roy Fielding explains, the separation between client and server is fundamental because it allows both to evolve independently. In fact, user-side applications only need to know which interface it needs (API) to get information on the data and they can be uninterested in the way it is saved on the server. Servers, on the other hand, do not have to worry about how to provide data to clients, thus avoiding updates to servers considered as sensitive processes.

Stateless

> *"Communication must be stateless in nature, [...], such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client." [1, pp. 78-79]*

Important feature that brings advantages especially from an architecture management point of view. No type of data relating to the connection with the client is saved within the servers. Each request is handled independently of the others, making it easier to scale the infrastructure as there is no need to share session information with other instances.

Cache

> *"In order to improve network efficiency, we add cache constraints to form the client-cache-stateless-server style […]. Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or noncacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests." [1, pp. 79-80]*

Property that allows to generate less traffic and reduce latency times between a request and a response. The trend is to reuse, if possible, the answers relating to the same call. Some types of responses are saved in the application cache and reused, avoiding forwarding the same requests to the server.

Uniform interface

> *"The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability" [1, p. 81]*

As Roy Fielding explains, using REST, access to services is standardized to simplify both architecture and implementation for developers. For this reason, it also defines four important features and constraints for REST interfaces:

- **uniform interface**: means having a standard abstraction, identified from a URI, which does not necessarily reflect the resource it exhibits.
- **manipulation**: means the ability to manage resources using HTTP verbs that allow, in this case, to create new resources, modify or remove them.
- **Self-descriptive**: ability of the request to incorporate the information useful to perform the activities described above.
- **hypermedia as the engine of application state (HATEOAS)**: one of the most important features of the REST architecture that allows easy access to resources using hyperlinks. During communication with the server, in addition to receiving the requested resource (via URI), it can also receive links that allow to access all the resources connected to it. An easy way to use an application and manage access to resources appropriately.

Layered system

> *"In order to further improve behavior for Internet-scale requirements […] the layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot 'see' beyond the immediate layer with which they are interacting. By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote substrate independence. Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared*

> *intermediary. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors." [1, pp. 82-83]*

In this constraint Roy Fielding emphasizes the concept of abstraction, an intermediate layer that stands between the client requesting the resource and the place where the resource resides. The abstraction allows the client to ignore the characteristics of the resource or server that contains it since it only sees a uniform interface that allows it to find the information it needs, but sees nothing of how the system behind manages the call.

Code-on-demand

> *"REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST." [1, p. 84]*

Currently widely used constraint that allows to reduce the complexity of server-side calculation. The resource, in this case, the executable script does not reside on the client, but becomes easily downloadable. On the one hand it makes the client application simple, on the other it prevents a certain functionality from being performed on the server for all applicants, thus distributing the workload among the various clients.

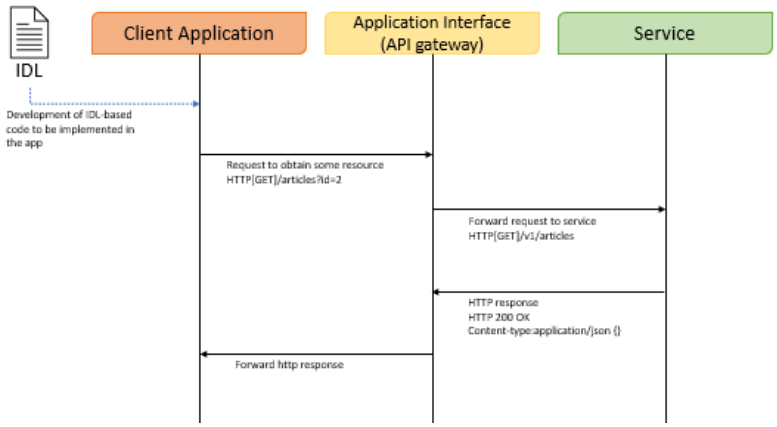


Figure 9 - example of HTTP REST communication

The diagram shows the flow of operations during a client-server communication that uses the REST architectural style. First, the developer builds the code to be inserted in their application based on the Interface Description Language (IDL), a document that describes the characteristics of the interface with which it must interact. There are two ways to develop this type of code: manual development or automatic development through document-based software (e.g. Swagger). In this case, the app performs an HTTP GET to find information about the items that have the number 2 as their identification code. The request is sent by the application interface, generally an API gateway, to the appropriate service. The answer is formed by the response code that indicates whether the request was processed correctly or if there were problems, and the resource requested by the client. In this example, the answer may also contain hyperlinks related to linked resources. Finally, application interface forwards the response to the client.

4 Problem statement

This chapter aims to address the business process that was described during the internship phase in Avio Aero and the reasons that led the company to adopt a solution based on API management. The project is called RFID Taranta and aims to ensure correct management of the various components of an aircraft engine during the maintenance phase. The chapter will continue with the description and explanation of an implementation prototype based on an API management platform, using an API gateway and tools that can lead to complete development and control of the APIs.

4.1 Avio Aero – a GE Aviation business

Avio Aero is a General Electric business company operating in the field of civil and military aviation field. It has the task of designing, creating, and maintaining the various components of air transport vehicles and their systems. The process of continuous technological innovation pushes Avio to invest in research and development to meet customers' needs. This process mainly concerns the development of technology that can lead to an increase in performance: aircraft weight, energy consumption, to name a few. Today Avio is recognized as world excellence thanks to the countless investments and collaboration with universities.

Avio has been able to keep up with innovation, and from a technological point of view has been able to exploit digitalization. *Brilliant factory* is the nickname used by Avio Aero to indicate a site, a production factory where, thanks to the adoption of new digital systems for the collection and analysis of data, try to automatically improve production efficiency. A nickname used to highlight their participation in the fourth industrial revolution [12].

4.2 Introduction to RFID technology

Radio-frequency identification (RFID) [13] is a technology used for the detection of generic objects using labels called tags or transponders. Tags are detected by devices called readers. The name reader should not be misleading since they have both read and write capabilities. The tag consists of a microchip, identified by a unique identifier,

with data storage capacity and is joined on what is called a substrate, generally made of PET (polyethylene terephthalate) or paper. The substrate is finally attached to the object to be monitored. The readers, on the other hand, are antennas capable of sending low frequency radio signals. There are different types of RFID tags depending on the type and way of power supply that the tag uses. In this context, the RFID tags are passive. The microchip that constitutes the RDIF tag has no power supply and the operation is simple: the antennas generate electromagnetic fields that allow to transfer the minimum necessary quantity of energy capable of activating the microchip. Once activated, the microchip sends the information contained within it. The same information will then be saved within a database and will be studied and used for different purposes.

4.3 Use case overview

The RFID Taranta project aims to trace all the movements of disassembled parts of an aircraft engine using RFID gates and smartphones. When an airplane engine needs maintenance, this arrives in the relevant factories, it is disassembled into the various components and for each of them a unique RFID tag is generated to be attached to the piece. This tag allows the antennas to identify all movements within the factory to be easily tracked. The reading of the labels is then saved within the database located in the Predix [14] cloud, a platform as a service (PaaS) developed by GE. The services have the possibility to access the database to retrieve the data and meet the needs of the workers. Sometimes the pieces are grouped in a single pallet and the workers have the possibility to find information about multiple pieces through a single scan and understand if all the pieces are in the right place Furthermore, if a piece is missing, the workshop worker can search for the missing piece both through the last position detected, and through audio signals that help the workshop worker during the search (for example through the emission of a more frequent sound when the worker is near the piece). Some of these services are located on-premise, others in the Cloud. Finally, access to the services takes place using mobile devices and following a registration process. To avoid using different credentials, access is allowed following a registration in the Avio system through SSO credentials.

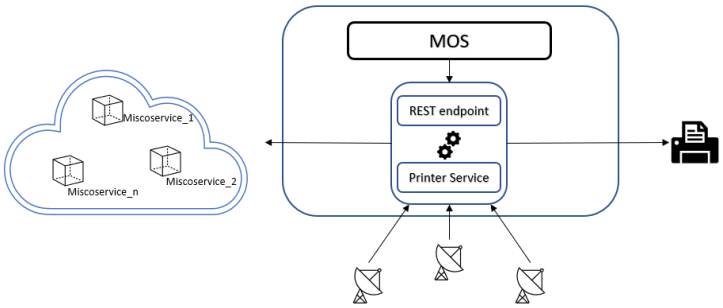


Figure 10 - high level view of the Taranta project

The main components of the system are:

- MOS: legacy system within the company where there is all the information relating to the engines and components of which it is composed. When the maintenance process begins, the MOS sends the engine component information to the printer service.
- Printer Service: on-premise backend system with RFID tag generation and forwarding functionality to the printer. It also has the task of receiving data from the MOS and forwarding the data relating to the labels to the services and database on cloud.
- RFID printer: specific printer for creating RFID tags.
- RFID gates: antennas for the detection of RFID tags.
- Microservices: on-cloud services accessible via mobile application.

4.4 Database design

The database used for saving data in this process is a relational and open source database: PostgreSQL [15], which allows the persistent saving of data. The use of a database for this type of process allows the data relating to tag readings to be saved in a consistent manner and data recovery to meet the needs of the services used by the mobile apps, accessible via APIs, and finally allows data analysis relating to data mining studies.

Internally, the tables and their relationships between them have been developed to allow mobile applications to interact efficiently. An important aspect in the design of

the tables is the absence of data regarding the engine components. As explained above, the various taxonomies are saved within the legacy MOS system.

TABLE	DESCRIPTION
Devices	List of antennas used by the project
Label	Historical association between part and RFID tag, it is generated during the printing process. Subject to variation
Position	Historical association obtained by a scanning (using RFID)
Location	List of the areas of the factory (kitting, IHR, desk repair, logistic etc)

Figure 11 - table within the database

The figure below represents the tables within the database and their connections through foreign keys. The contents of the Location and Device tables do not change. It is updated only in case of movement or addition or removal of the antennas. Generally, it remains unchanged. The Label table is updated whenever a new maintenance process occurs. During the label generation phase, these are saved within the database. Finally, the Position table is continuously updated during the detection phase by the antennas. Each detection corresponds to a save within the table and it can find all the information necessary for the tracking of the pieces. The timestamp in the position table allows for greater information accuracy.

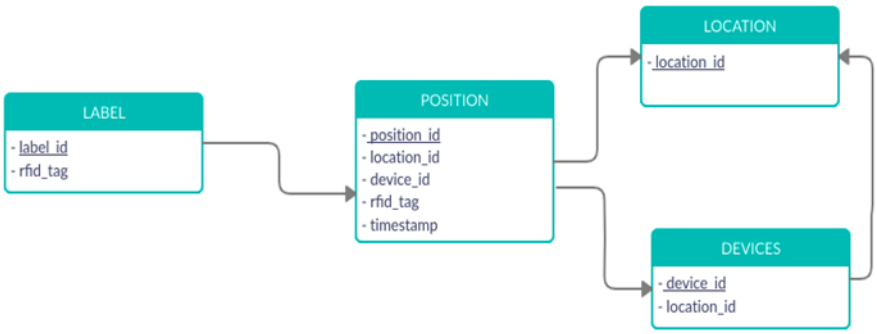


Figure 12 - representation of the relationship between tables

4.5 Printer microservice

Within the corporate network it can find a printer microservice which has the task of connecting the internal part with the external services that need the information of the labels. Note that the communication between the internal server and external services is unidirectional. When an engine needs maintenance, the legacy MOS system sends the information to the printer microservice via the REST endpoint. The printer microservice performs a dual function: it generates the labels and forwards them to the on-cloud database and sends the information of the labels to the printer after having transformed them into an understandable format for the printer. The RFID tag is identified by a 24-character alphanumeric string between A-Z, a-z, 0-9 generated within the printer microservice. In addition to the identification code, a two-dimensional QR Code is generated that allows easy reading through mobile devices.

4.6 Business evolution

The collaboration of the Avio Aero company, especially of the IT part of the Rivalta (TO) headquarters, with the Polytechnic of Turin, was born from the need of modernization, optimization of internal application development processes and improvement in the management of own services.

The company has several offices spread all over the world and often, business processes depend on data, sometimes generated in other locations. Operators from different locations therefore need data exchange, which must be as simple as possible so that the various business processes can function properly. Furthermore, in an era of microservices, the inevitable increase in services that depend on other services would risk the loss of full visibility of inter-process communications. The need therefore to have, on the developer side, a well-structured catalogue of APIs that can be easily used during the application development phase, on the manager side, an advanced control system that allows to keep under control the use of the services that the APIs expose.

Finally, corporate business processes change quickly unlike computer systems, which are often difficult to update and extend. Consequently, the company needs flexible, resilient systems that easily adapt to changes.

As explained above, communication and access to services is via APIs. In this regard, the birth of an API management project, initially made to a use case, such as the RFID Taranta project, which can lead to a simple and effective use and control of APIs.

4.7 Objectives

After understanding what are the principles that drive a company like Avio Aero to digital innovation, in this section we will try to understand what are the first objectives that the company wants to achieve through an API management solution. The main objectives can be summarized through the following points:

4.7.1 Simple access to information

One of the main objectives of the company is to make information accessible in a simple and secure way. The creation of applications that require access to information and services present in other systems is a widely used practice. APIs created for this purpose offer an effective solution, especially for those services requested periodically. The use of the same API to access the same data also allows for a substantial reduction in the number of integrations for each application development that wants to have access to the same information, thus avoiding numerous point-to-point solutions (logic of reusing an API).

4.7.2 Reduction of development times

There is a need for the company to optimize the development times of the various applications. The development process must be as simple as possible, delegating all those processes such as authentication, authorization, etc. to other systems. The developer must have as a first interest the development of the functionalities useful for that specific application, avoiding tripping over in the development of the "bureaucratic" aspects that risk postponing the delivery terms. In addition, an API-based service exposing solution allows developers to integrate existing services during their application development, avoiding creating solutions from scratch and taking advantage of existing services and features.

4.7.3 Reduction of TCO

Total Cost of Ownership is a methodology that aims to calculate "the relevant cost of buying a particular goods or service from a supplier. The concept takes into account all costs that the purchase and the subsequent use of components entail in the entire value chain of the company" [16].

Cloud computing in recent years has become of fundamental importance for companies because it allows them to be disinterested in all that part of purchasing and managing the IT infrastructure. The maintenance of the infrastructure, now known as Infrastructure-as-a-service (IaaS) is left to the service provider, who takes care of all those processes ranging from server distribution, to the management and updating of the operating system, to saving data and applications, in addition to the supply and management of the cooling system. The growing number of IaaS users allows the supplier to distribute management costs to all customers, reducing costs for each of them.

For this reason, there is an on-premise to on-cloud application migration process. Having well-documented APIs to support the application promotes a significant reduction in complexity during the migration and integration process.

4.7.4 Real-time analysis

This type of analysis is fundamental for the company because it allows to have a clear picture of the use of its services. The analyses carried out during the process of what is defined as Business Intelligence (BI) [17] concern the use of services, and therefore the number of requests per second for services, latency that occurs during the request and subsequent response, etc. The results obtained are generally represented in the form of graphs, easily understood by the human being. The graphs are in fact intended to show past and present trends by highlighting trend changes. This type of analyses helps the companies in their strategic and implementation architectural choices, allow the creation and development of scalable systems, help optimize the performance of certain processes rather than others, help detect types of problems. An API-based solution and therefore an API management platform facilitates the collection

and saving and representation of performance and usage data, thus allowing accurate analysis of them.

After an accurate analysis regarding the primary needs of the company, we can understand how, an API-based solution and consequent management platform, would be the ideal choice for business needs. The goal will be to design and then implement, under the supervision of my company tutor, an architecture that allows to manage the points described above. The study, implementation and architectural details will be described later, and applied to the Taranta RFID use case. The project aims to be a starting point for future implementations and extensions, trying to be able to cover a large part of the services and a large part of the business processes, thus promoting a continuous improvement and evolution of the company itself.

5 Proposed solution

As explained above, the solution is based on the use of the Kong API management platform, which will allow to create an embryonic architecture capable of exposing and managing the APIs and therefore the business services connected to them. The solution is mainly characterized using Containers Dockers that allow better architecture management. The containers will be deployed on a Kubernetes cluster, a container orchestrator capable of managing the various process instances and making the system scalable and resilient to various changes. Finally, the creation of a well-documented catalogue of APIs that can be consulted by the developers or anyone who wants to use the services offered by the company, in this case to internal developers and partners.

5.1 Why Kong?

The use of Kong as an API management platform for this proof of concept derives from a study of the features that this platform offers. Kong is a relatively recent project and in just a few years it has become one of the best known and most used worldwide, distinguishing itself for ease of use, adaptation to any type of system and extensibility. Kong provides two types of versions: enterprise and free. Despite the subdivision, the open source version offers a large package of features that allows to test the platform from the very first use. Among the features it can find all the fundamental ones for a gateway API: authentication and authorization protocols (basic authentication, HMAC, JWT, LDAP, Oauth2), security and access control protocols. Additional features are available only in the enterprise version, however valid alternatives are available for the open source version that Kong allows to associate. Kong offers the possibility to customize your plugins and add them through the development of scripts using LUA as a programming language, effectively making Kong very extensible. It is a light and high-performance platform. It fits easily with other types of systems, is easily scalable and supports hybrid and multi-cloud implementations. These characteristics make Kong suitable for the project of this thesis.

5.2 Architectural overview

This section tries to investigate in detail the architectural part of the project. Using containers and Kubernetes, even the instance of Kong, which in this case is called the Kong Ingress Controller, is deployed within Kubernetes in the form of a Pod. The Kong Ingress Controller application acts as an input interface for all communications with internal services. In addition, it communicates directly with the Kubernetes API server to find the information essential to its operation. Communication with the services takes place via HTTP calls, using the REST architectural style, currently used in the business process. The use of a dashboard allows to have a clear representation of the Kong Ingress Controller Custom Resource Definition (CDR) and their use.

Monitoring of system operation is possible thanks to the use of Prometheus [18] software capable of retrieving metrics from the Pods of Kong Ingress Controller. The same metrics will be analysed by the Grafana [19] software to provide a graphical representation.

About services, however, three backend applications have been created with features that come closest to the use case in question. The applications have been developed in Go [20] and two of these make use of a relational database, in this case MySQL [21] database, for saving data. Finally, a simulator consisting of two programs, also developed in Go, with the aim of making calls to internal services, stressing the system, and seeing its behaviour.

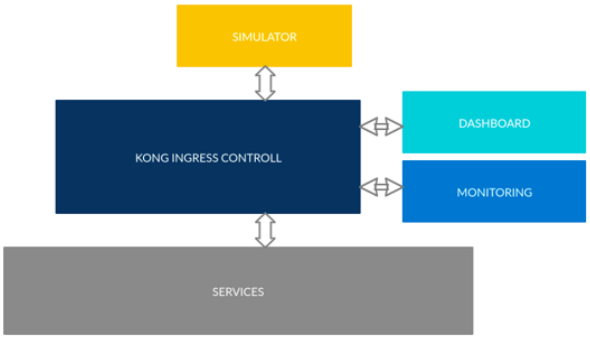


Figure 13 - architecture overview of proposal solution

5.3 Kong ingress controller in k8s

After giving an explanation about the architecture and the components an analysis follows on how Kong is used inside Kubernetes. As mentioned above, an instance of Kong is deployed as a pod in Kubernetes. The pod inside has two containers: the first, called Kong Ingress Controller which does not forward traffic to the services but has the task of finding information about the configuration and then passing the information to the second container; the second, on the other hand, has the function of proxy. It represents the input interface for all calls to all services. The fundamental information is Kong's Custom Resource Definition (CRD) defined and configured within our Kubernetes cluster. Kong's Custom Resource Definitions that extend the operation of Kubernetes will be described in more detail later.

The Kong Ingress Controller application periodically queries the Kubernetes API server to download the new configuration and update its Kong proxy Gateway. Using Kong in Kubernetes allows to take advantage of some native features of Kubernetes. In this solution it is possible to automatically scale our system by creating more instances of Kong proxy thus increasing throughput. Configuration described and implemented does not require the use of a database for saving information. Kong, however, can be used alongside a database by implementing hybrid solutions. The figure shown below provides a high-level representation of the architecture and how Kong communicates with the various components. The various instances of Kong are intended to show the scalability of the system.

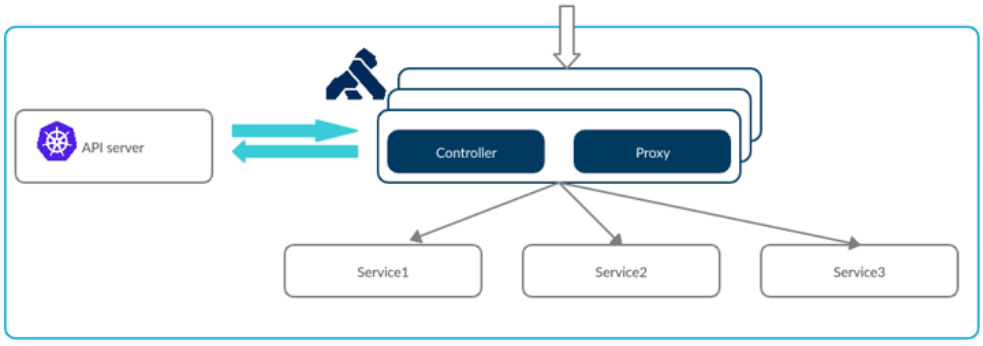


Figure 14 - high-level view of Kong on Kubernetes

Through the configuration information found by Kong Ingress Controller, the Proxy Gateway performs all the functions defined in the CRD before forwarding the calls to the service. In fact, CRDs have the task of defining the policies to be applied to HTTP requests. The policies may concern authentication, security and / or traffic control. They can also be global, and therefore applied to all incoming calls, or specific by service or by groups of services.

5.4 Dashboard and monitoring

After creating our architecture, it follows the creation of interfaces that can help to understand our system and its configuration. Having a clear vision of what happens inside becomes fundamental when, as in this case, we have many components accessible through API interfaces. The purpose, in fact, is both to understand the source of the problem when, for example, one of these components behaves unexpectedly, and to monitor usage. The Kong-based solution offers the opportunity to understand from the outside what is going on inside. The use of Kong's Enterprise solution offers custom alternatives regarding monitoring and, more generally, graphical interfaces. In this proposal solution, a dashboard was used that emulates the Kong Enterprise custom solutions to find information on Kong's CDR. It provides a graphical representation of the routes, services, plug-ins, and consumers. The specifications and detailed images of the dashboard will be commented on and explained later. About the monitoring interfaces, a solution based on Prometheus and Grafana, both open-source toolkits able to find and represent Kong's metrics and to give a graphical representation through graphs, has been adopted.

5.5 Services

The three applications have been developed to simulate and better understand the use case that has been assigned to me. They are REST servers developed in Go, an efficient and easy to use programming language. Two of these applications communicate with a database. The first application simulates respectively the APIs used by mobile devices to search for components, modify and remove readings by devices. Hence the need to communicate with a database that contains the information of the various readings.

The second, on the other hand, simulates the listening server for saving data following the readings by the antennas. Finally, the third application has the task of generating the codes following a request to print the labels. It receives information concerning a specific component of an engine and generates the identification code of that part. The proposed solution provides a relational database, MySQL, capable of saving all the information of the engine parts and of the readings made during the maintenance process.

5.6 Simulator

A simulator helps better understand how the various parts of our system work. As previously mentioned, the simulator is made up of two programs also written in Go and have the objective of generating HTTP REST requests for the various services. Since the authentication part via SSO is left to the Avio Aero system the mobile device can be used after the authentication process. Once authenticated, the device receives a token to use for requests to internal APIs. The two programs have a custom token inside which is added to the header of each call to simulate the process. The authorization phase is instead left to the Kong gateway API implemented in our architecture. The two programs make different requests. The first only makes HTTP requests with POST method on the service that allows to save the reading information within the database, simulating the RFID antennas. HTTP POST takes place every second. The second application simulates mobile devices by performing HTTP GET, PUT, DELETE to request the list, modification, and cancellation of readings. In addition, it sends HTTP GET requests for the service that has the task of generating the identification codes of the RFID tags. These requests are generated every 0.1 seconds and have the main purpose of verifying that the scaling process takes place correctly and verifying the complete functioning of the system. Below is a representation of all the components described above.

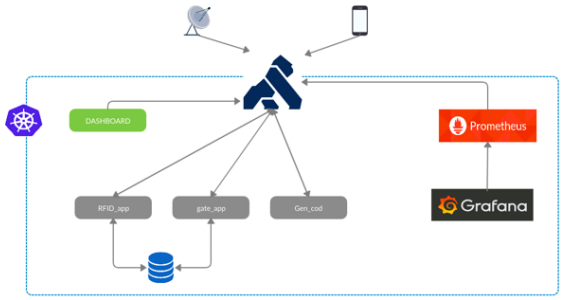


Figure 15 - communication between components of the system

6 Proof of Concepts

After describing the general structure regarding my proposed solution for the use case that was assigned to me by the company Avio Aero, this chapter aims to examine in detail the technical and implementation aspects of the proposed solution. Specifically, the first part will concern the set-up of the Kong API gateway within Kubernetes, followed by a description of the dashboard (the graphical interface that shows the Kong configuration). After an explanation regarding the implementation of Prometheus and Grafana for the graphical representation of the performance and use of our API gateway. Finally, the chapter will conclude with a quick description of the applications implemented and their use through the simulator.

6.1 Prerequisites

The following solution has been implemented on an Asus ZenBook with Ubuntu operating system (version 19.10), 1.8 GHz Intel Core i7 CPU, 512 GB SSD and 8 GB LPDDR3 RAM.

Below is the list of tools installed inside the PC necessary for the implementation of my solution:

- Docker (version 19.3.8)
- Kubectl (version v1.17.3)
- Helm (version v3.2.1)
- Minikube (version v1.10.1)
- GO (version go1.14 Linux / amd64)

The first four tools are of fundamental importance for Kong's implementation on Kubernetes. The first allows to create images and run containers inside Kubernetes, Minikube allows to create a containerized Kubernetes solution where inside there is a cluster formed by a single node, Kubectl allows to communicate with our cluster from external, Helm allows the installation of packages inside Kubernetes, finally, the use of Go for the creation of applications and the simulator.

6.2 Kong setting

After the installation of Minikube and Kubectl from the command line and after starting our cluster through the "minikube start" command, follows the deploy of our instance of Kong through the use of a .yaml file where there are the essential configurations by Kong Ingress Controller.

Inside there are Kong's Custom Resource Definition (CRD) which extend the functionality of Kubernetes. By adding resources such as KongConsumer and KongPlugin in Kubernetes, it is possible to define the respective Consumer and Plugin objects in Kong. The instance of Kong ingress controller in Kubernetes is created within the namespace called "Kong". As mentioned before, Kong ingress controller obtains the information necessary for configuring the kong-proxy instance through the exchange of data with the API server of Kubernetes. Obviously, the information about the configuration is not accessible to everyone. The information can be found after an authentication phase and after having received the appropriate permits. Kong uses role-based access control (RBAC) to find information concerning pods, endpoints, nodes, ingress resources, keys etc. So, in addition to the Kong ingress controller pod, other fundamental resources are set in parallel to perform the procedure for finding information such as:

- A ServiceAccount, associated with the Kong pod, used for saving the keys necessary for communication with the Kubernetes API server and for obtaining information.
- A ClusterRole type resource which lists the permissions and therefore the information available.
- A ClusterRoleBinding resource used as an association between the subject who intends to find the information and permissions listed within the RoleBinding.

Finally, inside the .yaml file there are the deployment of Kong-ingress-controller and Kong-proxy and the related Kubernetes service of the LoadBalancer type which has the task of exposing the Kong-proxy to the outside.

6.3 Autoscaling

The use of Kong ingress controller within Kubernetes allows to scale an infrastructure based on traffic needs, avoiding possible interruptions related to failures, bugs, or overloads.

The solution adopted, using a pod with two containers inside and without the use of the database, allows to scale the various instances of the two containers horizontally without further processes unlike the solution with database. In the latter solution, in fact, since the task of saving the system configuration within the database would be up to Kong ingress controller, in presence of scaling followed a process of election of the pod responsible for writing into the database to avoid continuous accesses by all pods which are expensive from a performance point of view.

The proof of concept in question exploits the scaling ability of Kubernetes. Thanks to the use of Kubernetes' Horizontal Pod Autoscaler (HPA) it can horizontally scale Kong's instances. This resource periodically monitors certain parameters to understand when to create or delete Pod instances. If the obtained data exceeds the set threshold then Kubernetes will take care of creating a new instance, vice versa, if obtained values are below the minimum threshold, Kubernetes will eliminate Pods as superfluous. In the proposed solution, CPU usage is monitored to understand when to scale our Kong instances.

Below, the definition of HPA used to manage Kong scaling.

```
1  apiVersion: autoscaling/v1
2  kind: HorizontalPodAutoscaler
3  metadata:
4          name: hpa
5          namespace: kong
6  spec:
7          scaleTargetRef:
8                  apiVersion: apps/v1
9                  kind: Deployment
10                 name: ingress-kong
11         minReplicas: 1
12         maxReplicas: 8
13         targetCPUUtilizationPercentage: 25
14
```

Figure 16 - HPA source code use to scale Kong instances

In this case, the Horizontal Pod Autoscaler is associated with the pod named ingress-kong that is, the one that contains the containers of Kong-ingress-controller and kong-proxy. During the monitoring phase, if the CPU usage exceeds 25% for each of the pods, a new instance will be created up to a maximum of 8, guaranteeing that at least one pod is always running. The figure shown the values of minReplicas at 1 for debugging purposes only. However, it is advisable to have at least three instances of gateway API working in parallel.

6.4 Applications

In this section we will see the applications created and exposed externally using APIs. With reference to the proposed use case, three applications have been developed with the same characteristics to applications already used in Taranta RFID processes. In accordance with the use of an API management platform, the application development will only concern the implementation of the functionalities. Authorization systems that allow to use applications will not be implemented in this development. The applications were developed using the GO programming language and with the help of tools such as Postman [22] the HTTP REST calls will be made to interact with our applications. After completing the development, Docker images of each of the applications will be created to be subsequently deployed within the Kubernetes registry.

6.4.1 Device application

The first application is the one that simulates the behaviour of the backend system used by the antennas for saving data within a database after reading the RFID tags. The antennas, in fact, make continuous REST calls using the POST method to save data within the database. The data will then be collected by the mobile applications during the research or tracking phases of the movements.

Figure 17 - structure used to represent data

The figure above shows the main structure used. Inside we find the RFID Tag code, the identification code of the antenna that performed the reading and the Area in which the antenna is located. The ID field shows a unique identification code automatically generated by the database during the data saving phase.

Figure 18 - source code used to save data within database

Saving the data is quite simple. The REST call made to the specific API is forwarded to the corresponding database after performing the unmashal of the body to retrieve the data contained. The same data will then be used in the Query for saving data within the database.

6.4.2 RFID application

The second application simulates the backend behaviour useful for mobile devices. In fact, mobile applications interact with this service during the search

or addition or removal of one or more pieces (tags) and can be considered as a middleware between the frontend applications of mobile devices and the database.

```
router := mux.NewRouter()
router.HandleFunc("/rfidapp", getPage).Methods("GET")
router.HandleFunc("/rfidapp/rfids", getPosts).Methods("GET")
router.HandleFunc("/rfidapp/rfids", createPost).Methods("POST")
router.HandleFunc("/rfidapp/rfids/{rfid}", getPost).Methods("GET")
router.HandleFunc("/rfidapp/rfids/{id}", updatePost).Methods("PUT")
router.HandleFunc("/rfidapp/rfids/{id}", deletePost).Methods("DELETE")
http.ListenAndServe(":8080", router)
```

Figure 19 - list of routes of the RFID application

Unlike the application in support of the antennas for saving data, this back-end service exposes different APIs to meet different needs. We can see from the figure above that it is possible to make HTTP REST calls with different methods. In accordance with the use case and therefore with the needs of the worker, the backend service can request information about all the readings saved on a database, it can find information about a single RFID tag to display the path inside the workshop during the maintenance process, has the ability to perform removals on a single reading or the modification of some fields. Finally, the backend service allows the creation of a reading in case of inconsistencies. The code for data management and communication with the database for the various functions mentioned takes place in a similar way to the Device application described above.

6.4.3 Gen_code application

This application has the purpose of generating the identification codes of the labels (RFID tags) which will subsequently be printed and attached to the various pieces to be identified during the maintenance process. Also, this application was written in Go and together with the others it needs an authorization phase before its use. The use of this service occurs after the request for a new maintenance. As explained in the previous chapters, the MOS system has inside information about the taxonomy of the engine. This data is sent to the code generation service within the body of a REST request and identification codes will be

generated for each of these. The proposed solution simulates this process through the function shown below.

```
const charset = "abcdefghijklmnopqrstuvwxyz" +
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

var seededRand *rand.Rand = rand.New(
    rand.NewSource(time.Now().UnixNano()))

func StringWithCharset(length int, charset string) string {
    b := make([]byte, length)
    for i := range b {
        b[i] = charset[seededRand.Intn(len(charset))]
    }
    return string(b)
}
```

Figure 20 - source code for generating the identification code

The generation of the code takes place using a random seed time-based. As shown in the figure, a byte-type variable of length "length" is created. In this solution, the length of the identification code is 15 characters. The "charset" variable allows to define the type of characters that will make up our final string. Specifically, the characters are between a-z, A-Z, 0-9 stay in line with the use case, however it can also use special characters. The Intn (int len) function returns a number between 0 (inclusive) and "len" thus returning a "charset" character. Executing this function "length" times (in this case 15) we will compose our final random string. The string, together with other information relating to the piece of the motor, will finally be printed on the label to be attached to the piece. The label will allow the tracking of the piece throughout the maintenance process.

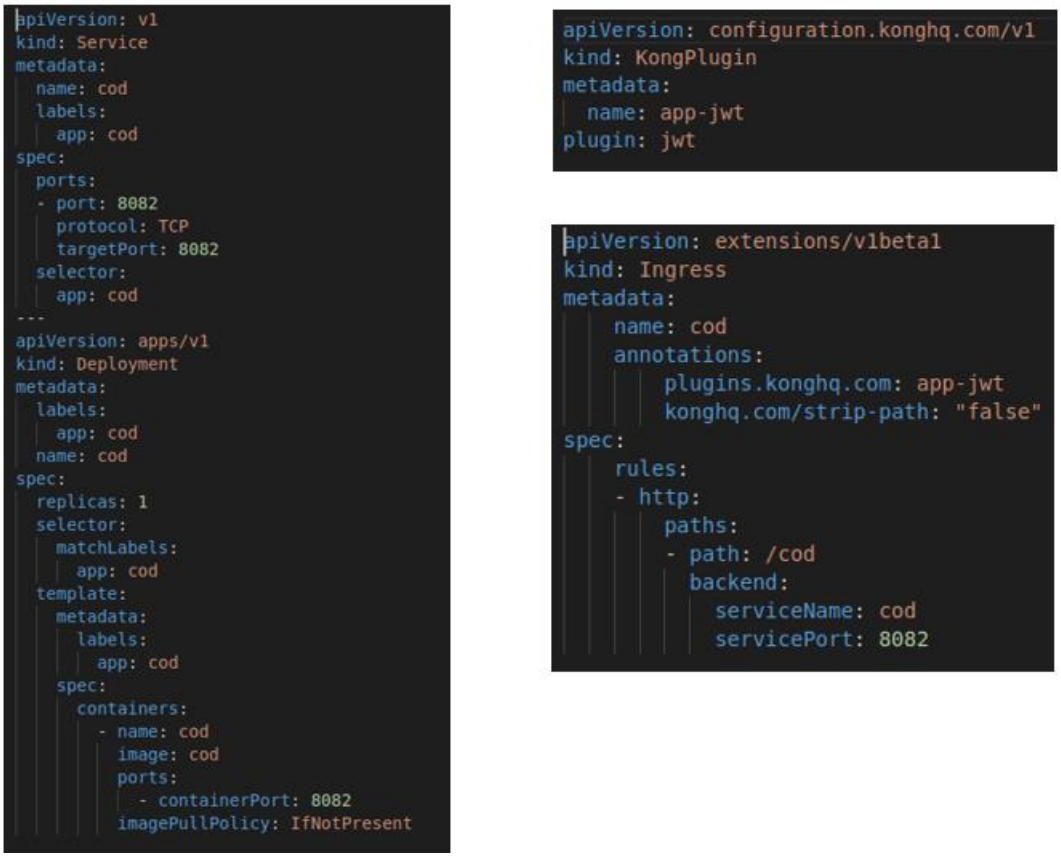
After creating the applications, an image creation process follows using Docker. The images created will then be uploaded to the Kubernetes registry to run the containers. The images are created from the Directory of our source code. Inside the directory we will create a file called Dockerfile where inside we write a series of commands that will be subsequently executed by the Docker daemon using the "docker build ." command.

Below we see the contents of the Dockerfile used for creating the images.

```
FROM golang:1.13.6-alpine3.10 AS go-build
COPY . go/src/app
WORKDIR go/src/app
RUN go install -v main/main.go
FROM alpine:3.11
COPY --from=go-build /go/bin/main main
EXPOSE 8080
CMD [ "./main" ]
```

Figure 21 - commands contained within the dockerfile

The figure shows the commands executed by the Docker daemon to create the images. Aside from the port number that the container will expose, the file is the same for all three applications. Starting from an existing image, in this case "golang: 1.13.-alpine3.10", we will copy the source code and the necessary files, then, using the RUN command, we will install the tools useful for our development. This solution adopts a multi-stage approach that allows to create a new image, copying inside it only the components strictly necessary for the execution of the application, in this way, we will have a simpler and smaller image. Once our container is started, the command to launch the application (./main) will be automatically executed. This container can be reached through port 8080. At this point, they are started using .yaml files where resources such as Deployment, Service and Ingress are defined inside.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: cod
  labels:
    app: cod
spec:
  ports:
  - port: 8082
    protocol: TCP
    targetPort: 8082
  selector:
    app: cod
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: cod
  name: cod
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cod
  template:
    metadata:
      labels:
        app: cod
    spec:
      containers:
        - name: cod
          image: cod
          ports:
            - containerPort: 8082
          imagePullPolicy: IfNotPresent
```

```yaml
apiVersion: configuration.konghq.com/v1
kind: KongPlugin
metadata:
  name: app-jwt
plugin: jwt
```

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
    name: cod
    annotations:
        plugins.konghq.com: app-jwt
        konghq.com/strip-path: "false"
spec:
    rules:
    - http:
        paths:
        - path: /cod
          backend:
            serviceName: cod
            servicePort: 8082
```

Figure 22 - source code for generating resources in Kubernetes

The figures show the deployment of an application and its connection to the Kong API gateway. The service was created as a representation of our application visible in Kong, the ingress resource was deployed, which in Kong represents the route, that is, the path that allows to reach it. Furthermore, since the development of the applications concerned only the implementation of the required features, we had to configure plug-ins that would perform the authentication process, before forwarding the calls to the applications. The KongPlugin resource is initially created which makes the use of the "jwt" plugin available, then it is associated with the Kubernetes ingress resource. Finally, plug-ins for the collection of metrics have been enabled. We will see the configuration phase of the plugin related to metrics later.

> "JSON Web Token (JWT) is an open stardard (RFC 7519) that defines a compact and self-contained way for securely trasmitting information between parties as a JSON object. This information can be verified and

trusted because it is digitaly signed. JWT can be signed using a secret (with HMAC algorithm) or a public/private key pair using RSA or ECDSA" [23].

6.5 Dashboard

Information relating to the configuration of our Kong gateway API can be found using the Admin API, generally set on port 8001. The Enterprise version already offers the necessary tools to represent Kong's resources and configurations. However, there are services with similar functionality that can still connect to the Kong API to find information and give an easily understandable graphic representation. This solution uses the Konga graphical interface, an open source tool capable of connecting to the Kong admin API and finding information such as Routes, Services, Plugins and Consumer. After downloading the image from DockerHub and uploading it to Kubernets, follows the set the settings that allowed Konga to connect to the Admin API of Kong-ingress-controller.

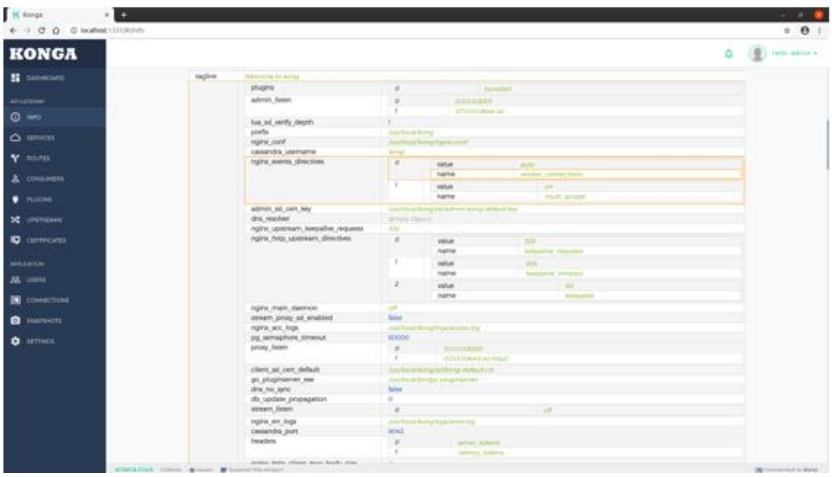


Figure 23 - Konga interface

The interface presents an overview of the Kong configuration where, in this case, we do not mean the services that the gateway API manages but details that provide us information about the node such as network information, information relating to communication with the database (if any), supported plugins etc.

On the left, we find all the information concerning the configuration of our services that we want to expose and that it is possible to manage and therefore information

about the Routes, the services, the active plugins and information on how they are used, the certificates should they be present in case of SSL / TLS communications. Finally, at the bottom right, we find information on the use of the application itself as the connected user, the setting of the connection with the Kong node, etc.

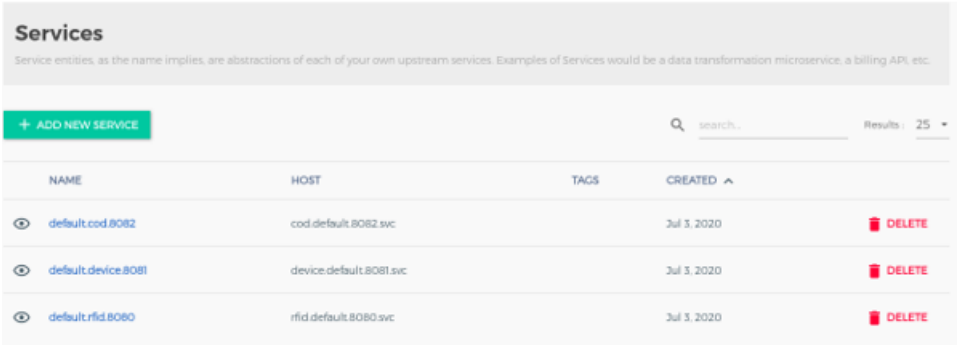


Figure 24 - information on the services used

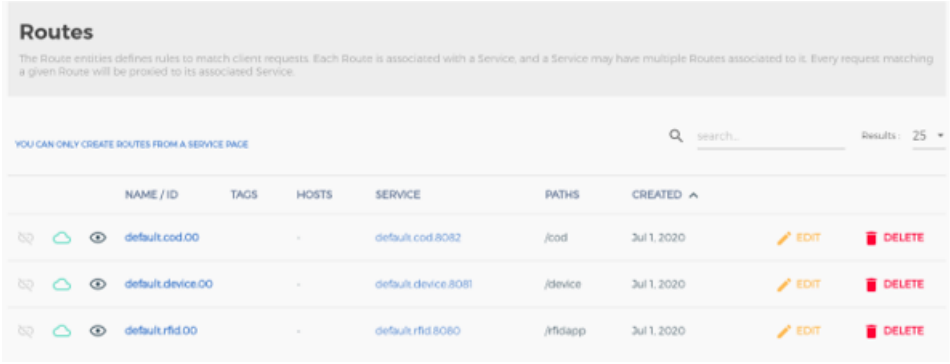


Figure 25 - information on the routes used

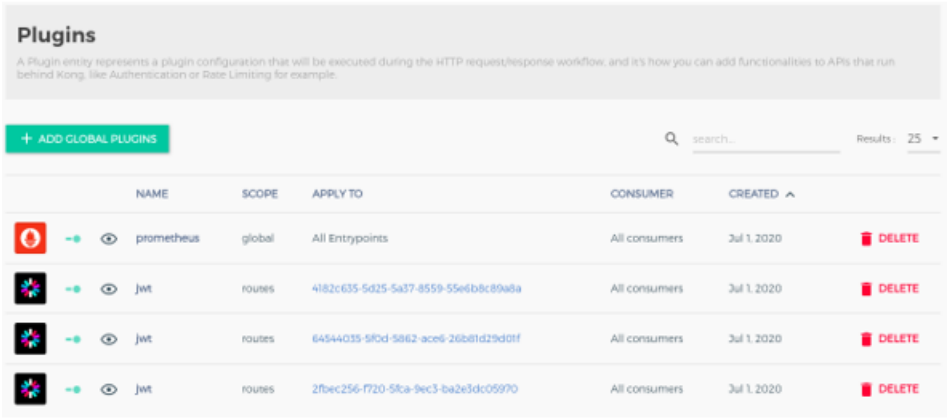


Figure 26 - information on the plugins used

The figures above show us the interfaces of our configured resources. As described above, we can see the three services related to the three applications in Kubernetes. The second figure shows us the routes associated with each service. In fact, we find the service code and the path to reach the requested application. About the plugins we can see that we have configured two different types. On the one hand we have configured Prometheus for finding metrics and as shown by the "SCOPE" field it is a "global" plugin. These types of plugins have no specific associations and are, in fact, applied on each request regardless of the route or service called. The other plugins shown are all the same but applied to specific routes. In fact, the scope field has the wording "routes" and next to it we can find the identification code of the route associated. In this Proof of Concept, plugins are applied independently of the consumers making the calls. As you can see from the figure, the "consumer" field is set in "All consumers".

6.6 API page

In line with the basic rules of good API management, after the creation and deployment of services and after exposing our APIs, the creation of the catalogue of interfaces that allow access to services follows. The catalogue represents a valid contribution to support the developer's experience (DX). The API collection is provided where, for each of them, we can find the documentation that describes the characteristics.

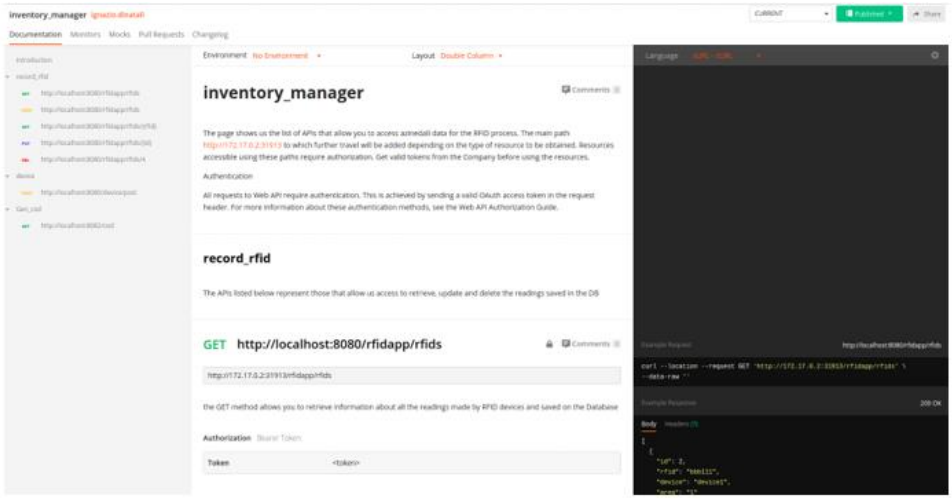


Figure 27 - representation of the API page created

The Postman open source software was used to create the API page, which supports the developer during the creation, and more generally, throughout the life cycle of an API. Postman has also been used as an API testing medium as it allows REST calls to be made by setting the header and body fields and displaying the relative response.

The **Inventory**_manager shows on the left the API catalogue relating to each service, follows a generic description on each of them, finally the details for each individual API. On the right, an example of a call where you can view both the header and body structure, followed by an example of an answer. The request format can be changed according to the developer's needs. In fact, in addition to giving the representation of a cURL request, it is possible to display the request using other programming languages such as JavaScript, Go, etc. Finally, the link at the top right allows to test the API directly using Postman.

6.7 Observation

This section concerns the implementation of control mechanisms that allow to manage and have a clear view of our system from a point of view of generic information about the use of our APIs within the Kubernetes cluster.

The use of tools such as Prometheus allow to find the metrics from our Kong-ingress-controller, while Grafana provides a representation of the metrics captured using graphs.

Kong-ingress controller supports the exposure of metrics. During the phase of setting up Kong on Kubernetes, a resource of type configMap is created. This type of resource defines independent environment configurations / variables that Pods can use internally. The purpose of configMaps is in fact not to change the basic configuration of a Pod to improve the flexibility and use of the Pods themselves. Pods that intend to use additional configurations can refer to the content of the configMaps that can be used by multiple Pods. In our use case, inside the configMap we define the useful configuration to expose the metrics.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kong-server-blocks
  namespace: kong
data:
  servers.conf: |
    server {
        server_name kong_prometheus_exporter;
        listen 0.0.0.0:9542;
        access_log off;

        location /metrics {
            default_type text/plain;
            content_by_lua_block {
                local prometheus = require "kong.plugins.prometheus.exporter"
                prometheus:collect()
            }
        }
```

Figure 28 – information to enable the use of Prometheus

The configMap is defined within the "kong" namespace. The kong-ingress deployment resource specifies the link to the configMap named "kong-server-blocks". The figure shows the configuration, in this case IP address, port and path that must be accessed to find the metrics and the type of data. At this point, Prometheus and Grafana are deployed on Kubernetes, with configurations in line with those set in the configMap. Grafana will never communicate with Kong but will query Prometheus which already has the data obtained by Kong-ingress internally. The Prometheus plugin is subsequently set globally to generate and display information about all incoming calls independent of the requested service.

```yaml
apiVersion: configuration.konghq.com/v1
kind: KongPlugin
metadata:
  name: prometheus
  labels:
    global: "true"
plugin: prometheus
```

Figure 29 - definition of Prometheus plugin

Prometheus and Grafana allow to manage metrics. The use of queries provides with the opportunity to organize information according to our needs and therefore to create graphs useful for understanding specific information. We can, in fact, be interested in all HTTP REST requests in a given time slot or we want to know the number of replies with the status value is 500, we are interested in knowing the number of requests for a particular service etc. Finally, the possibility of setting alarm systems that signal, through notifications, the presence of unexpected behaviour during the life cycle of our system.

The most important metrics for API management that Kong manages to expose through the activation of the Prometheus plugin are:

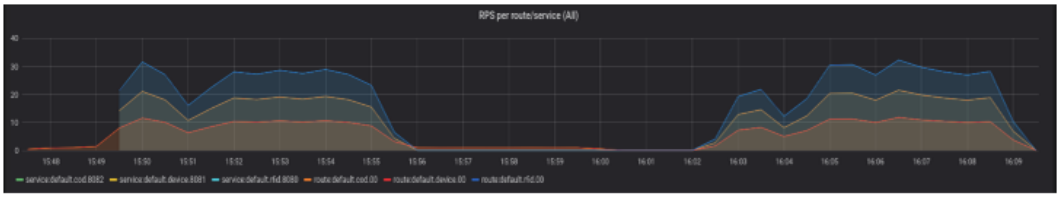
- Number of total requests per second.
- Number of requests per service and per second.
- Number of requests per second divided by status code.
- Latency between a request and the subsequent reply.
- Latency of Kong API gateway to run plugins and submit the request.
- Upstream latency
- Bandwidth.

Below the representation of the listed specifications.


```
sum(rate(kong_http_status{instance=~"$instance"}[1m]))
```

Figure 30 - total requests per second

The first graph shows a representation of the number of total requests, that is, the number of requests that Kong manages and forwards to the various services. We also find the query used by Grafana to get the information saved on the Prometheus server.




```
sum(rate(kong_http_status{service=~"$service", route=~"$route", instance=~"$instance"}[1m])) by (service)
```

```
sum(rate(kong_http_status{service=~"$service", route=~"$route", instance=~"$instance"}[1m])) by (route)
```

Figure 31 - request divided by routes and services

The figure above shows us the same graph with an additional detail. From the graph we can see the number of total requests divided by service and by route.

In this case there are two queries because a service does not always correspond to a route. It is possible to set up multiple routes for a service for various reasons, generally related to access policies. For example, the ability to access a service for free through a route a limited number of times. When you reach the maximum number of accesses you must subscribe to be able to use the service and therefore have the possibility to access an unlimited number of times. In this scenario, access occurs through a different route, in which authentication and authorization policies are performed.


```
sum(rate(kong_http_status{service=~"$service", route=~"$route", instance=~"$instance"}[1m])) by (service,code)
```

```
sum(rate(kong_http_status{service=~"$service", route=~"$route", instance=~"$instance"}[1m])) by (route,code)
```

Figure 32 - representation of requests per status code

The figure shows a very important metric for quickly identifying errors within our system. Through two different queries (for the same reason described above), we have the possibility to see the HTTP status code relating to both routes and services. Also, in this case we find an overlap in the representation of the trend between services and routes because, in the proposed solution, each service is associated with only one route.


```
histogram_quantile(0.90, sum(rate(kong_latency_bucket{type="kong",instance=~"$instance"}[1m])) by (le))
```

```
histogram_quantile(0.95, sum(rate(kong_latency_bucket{type="kong",instance=~"$instance"}[1m])) by (le))
```

```
histogram_quantile(0.99, sum(rate(kong_latency_bucket{type="kong",instance=~"$instance"}[1m])) by (le))
```

Figure 33 - representation of graphs about latency

The set of graphs above provide a representation of the latency between a request and the subsequent response, useful both to easily understand possible delays, and to

understand if our system and our services operate in compliance with the Service Level Agreement (SLA). The graphs shown can be divided horizontally, where for each column there are the subdivision: total, by service and by route. In the first line we have a representation of the latency made by Kong during the process of managing, running plugins and forwarding calls. The second line shows the total latency between a request and a response. The third line represents the latency of the services.

A special feature regarding this type of measurement is the way in which these data are interpreted. As regards the latency data, it would not make sense to reason in terms of maximum values (this particular type of service must never exceed a certain latency value). Furthermore, the calculations of the average value are inappropriate since it would not provide an indicative value other than not very reliable (the presence of a high peak of the latency value would shift the average value of all data). To have a reliable value, percentiles (p numbers) are used in this solution.

"Percentiles provide an indication of how the data values are spread over the interval from the smallest value to the largest value. Approximately p percent of the data values fall below the pth percentile, and roughly $100 - p$ percent of the data values are above the pth percentile" [24].

For example, if the percentile (p) is 90, the maximum value (in this case of latency) of 90% of my observations will be represented. Three different percentile values were used in this solution (p90, p95, p99). The use of this type of measurement provides reasonably sensible information for calculating latency. The queries shown refer to the first graph in the upper left corner. Queries relating to the other graphs are easily obtained by changing the value of the "type" field and adding the service and route fields.


```
sum(irate(kong_bandwidth{instance=~"$instance"}[1m])) by (type)
```

```
sum(irate(kong_bandwidth{type="egress", service =~"$service"}[1m])) by (service)
```

```
sum(irate(kong_bandwidth{type="ingress", service =~"$service"}[1m])) by (service)
```

Figure 34 - representation of bandwidth

The latter figure shows the bandwidth graphs used during the transfer of calls and answers. The first graph provides a representation concerning the incoming and outgoing band. It also represents general information such as maximum and minimum width, average and current bandwidth. The two graphs below provide the same information by specifying the bandwidth for each service.

6.8 PoC Simulator

The latter part is intended to test our system, trying to simulate a part of the maintenance process of the parts of an aircraft engine, paying attention to the use of APIs by antennas and mobile devices. As previously described, two simple GO applications have been developed with the aim of making HTTP REST calls to the APIs and seeing how the Kong Gateway API behaves. The applications are characterized by the presence of a "while" cycle where the requests for the various services are present. The first application simulates the behaviour of the antennas and has the task of making POST requests, where inside the body we find the information relating to a single reading. The second application, on the other hand, simulates the behaviour of mobile devices. Within the while loop we find HTTP REST requests with various types of methods. The requests allow you to make GET, POST, DELETE requests. The various requests are made every 100ms.

```
url := url_base + "/device/post"
method := "POST"
str_rfid_number := strconv.Itoa(rfid_number)
str_device_number := strconv.Itoa(area_number)
str_area_number := strconv.Itoa(device_number)
payload := strings.NewReader("{\n    \"rfid\": \" " + rfid + str_rfid_number + "\",\n
                    \"device\": \"" + device + str_device_number + "\",\n   \"area\": \"" + area + str_area_number + "\"\n}")

client := &http.Client{}
req, err := http.NewRequest(method, url, payload)

if err != nil {
    fmt.Println(err)
}
req.Header.Add("Authorization", "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ1c2VyIiwiaWF0IjoxNTI3MjQ5MDIyfQ.5wWx5kt1dlAEdbGc1jKJqFD39bhRzQh2W4JHaB5WqrI")
req.Header.Add("Content-Type", "application/json")
res, err := client.Do(req)

if err != nil {
    fmt.Println(err)
}

defer res.Body.Close()
body, err := ioutil.ReadAll(res.Body)

fmt.Println(string(body))

time.Sleep(1000 * time.Millisecond)
```

Figure 35 - source code of simulator

The figure shows an example of a request, in this case a POST towards the "device" application. The request has the aim of adding a record, a reading, within the database. We can note the creation of the payload and the use of the Bearer token within the header. As we have already seen, applications require authentication and authorization to be used.

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS
hpa	Deployment/ingress-kong	2%/25%	1	8	1
hpa	Deployment/ingress-kong	49%/25%	1	8	1
hpa	Deployment/ingress-kong	49%/25%	1	8	2
hpa	Deployment/ingress-kong	69%/25%	1	8	2
hpa	Deployment/ingress-kong	69%/25%	1	8	3
hpa	Deployment/ingress-kong	37%/25%	1	8	3
hpa	Deployment/ingress-kong	30%/25%	1	8	3
hpa	Deployment/ingress-kong	30%/25%	1	8	4
hpa	Deployment/ingress-kong	9%/25%	1	8	4
hpa	Deployment/ingress-kong	9%/25%	1	8	4
hpa	Deployment/ingress-kong	4%/25%	1	8	4
hpa	Deployment/ingress-kong	2%/25%	1	8	4
hpa	Deployment/ingress-kong	1%/25%	1	8	4
hpa	Deployment/ingress-kong	1%/25%	1	8	4
hpa	Deployment/ingress-kong	2%/25%	1	8	1
hpa	Deployment/ingress-kong	2%/25%	1	8	1
hpa	Deployment/ingress-kong	39%/25%	1	8	1
hpa	Deployment/ingress-kong	39%/25%	1	8	2
hpa	Deployment/ingress-kong	57%/25%	1	8	2
hpa	Deployment/ingress-kong	57%/25%	1	8	3
hpa	Deployment/ingress-kong	33%/25%	1	8	3

Figure 36 - representation of scaling

The simulation begins with the start of the first simulator, (the application that performs POST every second), followed a few minutes after the start of the second simulator. The figure shows the use of the Horizontal Pod Autoscaler (HPA) which allows to scale the Kong instance based on the CPU usage. The system initially has a replica that works at 2% of its capabilities. We note that the CPU usage increases

following many requests to manage. Beyond the 25% threshold we can see the creation of new replicas. In this case we arrive at a situation of stability with the presence of 4 replicas that manage the various calls. At this point the simulators are stopped. We notice a reduction in CPU usage and consequently a reduction in replicas, which becomes only 1. A subsequent start of the simulators pushes Kubernetes to a new creation of replicas until stability is achieved.

7 Conclusions

The internship in a company aimed at achieving the master's degree thesis had as its object the study and embryonic development of a solution that seeks to meet the needs of the market and the company. In recent decades there has been an unprecedented technological evolutionary process, which in many ways has put many companies in difficulty, given the use of widely outdated technologies and systems. The collaboration by Avio with the Polytechnic of Turin underlines the willingness to pursue technological innovations capable of increasing production efficiency and reducing costs. The collaboration, for the project described in this thesis, began in February 2020 and ended at the end of July 2020 with the aim of renewing the corporate IT system and, in particular, promoting simple, organized and secure access to data and functionality offered by the company especially for employees and business partners. In this regard, the study, design, and development of a resilient and scalable system capable of organizing and managing the APIs that provide access to resources was undertaken.

The solution is based on the study of the most modern technology used to deal with this type of problem: the API management platform. The technology is characterized by an API gateway that acts as an entry point towards the services to be offered, which are associated with various tools and functions for the management of the APIs during their life cycle, to allow the subscription to a microservice (developer portal), for the creation of an API catalogue and the description of the characteristics, for the analysis and monitoring in real time on the use of services. To create a scalable and resilient system, the platform has been associated with other technologies such as Doker and Kubernetes capable of managing the API gateway and microservices exposed in the form of containers. Finally, the development of web servers deployed within Kubernetes and accessible from the outside for system simulation.

The project carried out turned to be a good starting point by focusing on some services related to a specific use case. The company's goal would be to expose the technology on all the services offered. The solution adopted ensures scalability based on parameters such as the CPU. The same solution could be extended by providing additional parameters to build upon by providing more effective scaling. Finally, to

make the use of the API better for developers, it would be possible to implement a developer portal that exposes the APIs and allows subscription, as well as a blog where developers can discuss and propose advice and appropriate improvements.

8 Bibliography

[1] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," 2000.

[2] W. Santos, "ProgrammableWeb," 17 July 2019. [Online]. Available: https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17. [Accessed 7 August 2020].

[3] C. Christensen, M. Raynor and R. McDonald , "What Is Disruptive Innovation?," *Harvard Business Review,* pp. 44-53, 2015.

[4] *REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC,* 2016.

[5] *Directive (EU) 2015/2366 of the European Parliament and of the Council on payment services in the internal market, amending Directives 2002/65/EC, 2009/110/EC and 2013/36/EU and Regulation (EU) No 1093/2010, and repealing Directive 2007/64/EC,* 2016.

[6] P. Mell and T. Grance, *The NIST Definition of Cloud Computing,* National Institute of Standards and Technology, 2011.

[7] G. Hulme, "CSO from IDG," 2 July 2020. [Online]. Available: https://www.csoonline.com/article/3222095/ddos-explained-how-denial-of-service-attacks-are-evolving.html. [Accessed 8 August 2020].

[8] Available: https://www.docker.com/.

[9] Available: https://kubernetes.io/it/.

[10] Available: https://kubernetes.io/docs/concepts/overview/components/.

[11] Available: https://konghq.com/kong/.

[12] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld and M. Hoffmann, "Industry 4.0," *Bus Inf Syst Eng,* 2014.

[13] R. Want, "An introduction to RFID technology," *IEEE Pervasive Computing,* 2006.

[14] Available: https://www.ge.com/digital/iiot-platform.

[15] Available: https://www.postgresql.org/.

[16] S. Parkhi, Total Cost of Ownership (TCO), 2013.

[17] Available: https://en.wikipedia.org/wiki/Business_intelligence.

[18] Available: https://prometheus.io/.

[19] Available: https://grafana.com/.

[20] Available: https://golang.org/.

[21] Available: https://www.mysql.com/it/.

[22] Available: https://www.postman.com/.

[23] Available: https://jwt.io/introduction/.

[24] D. J. S. T. A. W. David R. Anderson, "Encyclopaedia Britannica," 03 02 2020. [Online]. Available: https://www.britannica.com/science/statistics. [Accessed 01 08 2020].