UNIVERSITY OF PIREAUS
Department of Digital Systems

Postgraduate Programme “Information Systems & Services”

MSc Diploma Thesis

Cloud gateways for heterogenous data sources

Προσαρμοστικές πύλες υποδομών υπολογιστικών νεφών για διαφορετικές πηγές δεδομένων

Author: Dimitrios-Stylianos Kakomitas
Academic Supervisor: Prof. Dimosthenis Kyriazis
Piraeus, September 2021

Table of Contents

Table of Figures

Abstract

Cloud computing has gained wide popularity during last decade. More and more organizations and enterprises are moving their infrastructure in serverless platforms in order to enjoy the benefits and the flexibility that the cloud technology provides. Serverless platforms offer the possibility to deploy and execute services as functions f(x) and compose serverless workflows. Certain types of services, e.g., analytical services, require access to data in order to operate and provide results. Accessing external data resources in serverless environments can often become a problem due to several restrictions because of platform policies that might exist. Alongside this fact, specific efficiency considerations may arise regarding the long running tasks running on the cloud.

Extracting data from different data sources is a challenging task in itself. There are literary countless external APIs, database technologies and file formats, that in order to be consumed different tools, libraries, or protocols need to be utilized.

A unified gateway component, operating as a single point of entry for the cloud platform may prove to be the solution to these above-mentioned problems. The main goal of this component is to combine and orchestrate several microservices, each one responsible to collect data from a specific data source and additionally provide data-cleansing and data-transformation mechanisms in order to produce datasets that meet the standards of the gateway's users . The collected data would be accessible via REST endpoints of a single API.

The "Cloud Gateway", as will be mentioned in the rest of this thesis, was designed, and implemented considering the problems and challenges of a production environment like load balancing, scalability, high availability, authentication, and authorization strategies etc. On top of this, state-of-the-art methodologies and tools will be utilized to achieve the goals without compromising performance.

To avoid misinterpretation, the designation "Cloud Gateway" does not refer to a gateway that combines PaaS infrastructure from different cloud providers, but to a gateway that is acting as intermediate between any cloud platform and its external and heterogeneous data sources from which data should be fetched.

Keywords

Gateway; Cloud Infrastructure; Microservices; DevOps; REST API.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor Prof. Dimosthenis Kyriazis for mentoring and offering great guidance, feedback, and support throughout the composition of my thesis. I would like to thank him for giving me the opportunity to work on such an interesting topic and expand my knowledge over these area and “state-of-the-art” technologies.

Also, I would like to thank my family and beloved ones for their support and patience through this academic journey.

A special thanks to PhD candidate George Manias for offering valuable advice in writing the thesis and to my colleagues for helping to overcome technical difficulties during the implementation part.

1 Introduction

This thesis is covering a lot of the technical aspects of gateways and gateway related technologies thus there are many references to Grey Literature e.g., blogs, documentation, tutorials, and other web resources instead of normal literature e.g., scientific papers, as it is recommended for new technologies [1].

1.1 Overview of "Cloud" and cloud computing

The term "cloud" has made its first appearance since the very early days of computing, but its usage has been escalated during the last decade. Cloud servers are servers that are located in data centers and companies or individuals that use them do not have physical access to these machines. The cloud relies on the technology called virtualization. Virtualization enables running multiple "computers" in a single physical machine, called virtual machines (VMs). VMs are isolated and do not interact with each other and work in their own sandboxes. In that way a physical server can host many virtual servers making very efficient use of hardware and allow business to find the required resources to run their applications. Cloud vendors quickly became very popular offering flexible services that can meet even the most demanding requirements.

Moreover, this technology has made fundamental changes to the Information Technology field, by changing how the software is been developed, deployed and also how services are billed. Many organizations and enterprises have already migrated to cloud, and more are expected to do in near future. Some key advantages of cloud computing include [2]:

- Lowering the costs spent on infrastructure: Typically, only a small percentage of the resources are used and high demand on resources is required only for a relatively short period of time. The cloud technology enables such dynamic resource provisioning by scaling up or down depending on the current demand.
- Faster development and deployment: The process of moving new software from development to production has become faster and more secure by using automation tools for deploying and testing before going public. The latter offers quality control and disaster recovery.
- Instant access to resources: new business and startups with limited access to resources and infrastructure. Cloud technology facilitates on reducing the upfront costs, allowing to getting faster to market.
- Supports innovation: Cloud technology removes technical barriers that are produced from the operation and scaling, so companies can focus more on developing their products further and be innovative.
- Delivery of new services: By enabling human-machine interaction with sensors and wearable devices.

But in contrast with the advantages mentioned above, there are several disadvantages and risks of acquiring cloud technology. The main risks involve:

- Security: The most important issue of the Cloud is security of data. There are profound risks of storing sensitive information like medical data in 3rd party cloud service provides. These platforms are prone to attack and even if the best security practices are followed data breaches can cause great damage to organizations [3].
- Inflexibility: Choosing and adapting a cloud computing provider, often leads to a phenomenon named "lock-in", meaning that inability of clients to change to another vendor because their dependent on their current vendor [4].
- Shortage of staff skills: Cloud environments management is complex since tools and technologies can vastly differ from one cloud provider to another leading to the need for extra investment in training and more competitive salaries [5].
- Cost for certain types of applications: The cost efficiency of cloud is based on the fact that services can utilize resources only when needed. But this is a problem for long running tasks that demand resources for long period of time.

Business wise, cloud computing is still in at an early adoption stage with studies estimating that only 10% of the workload that can be transferred to cloud have been implemented. The reasons that many companies remain sceptic is deal with the idea of storing data in remote location outside from the company premises, and others are still considering the costs that remain relatively high comparing with current solutions. After all, moving to cloud requires a digital transformation and probably rethinking the existing business processes [6].

1.1.1 Service Types of Cloud Computing

Cloud computing offers four (4) different service types each one providing different capabilities and flexibility each satisfying specific business requirements [7]:

1. **Infrastructure-as-a-Service (IaaS)**: A company can rent physical servers, VPNs or storage from a provider and build their cloud infrastructure on top of that.
2. **Platform-as-a-Service(PaaS)**: A company does not buy actual resources from a provider, but instead they pay for the resources the use. Those providers they often provide development tools and specialized technologies that are more performant on their platform. Examples of those providers are Amazon AWS, Microsoft Azure, Heroku etc.
3. **Software-as-a-Service (SaaS)**: Clients do not buy any resources or software from the provider. Instead, the buy a service that satisfies their business needs without dealing with technical issues. Billing systems for SaaS application are often some available price tiers, depending in the needs of the client. Examples of this model in Shopify, Dropbox, etc.
4. **Function-as-a-Service(FaaS)**: This is the most recent model for cloud computing, and it is often referred as serverless computing. A well-known example of this model is Google's

Firebase [1]. Clients can connect a frontend application to Firebase without requiring a backend system. It offers out of the box goods like authentication, authorization, databases and even running small pieces of software as “cloud-function” and be billed only when they use them.

1.1.2 Deployment Models of Cloud Computing:

Depending on how the cloud infrastructure is utilized by clients there are four (4) deployment models [8]:

1. **Private Cloud**: This type of cloud is often found in large enterprises when there is the need for a dedicated server and network, usually for security policy reasons.
2. **Public Cloud**: This type of cloud is managed by a vendor and many clients are hosted together in the vendor’s infrastructure.
3. **Hybrid Cloud**: This type of cloud is a combination of both of the above types. It consists of on-premises infrastructure and public cloud with the required orchestration between the two platforms. A use case for this is backup mechanisms to the public cloud in order to support the proprietary infrastructure.
4. **Multi-cloud**: This type is the deployment of a client’s services into different public cloud providers.

1.2 Microservices

Nowadays the monolithic approach on designing information systems, tends to disappear. Many organizations are starting to migrate their monolith applications to sets of modular microservices. Microservices are small and modular software services that alone satisfy a business process and are completely decoupled from each other. Microservices architecture is efficient especially for bigger and more complex information systems, since it is easier to understand and develop each business process separately and in isolation. Furthermore, it supports developing different parts of the same system simultaneously and by different teams, making the development process faster and more precise. Also, microservices architecture can help on overpassing a serious problem that arises very often in large enterprise systems and is called technical depth. Technical depth is the additional work that a system requires, because the of wrong design decisions or use of technological stack. Each microservice can be maintained, refactored and also be reimplemented without having a huge impact on the system as whole.

The Microservices architecture also has its own drawbacks. Managing and leading many developer teams can be very challenging. The classic waterfall approach in building software cannot apply in this case. That is the reason that new more “agile” techniques are used, ensuring

[1] https://firebase.google.com/

the efficient cooperation between different software teams. Deploying microservices can also be a challenging part, especially in the on first deployments. Many microservices means that many different servers, databases, protocols that have to be mastered in order to have the desired result. The complexity of this particular task is so complex that a new kind of engineers are appeared into business and are called DevOps. DevOps engineers combine software development(Dev) with IT operations(Ops), combining this new term. This new specialty is popular that in the Stack Overflow's 2020 developer survey [9] lists 3rd in terms of annual income.

Many well-known enterprises have made the transition to microservices and resulted to innovations that disrupted the business. Amazon was one of the first adapters, initially trying to improve their own codebase. Until then, different components were tightly coupled, and it was very hard to deploy new features into production. These bottlenecks led to the refactoring of the codebase following a service-oriented architecture. Amazon open the way and alongside developed products like Amazon AWS (Amazon Web Services) to help other enterprises to follow their example [10].

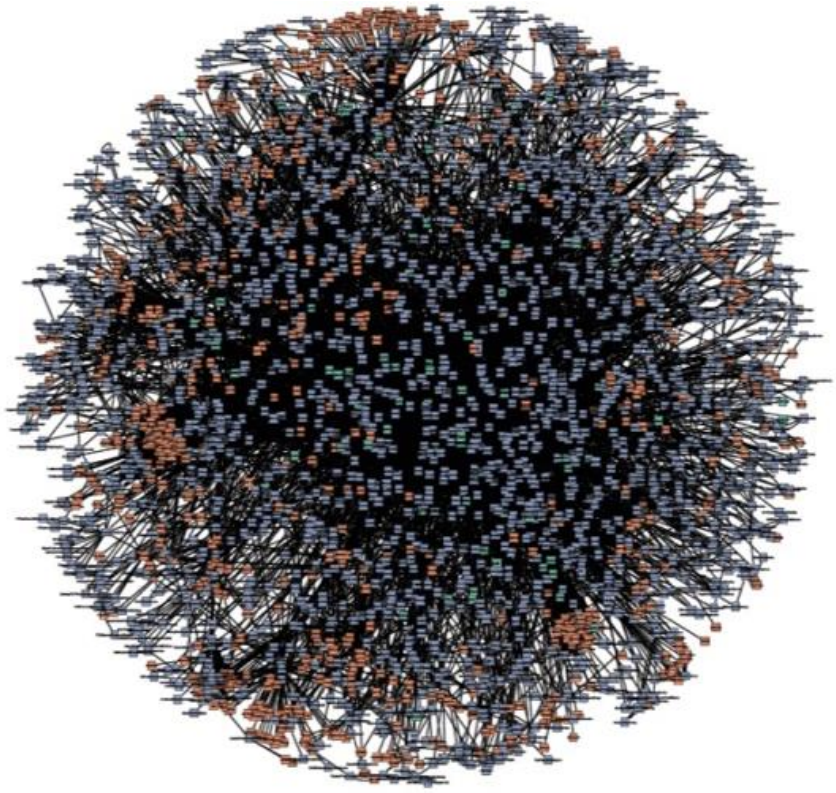


Figure 1: Amazon's microservices system representation

1.3 Problem Statement

Obtaining data from heterogenous data sources is a very challenging task. There are any different adapters, APIs that each one requires integration in order to be consumed leading to a problem

known as information overload. Since Integration with each data source consists of different technologies and methodologies, the implementation of it can been approached as a different component. This is an ideal case to follow the microservices architecture, running each component as standalone having no other dependencies.

But developing and maintaining many microservices can easily end up to absolute chaos as a network grows and new services are added, removed, or replaced. Furthermore, applying the different policies and authentication mechanisms to each microservice, clearly is not an efficient option.

Additional to this problem, in cloud environments like a FaaS platform, there are cases that strict policies and regulations does not allow access to several data sources. But although in some cases policies is not the problem, data-mining tasks like data parsing, web-scrapping are not optimal for cloud environments because are “long running” tasks, meaning that they require system resources for longer time periods, a fact contrary to the philosophy of cloud services that resources are invoked only when needed and for short periods of time. This makes such services extremely cost insufficient and hence other solutions must be found.

Thus, creating a unified a unified Gateway that orchestrates all this microservices and it is responsible for applying centralized policies like authentication, and authorization and also sharing many characteristics with cloud platforms like resiliency, scalability and elasticity but not necessarily running on cloud infrastructure but instead on a distributed network of dedicated or virtual servers.

2 Gateways

2.1 Gateways Overview

Gateways can be roughly described as the entry and exit point of a network. Gateways are designed based on the API Gateway, which is an architecture pattern that is similar with Facade pattern, a widely used term from the object-oriented programming domain. This pattern is actually an abstraction layer between multiple services and the user.

The latter implies that all incoming and outgoing traffic of a network has to be routed through it. “Cloud Gateway” is a microservices gateway connecting a cloud network with external and heterogeneous data sources, by providing services that are responsible for extracting, cleansing, transforming, encoding and storing data [11]. The gateway architecture makes the design and management of a system considerably simpler and cleaner since it abstracts microservices from their consumers and creates a central location for managing and applying policies.

To this end, Cloud Gateway offers several key features that are including below:

- Ability to serve and establish communication between different clients and services.
- Monitoring each microservice and provide uptime alerts if it is not accessible.
- Provides flexibility of adding and removing microservices.
- Allows services discoverability since all available services are registered in a central registry.
- Enables better security policies like central authentication provider and throttling to avoid misuse of the resources.

Also, a Gateway can work as an optimal location to perform data transformation tasks, in order to ensure that each service will receive data exactly in the needed form and schema and to install data analysis mechanisms since all the traffic is routed through the Gateway. To this end, the component will, also, be able to direct incoming data into the appropriate data store based on their privacy level. Therefore, it makes easy to differentiate the queries/requests having to be redirected to the overall data management, analysis, and storage system of the internal cloud platform. Furthermore, flexible schemas and metadata across multiple frameworks and sources will be defined. Finally, this component will provide the ability to design API specifications and blueprints, and aid in providing security in policy makers and managing APIs centrally.

Organizations and companies that deliver software using the microservices approach, are connecting every component using APIs.

Since integration and connectivity becomes a very important part of the business process, having a standalone API gateway component becomes more and more essential.

Moreover, developing apps in a serverless environment requires the use of APIs to provision the required infrastructure. Using an API Gateway facilitates the deployment, management, and utilization of serverless functions through various single points.

2.2 Gateway capabilities

2.2.1 Authentication and Security

Security is an essential requirement for every Information System. Implementing the authentication and authorization mechanisms in the Gateway level, is a good tactic since every API request to each microservice would require implementing the authentication mechanism for each microservice. Since the Gateway is the single-entry point of access to the system, every request can firstly be authenticated in Gateway level, and then the request will be redirected to the needed microservice. This way, the Gateway is responsible for Identity and Access Management (IAM). For implementing authentication in Gateways, Federation Identity Management is a common solution. Federation Identity relies in identity providers that can verify your identity across multiple platforms by using a set of attributes of the authenticated entity. The most common federated identity management technologies include:

- OAuth: This authentication method includes three roles, user, consumer, and service provider. The authentication flow is: 1) consumer gets the request token and secret from provider. 2) Consumer then redirects user to service provider so that the former can authorize a request token. 3) The service provider returns the token to the consumer in order to have access to restricted resources on behalf of the user.
- OpenID Connect: It involves 3 parties in the authentication process, the client, an identity provider, and the end user. The flow of the authentication process is that the client redirects the user to the provider where the user is requested to enter his credentials and to authorize access to the client. Afterwards, the authentication provider sends authorization code to the client, that it can be used for later request of authentication tokens from the provider.
- Security Assertion Markup Language (SAML): This authentication method has three actors, principal, service provider and identity provider. XML is used to pass authentication messages between the actors, that they are called assertions. This method has several disadvantages because of the XML, making it not suitable for modern asynchronous background API requests that are used commonly in single page applications (SPAs), mobile apps, smart TVs, etc.

The Gateway must provide mechanisms that protect the resources from unauthorized access and attacks. For that reason, there are several :

- Firewall that keeps a Whitelist/Blacklist for Ips that have access or not to the resources.
- Blocking requests that have suspicious headers.
- Rate limiting the requests a user can make to a specific resource for a given time frame, so to avoid abuse.
- Limit maximum number of connections for a single client.
- Limit maximum number of connections of multiple clients to a specific microservice.

It is also very important to use encryptions for all endpoints available from the API Gateway. SSL/TLS encryption can protect clients from attacks like Man-in-the-middle attack and prevent credentials leaking.

2.2.2 Data Transformation

Data transformation includes mechanisms that are responsible for checking the reliability of the data provided by performing filtering on the obtained datasets before providing them to the cloud's services. Inaccurate records or corrupt data must be removed from the datasets and incomplete data must be identified before storing inside the cloud's infrastructure. Data also must be evaluated according to privacy level. Not any data such as personal information must be stored that will to comply with GDPR regulations [2].

2.2.3 Data Serialization

Data serialization can be described as the process of transforming data into another format in order to be transmitted. When data is transported then it can be deserialized into its initial form.

2.2.3.1 Apache Avro

Apache Avro is an open-source serialization system for exchanging big data between different applications. Every Avro-encoded message includes both the definition and the data. The definition or schema is in JSON format, a feature that makes it very easy to read and understand. The data is store in binary making it very size efficient. By default, it includes markers that can help with the serialization of big datasets by splitting into subsets. The greatest advantage of Avro is support for data schemas and data types. Avro also includes APIs for many programming languages including NodeJS. These features make possible to transport data from a modern scripting language like JavaScript to strictly typed and compiled languages like C [12].

Other serialization systems include Thrift, Protobuf. The key difference of Apache Avro is the aforementioned data schema allowing full processing of the data without intermediate steps and code generation. This fact also helps with significantly reducing size of serialized data because there is no need to constantly send data schema along with the actual data [13].

2.2.4 Monitoring

As mentioned before, the fact that all traffic is routed through the Gateway facilitates the overall monitoring of the system. The latter refers in two separate types of monitoring.

[2] https://gdpr.eu/what-is-gdpr

Health monitoring: Health monitoring includes all those meaningful statistics that give us an accurate view of our system status and especially:

- resources usage (CPU, Memory, etc.)
- network usage and status
- system logs that are very helpful for possible troubleshooting
- backups and system recovery mechanisms

Traffic and Data monitoring: Traffic and Data monitoring is about collecting metrics, events, and metadata by monitoring that data flows in the system for example the API requests. Except the fact that enables generating insights via graphical interfaces, reports and alerts it also a very useful tool for finding possible policy violations, attacks, and unauthorized access to specific resources.

2.2.5 Service Registry and Discovery

2.2.5.1 Service Registry

While scaling up, new microservices have to be added in the network, to be replaced or removed. Services very often are requiring data from other services in order to perform a task. That means that by updating a service its connections should be reestablished in some way. That is the reason why, microservices instances have to dynamically assign locations. The Gateway's registry keeps track of the instances by running an internal database in which all the available services are stored. The registry is important to be updated every time there is a change in the availability status of each microservice. There are several ways that this task can be achieved:

- The Self-registration model: The service itself is responsible for ensuring the registry that is currently active or offline. In most cases the service registry offers a REST API where services can send POST requests with minimum payload, also known as "heartbeat", in order to make known that are still online. When the service registry receives no "heartbeat" from the registered services, assumes that the services are offline and flags them as unavailable. The disadvantage of this model is, despite being very simple and straightforward, it requires implementing the previously cited methodology to all services, in different languages and frameworks and also it adds some extra payload in the service registry to achieve that handshake.
- The Third-party registration model abstracts service registry to the deployment platform level. This way the registry is directly polling the development environment to find registered service instances and get updates on their status. Some of the most popular container management solutions like Kubernetes and Amazon EC2 offer service built-in service registries.

The service registry is a critical component and is important to maintain its high availability. Creating a cluster with more available replicas in case a problem occurs is a good workaround. Of course, techniques like caching the registry are totally the wrong approach

because a cached version of the registry status, may differ from its real status. Providing out of date information to the microservices and users, will trigger a series of problems.

2.2.5.2 Service Discovery

Because microservices may have multiple instances that have dynamic location, there is the need for a discoverability mechanism. There are two types of service discovery:

- **Client-Side**: In this case, clients are responsible to find the location of the instances of services that aim to use. Initially the client makes a request to service registry that returns a response with the preferred instance to use. The load balancing logic also takes part on that decision. In most cases, the service registry is updated with the status of all services and using popular algorithms like round-robin or other weighting algorithms to select the most appropriate usage at that particular moment. Client-side discovery can be performed in two ways. **1)** Active discovery: The registry makes periodic requests to all targets to obtain their status, generating some additional traffic. **2)** Passive discovery: By monitoring requests that are routed to targets, the service registry is able to distinguish healthy from unhealthy nodes based on the response. If a target is marked as unhealthy, no further requests are routed to it.
- **Server-side:** In this case there is the need for implementing a DNS server or using a 3rd party DNS service. All requests are routed to the DNS that also handles the registration of instances. Internally the DNS uses similar to client-side discovery algorithms, in order to select the most appropriate target instance.

2.2.6 Orchestration

The word 'orchestration' itself implies to a musical orchestra, a perfectly organized group of musical instruments that all together create music. In the technology word, the meaning of orchestration is similar but instead of musical instruments there are microservices, each one having a specific responsibility and purpose in order for a system to actually work meaning by accomplished the business logic [14].

Orchestration in gateway level is possible but should be avoided especially because the gateway should be considered as a component itself with specific responsibilities and by implementing orchestration at this level, the single responsibility rule is being violated. On the contrary, the progress made in virtualization technology has led to the creation of "containers" that can be really helpful for orchestrating our system. Containers are a type of virtual machine, but it bundles together all the needed parts for a microservice to be able to run smoothly.

Running different containers across multiple servers, demands offering many resources for this purpose. Orchestration can help developers to keep track of these services and containers and

also making it easier for future scaling. It also can make connectivity and data flows from one component to another and to ensure high availability of services in general.

Following the container strategy, there are many orchestration tools that you can utilize. Some examples are Kubernetes by Google, that is the most popular among the others, ECS (Elastic Container Service) by Amazon that is the go-to solution for AWS, Azure Kubernetes Service (AKS) that is a Kubernetes version optimized for running on Azure, Apache Mesos for running both containerized and non-containerized instances of services ,etc.

3 Related Technologies and Methodologies

3.1 RESTful APIs

3.1.1 History of REST

REST stands for Representational State Transfer and is an architectural style for intercommunication between client-server applications over the World Wide Web. It was defined in 2000 by Roy Fielding in his PhD dissertation “Architectural Styles and the Design of Network-based Software Architecture” .

3.1.2 REST API Design

Nowadays, the API (Application Programming Interface) is an essential part of any application because it enables the interaction with clients and the integration with other services and in many cases defines the business success and growth.

Therefore, the need for a set of best practices has emerged in order to avoid building an API that has poor performance, it is not maintainable and scalable. The most important practices are listed below [15] :

- **JSON for HTTP requests and responses:** Using a REST API requires using JSON format for both payload and responses. Data in JSON format can easily be especially manipulated and in some languages e.g., JavaScript built-in methods are provided to do so. To ensure JSON API responses the Content-Type header have to be set to “application/json; charset=utf-8” for every request.
- **Proper naming conventions:** It is very important for a REST API to have a strong and consistent naming convention strategy. When an API is named properly, is it easy to use and understand. In the opposite case, a poor naming strategy can lead to confusion and misusage of the API from its own users. A common naming convention is using nouns instead of verbs in endpoints following the CRUD (Create, Read, Update, Delete) operations. e.g., “GET”, “POST” => “/posts” and “DELETE“, “PUT” => “/posts/:id” .
- **Filtering and Pagination:** Reponses from APIs contain many records making difficult to read or most importantly cause timeouts and memory overload error. In order to maintain good performance and ensure readability and searchability of the responses, pagination and filtering must be used.
- **Caching:** Caching mechanisms can help retrieving data from local the memory instead of repeating the same queries to the server. Using caching can significantly reduce resources usage and increase performance.
- **Versioning:** Following a versioning system is important for both our system’s maintainability and for our users. The most popular versioning system is Semantic Versioning [16] following the MAJOR.MINOR.PATCH pattern.
  1. MAJOR version for backwards incompatible changes,

2. MINOR version for newly added backwards compatible functionality to the API,
3. PATCH version for backwards compatible patches and bug fixes,

Using semantic versioning the issue of what in programming world is referred as "dependency hell" can be handled. The latter implies the inability to further extend a system because an application uses many shared libraries and this library may depend on another library causing compatibility problems to the system, adding on complexity, and causing the frustration of the users.

- **Error Handling:** To eliminate the possibility of an error to bring the cause confusion and panic, we need to gracefully handle it and return well defined responses that indicated the kind of error that occurred. This allows maintainers to better understand and fix possible bugs faster. Common error HTTP status codes are:
  - **400**: Bad Request
  - **401**: Unauthorized Request – Only authorized users have access to this resource
  - **403**: Forbidden – Access to that resource is forbidden
  - **404:** Not Found – Resource not found
  - **500:** Internal server error – Generic error indicated some problem with the server
  - **502:** Bad Gateway – Invalid server response
  - **503**: Service Unavailable – Error occurred in server
- **Security Practices:** To ensure we are not exposing sensitive data in our REST APIs permission mechanisms, e.g., roles and authentication should be supported by the application. Furthermore, since the REST APIS exchange information over the HTTP protocol, SSL/TLS[3] encryption will add an extra layer of security in the communication channel.

[3] https://en.wikipedia.org/wiki/Transport_Layer_Security

3.1.3 OpenAPI Specification

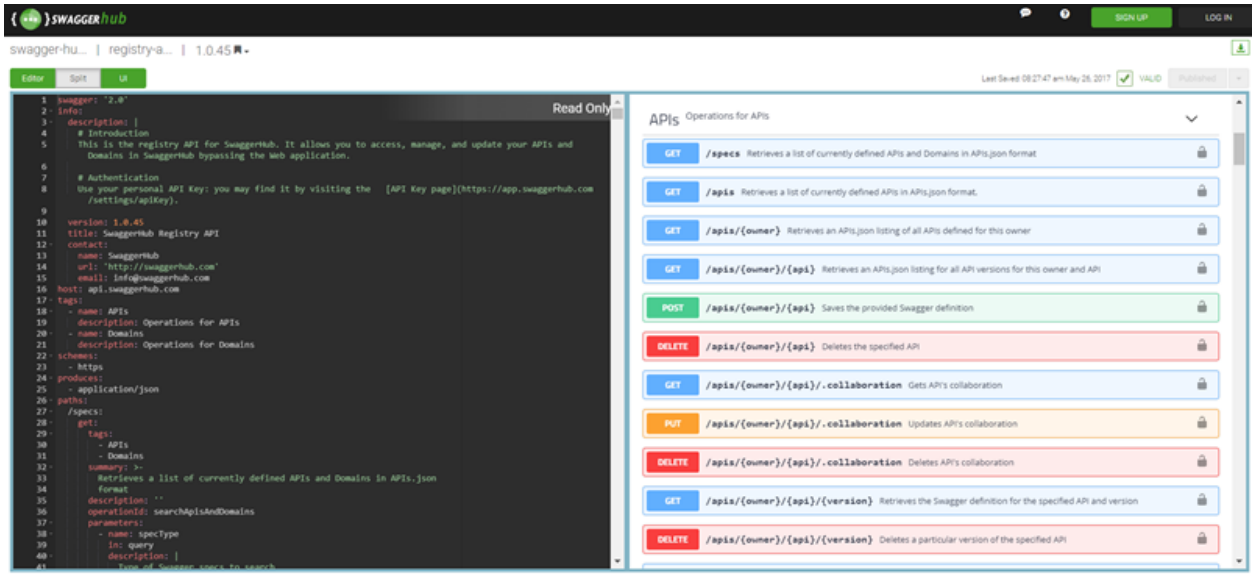


Figure 2: OpenAPI specification example

The term OpenAPI specification (OAS), formerly Swagger Specification, is a programming-language agnostic framework for describing RESTful web services. It consists of a set of rules for implementing and API that is well-defined and readable both by people and machines [17].

The OpenAPI promotes the API Drive Development, that its main concept supports that API definition must be implemented even before the development lifecycle stats. The benefits of using OAS in API development includes [18]:

- Improved Developer Experience: Developers can easily interact with the REST API and have a better understanding of its capabilities and problems. Providing good developer experience is very important factor for success especially when your product is focusing on the integration with other platforms like for example payment gateways e.g., Stripe.
- Independence between developer teams: A definition of an API help teams that work in different parts of an application, like back-end/front-end teams, to always stay aligned without affecting each other's progress.

3.2 Serverless

The term "serverless" means not using servers and was firstly referring to peer-to-peer (P2P) software. Serverless computing in the cloud context, has become very popular, and it is considered one the "hottest" topics in the IT . More and more companies choose to adapt a serverless model for their infrastructure and many new serverless providers are appearing in the market.

The term "serverless" means hiding all the server operations from developers and allowing caring only for the business logic and remove the caring about the operational hustle like

deployment, scaling, monitoring and resource management. More specifically, serverless applications are a combination of fully provided back-end solution (backend as a service – BaaS) providing out of the box features like authentication, databases, storage, and messaging system with stateless and ephemeral use of computing resources from custom applications that are invoked only when needed (function as a service – FaaS).

The billing model for serverless applications is the "pay as you go" meaning you are billed only when you are using resources. To enjoy most out of these there are quite a few parameters. Serverless should me short-running tasks because long running tasks might not be cost efficient. Serverless providers provide support for many technologies and tools. Interpreted languages like Python and JavaScript can be considered as more efficient for such environments because of the shorter latency during cold starts when comparing to compiled languages like Java or C#. Cold start is the referring to the time needed to create a new instance of the application on the stateless cloud resources. Of course, cloud functions cannot be a compared by any means to programming functions in terms of size but is a good choice for low-latency and mission critical tasks functionalities.

One serious disadvantage of serverless in public cloud is the (till now) absence of SLAs making it impossible to be used by health, government, or banking organizations. In those cases, private cloud can be used as an alternative but in this case the IaaS concerns would not be avoided.

3.3 Reverse Proxy Server

In contract with a forward proxy, a reverse proxy server stands in front of a group of server machines, intercepting all requests from clients [19].

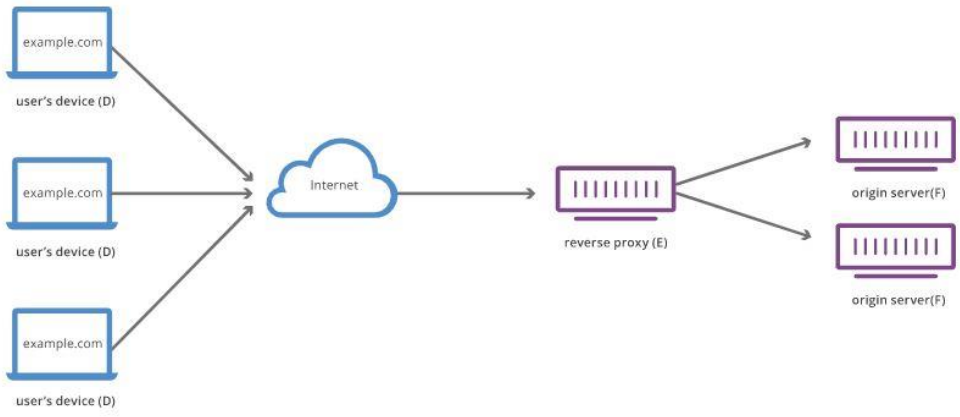


Figure 3: Reverse Proxy Architecture

Some of the most important reasons for using reverse proxies are including:

- Protection from attacks: If a website is using a reverse proxy, it is much more difficult for attackers to trace its real IP and perform attacks like DDoS (Denial of service) attack or attract spam bots.
- Caching: It is possible to cache content since all traffic is handled there. Users can access a website from a proxy server that is closer to their location than the origin server. All other users that try to access the website from the same location will view the cached version of the website from the proxy. That will benefit the performance and remove some serious load from the origin servers.
- Load Balancing: A website with many visitors is usually served not by a single machine but by a group (pool) of machines. A reverse proxy server can act like a load balancer receiving all requests and simply distribute them among the available instances, maintaining the good overall performance of the website.
- Global Server Balancing: To reduce slow load times, that originate from the distance between the client and the origin server, placing servers around the world and having proxy servers to distribute the request to the closest location, can have a positive impact in performance and it is very important for mission critical systems.
- Encryption: SSL/TLS encryption and description can be performed in the proxy server instead of the origin server, removing in that way a serious load from the origin.

3.4 Proxy Server

A proxy, web proxy or proxy server is a computer that stands in front of a group of client machines. All requests made from clients are coming through the proxy server before reaching the internet.

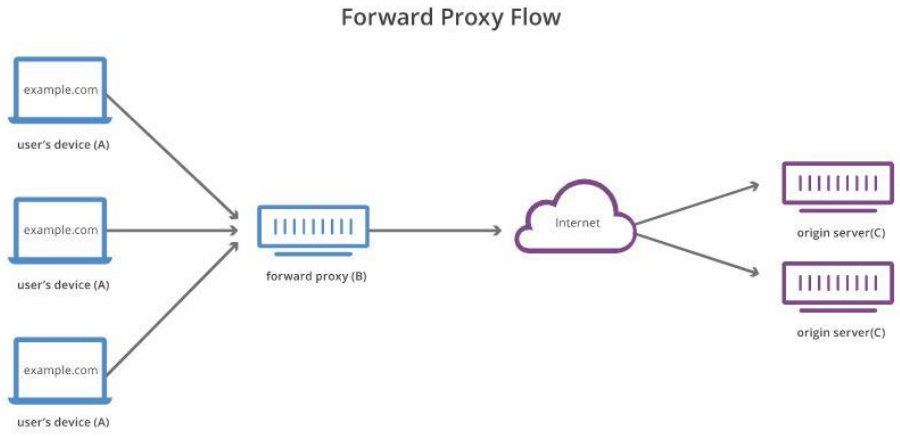


Figure 4: Proxy Architecture

In a typical internet communication example, a user would communicate directly with an origin server. In the case presented in figure a proxy server is in place acting as a middleman and

controlling all incoming and outcoming traffic. The most important reasons proxy servers are used are:

- Blocking access to certain content: Restricting certain user groups from accessing online content is usually applied to school networks.
- Identity protection: Using proxy server to access the Internet is considered an identity protection measure. It is much harder for the clients to be tracked back since their IP is not appearing in the server access log, but instead the proxy's IP is visible.
- Overcome restrictions: In some cases, there are certain restrictions in Internet usage by government, organization, etc. Clients can overcome those restrictions by using a proxy server.

3.5 DevOps

DevOps is a relatively new IT field that can be generally described as the mix of development "Dev" and operations "Ops". It includes all essential processes for development and deploying software in production environments. DevOps engineers are using automation tools and scripts to ensure the fast, stable, and risk-free deployment of new software. Also, scalability when required, and overall monitoring of the infrastructure of the organization. Migrating legacy software to the cloud is also a complex process that a DevOps engineer could be responsible for.

DevOps is not just automating the deployment of an application following the old monolithic approach. Instead, the importance of DevOps is obvious when different teams develop different software components using the microservices architecture and interconnecting every component with an API [20]. Continuous integration and continuous delivery (CI/CD) are very important for the business processes and can add great value for clients from early stages of software delivery.

3.5.1 Docker

3.5.1.1 About Docker

Docker is a virtualization platform that that gives the ability to run applications in isolated environments called containers. Each container contains all necessary software to run and application and a be very size efficient. Multiple containers can be running simultaneously on the same host. It is written in Go programming language and the underlying technology that uses to provide containers is called namespaces.

By using Docker in your workflow, you can dramatically speed up the development progress and deploy faster and safer in production. One other advantage of using containers is the ability to share between developers exactly the same developer environment

3.5.1.2 Docker Terminology

Image	Image is a set of instructions used to create a container. Images can use other images for example NodeJS can use alpine Linux as its base image. Images are immutable once there are created. Uses can create their own images and even publish it in a registry making it public for other to use. Images are stateless.
Container	Container is a running instance of an image. I can be fully controlled by the Docker API or the command line. Containers are stateful meaning that files, configuration settings etc. are being stored until the container is deleted
Dockerfile	Dockerfile is a document that contains the commands for building an Image.
Docker-compose	Docker compose is a tool for building and running complex applications consisted by many services hosted in different containers. It uses a single YAML file named docker-compose.yml to configure all services .
Docker-Desktop	Docker Desktop is a user-friendly application for Windows and Mac that allows creating and controlling containers by using a UI interface. It comes with docker-engine, docker-compose, Kubernetes. It also provides the ability to download certified directly from Docker registry.
Registry	Docker Registry, named Docker Hub, is a registry where everyone can download or upload Docker images from public or private repositories.
Volume	Volumes is a docker native mechanism to persist data of containers. It is the recommended storing mechanism. [21]
Docker daemon	Docker daemon or dockerd, listens for requests to Docker API such as docker commands, and manages docker objects.
Docker objects	Docker objects is referring to all Docker related entities like containers, images, volumes, networks etc.

3.6 Event Streaming

Event streaming is the process of capturing real-times data from different kind of sources like IoT devices, sensors, databases in the form of events. An event in, business terms, is something that happened in an environment of observation. Some real-life use cases of using event streaming are:

- Automobile: Tracking of vehicles in order to reveal its exact location at any moment
- Healthcare: Real time notifications of patient's condition and alerts in case of emergency incidents.
- Banking and Trading: Real time processing of transactions, and alerts on stock value changes.
- Meteorology: Capturing real-time events from meteorological stations and sensors in order to provide forecasts and alerts in case of extreme weather conditions.

3.6.1 Kafka

Apache Kafka is a distributed streaming platform. Kafka main concepts are the "producers" that are the applications responsible for producing events to Kafka and "consumers" that listen for those events. Consumers and producers are functioning totally asynchronous and independently giving Kafka the ability to scale.

The Kafka broker is responsible for enabling the communication between producers and consumers. [22]. The received messages from producers are stored in disk with a unique key and also enables consumers to retrieve messages by topic, key, or partition.

Kafka groups events in "topics". A producer can be configured to send new events to a particular topic and respectively a consumer to listen to it for updates. One topic can be connected to many consumers and many producers. There is configuration available to store events for a selected period of time and not delete them right after consumption. Also, the consuming frequency can be configured.

In order to install in production, Kafka is requiring Zookeeper as a dependency (from Kafka version 2.8 that won't be necessary, but this is not a stable update [23]). Zookeeper is an open-source software distributed under Apache license. It allows synchronization between Kafka nodes and also allows monitoring topics and partitions. One other responsibility of Zookeeper is maintaining the relationship of leader-follower in nodes, so whenever a leader node fails, all other nodes and replicas has to be informed and new leader node is selected by polling between the available nodes. [24]

4 Cloud Gateway Implementation

4.1 Architecture

The Cloud Gateway has been designed and implemented following the microservices architecture. It consits of five (5) basic components:

1. API Gateway Micrsoservice: It is the most important component of Cloud Gateway. It is responsible for essential processes e.g., routing, applying authentication, middleware, serving the administration and monitoring panels, service registry and discovery etc.
2. Storage Microservice: Receives all data obtained by the other microservices and persist them in the database.
3. Twitter Microservice: Responsible for obtaining data from Twitter using Twitter's REST API and streams.
4. GTD Microservice: Responsible for obtaining and parsing a global terrorism dataset in CSV format.
5. Water Quality Microservice: Responsible for obtaining and parsing a water quality dataset in XLSX format.

Additional compononents that support the operation of Cloud Gateway are incuding:

- Kafka: Used for message streaming of data between data microservices and the storage microservice.
- MongoDB: The database used for persisting data from external sources.
- Keycloak: Serves as the authentication and authorization mechanism.
- Postgres: The database used by Keycloack for storing users, realms etc.
- Traefik: Used as the Cloud Gateway's reverse proxy and load balancer.

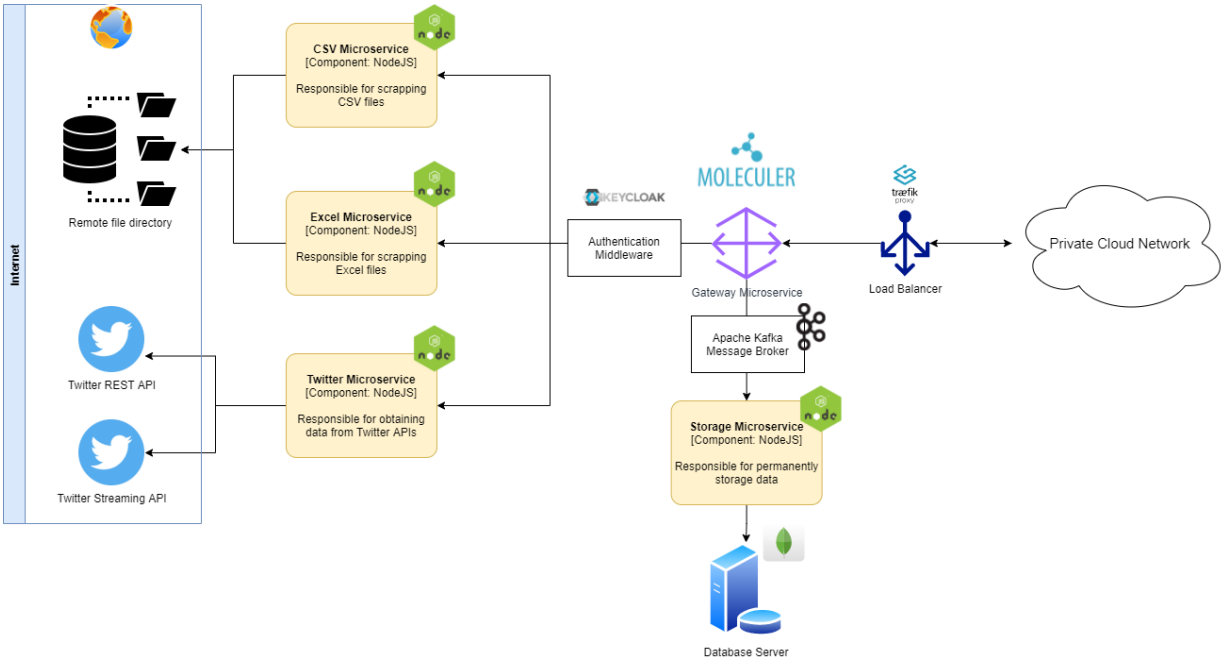


Figure 5: Cloud Gateway Architecture

4.2 Initial approach and considerations

The initial approach was to build the Cloud Gateway using a non-gateway specific frameworks LoopBack 4 [25] . Each component was built in a different technological stack. Even though, the microservices architecture was used, the development progress of every component started to become more difficult leading to bulk containers with all the different bootstrapping for every framework used. The biggest problem appeared in the deployment stage with slow build times and constant misconfigurations between the components, and also complex Dockerfiles in order to setup each container. After researching market trends [26], microservices oriented frameworks seemed like a better solution for building the microservices gateways than more generic frameworks.

4.3 API Gateway Component

For the main gateway component, the MoleculerJS framework is utilized. MoleculerJS is a lightweight Node.js framework oriented in building and managing microservices. It provides many out-of-the-box features that make the development process faster easier.

NodeJS is built with V8 runtime engine and can be very efficient for input-output (IO) heavy tasks but for CPU bound tasks like calculations, have high demand on resources. For IO-bound tasks that consist of the Gateways main functionality NodeJS is an excellent choice. Along with a very

large community, many innovative enterprises add NodeJS to their main technology stack as it increases productivity and offers high performance at a much lower cost.

In terms of performance, moleculer is considered one of the fastest Node.js frameworks according to benchmarks measuring all critical parts of the framework.

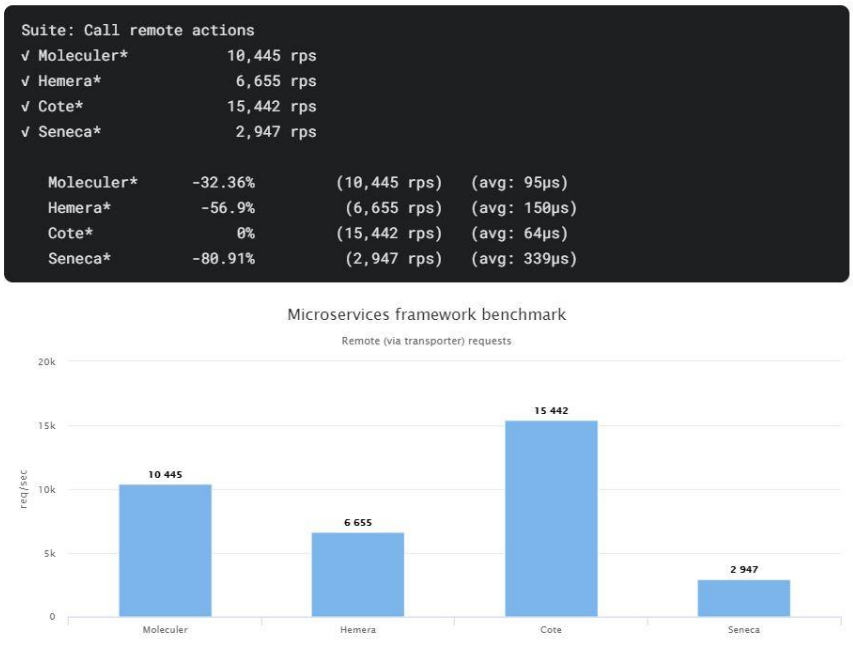


Figure 6: Microservices Comparison

4.3.1 Service Discovery

In a microservices environment the running instances of services dynamically change location inside networks. In order for client or services to be able to make requests to a service it must use a service-discovery mechanism. MoleculerJS provides a bult-in module to handle service discovery but also to handle service discovery and registry but also supports integration with 3rd party in-memory data stores like Redis or etcd3.

For reasons of simplicity, the Cloud Gateway uses the local driver that may be very easy to configure and faster since it all happens in a local scope, but it comes with serious drawbacks like affecting request load time In production environments that are consisted by a large number of nodes, it is not recommended to use that option since it uses the already existing transporter module (in our case simple TCP) that serves all the requests and adds extra load.

The underlying concept of service discovery is the exchange of heartbeats packets between the registry and the available nodes, to list the working services. If a node fails to broadcast a heartbeat is not used to serve requests made for this particular service.

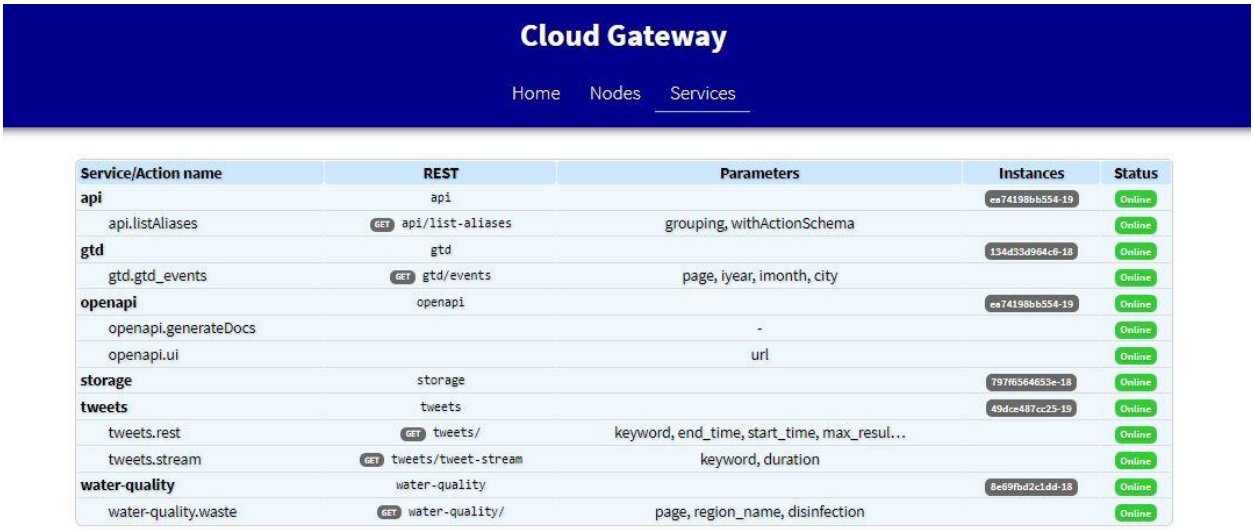


Figure 7: MoleculerJS - Service Discovery & Service Registry

4.3.2 Monitoring – Metrics

Monitoring is a very important aspect when developing microservices in order to ensure that all exposed APIs are working as expected without errors and also get metrics for API calls to ensure that your infrastructure can handle the incoming traffic load or to detect attacks, etc. For the Cloud Gateway monitoring we have utilized swagger-stats, a Node.js library that collects metrics and traces API requests. [27] Metrics are in Prometheus format, so that enables integration with other popular metrics and alerting systems like Prometheus [28] and Grafana [29]. Also, further integration is supported for in depth analysis of the requests using Elasticsearch [4] and Kibana[5].

[4] https://www.elastic.co/elasticsearch
[5] https://www.elastic.co/kibana/

Figure 8: Swagger Stats UI-Summary

Some very interesting and important metrics that are provided include:

- CPU and memory utilization
- Error logging and latest error that occurred
- Request tracing and long request tracing with details request details like headers, parameters, and times
- Statistics and summaries, overall payload measurement during periods of time
- Timelines to help you analyze trends of each request and peak periods
- Bult-in Telemetry UI with minimum configuration and many settings that provides good user experience.

For utilizing swagger stats for Cloud Gateway, the npm package @slanatech/swagger-stats [6] was installed, and in order to collect metrics it was configured as middleware on every request in the gateway microservice.

[6] https://swaggerstats.io/guide/#installation

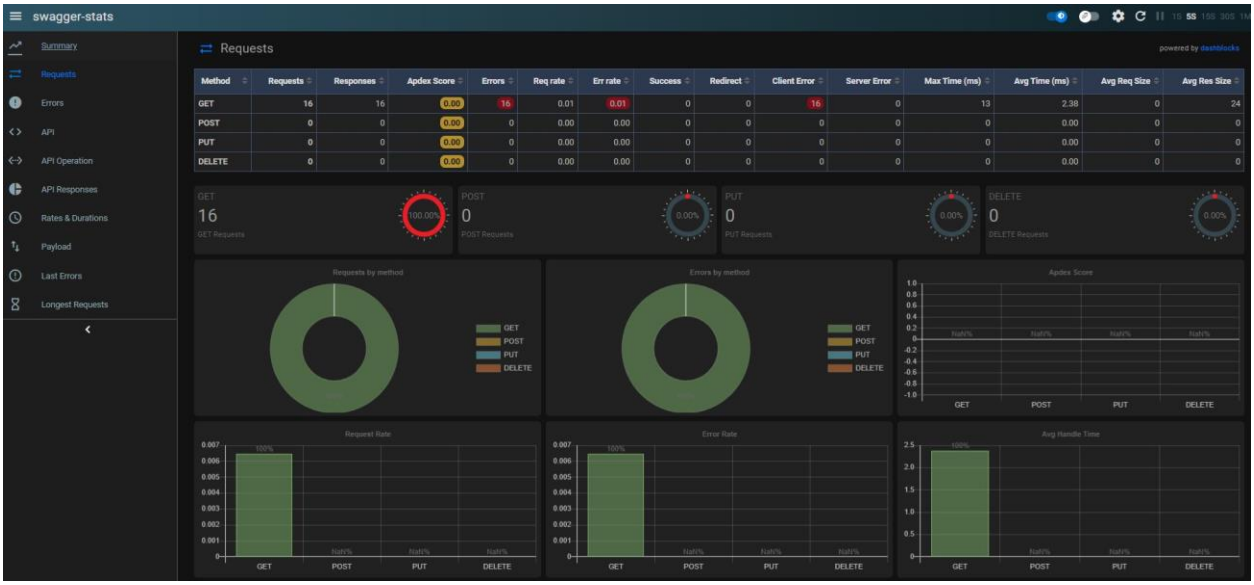


Figure 9: Swagger UI-Request and Errors

Configuration for enabling Prometheus metrics on MoleculerJS requiring adding the following code to the moleculer.config.js file:

```
metrics: {
   enabled: true,
   reporter: {
      type: "Prometheus",
      options: {
         port: 3030,
         path: "/metrics",
         defaultLabels: registry => ({
            namespace: registry.broker.namespace,
            nodeID: registry.broker.nodeID
         })
      }
   }
},
```

Afterwards, in order to use swagger stats in the api.service.js file it is required to add swMiddleware in the routes section.

```
module.exports = {
   name: "api",
   mixins: [
      ApiGateway,
   ],
   settings: {
      port: process.env.PORT || 3000,
      ip: "0.0.0.0",
      use: [
         swMiddleware
      ],
```

```
    routes: [
       {
          path: "/api",

          whitelist: [
             "**"
          ],
          use: [
             swMiddleware,
             ...keycloakMiddleware,
          ]
```

4.3.3 Fault Tolerance

In a multi-node microservices environment, there is the possibility for every request to fail for various reasons like networking problem, timeouts etc. In order to handle these errors ensure service resiliency, there are several strategies available. If the reason of failure is a connectivity issue, it is possible that the Retry Pattern [30], meaning the practice of repeating the same request is the most suitable. But the reason of failure is caused by a more serious problem, by repeating the same request over and over we increase the load to the services, utilize crucial resources like memory and CPU, until the requests timeout, a scenario that would possibly affect other working microservices. A suitable strategy for this problem, and a fitting solution for Cloud Gateway implementation, is “The Circuit Breaker Pattern”. This pattern simply abandons the retry attempts of a more liking to fail request, by running as a proxy on a retry pattern counting the failures and cutting the circuit when the number exceed a specified limit. [31]

In Cloud Gateway implementation the "Circuit Breaker Pattern” is used along with other precautionary measures. MoleculerJS has built-in configuration for setting up the preferred fault-tolerance solution or a combination of them. The configuration for the circuit-breaker requires adding the following code in the moleculer.configuration.js file:

```
circuitBreaker: {
   // Enable feature
   enabled: true,
   // Threshold value. 0.5 means that 50% should be failed for tripping.
   threshold: 0.7,
   // Minimum request count. Below it, CB does not trip.
   minRequestCount: 10,
   // Number of seconds for time window.
   windowTime: 60,
   // Number of milliseconds to switch from open to half-open state
   halfOpenTime: 10 * 1000,
   // A function to check failed requests.
   check: err => err && err.code >= 500  && err.code >= 401
}
```

The “check” value of the JSON object, accepts a function that declares all the error that indicate a non-working service and responses with these codes are considered as failures.

Furthermore, a good practice to set timeouts for requests. A long running request in most cases indicates a problem and in order to prevent utilizing resources for a long period of time it is reasonable to have a relatively small timeout. in configuration for setting up the preferred fault-tolerance solution or a combination of them. The configuration for timeout simply requires adding the timeout time in milliseconds in the moleculer.configuration.js file:

```
requestTimeout: 30000
```

4.3.4 Load Balancing

In distributed systems that a single service is server by many different nodes, in order to effectively use the available processing power and equally distribute the workload, a load balancing technique is required. There are several different algorithms for this purpose, mainly divided in two (2) categories [32]:

- Static load balancers: Performance of processors is determined before the execution. There is a master processor, responsible for allocating tasks and slave processors that calculate their load and submit back to master. The main advantage of static load balancers is the small delays in communication because the task-allocating node is predefined from the beginning although the static scheme may not be optimal.
- Dynamic load balancers: The work distributed to processors is defined during runtime. There is also a master node responsible for distributing tasks and slave nodes that are reporting their current load.

Since all nodes on Cloud Gateway are supposed to have the same available resources, load-balancing is implemented following the “Round-Robin Algorithm” strategy. All workload is distributed equally between the nodes, always respecting the threshold for maintaining good performance. Extra communication between nodes regarding the current status of the nodes is not required.

Load balancing is part of MoleculerJS registry module. It supports a few load balancing techniques including “Round-Robin”, add in addition there is support for custom made load balancers. In order to activate “Round-Robin”, the following code in moleculer.config.js file is required [33]:

```
// Settings of Service Registry. More info:
https://moleculer.services/docs/0.14/registry.html
registry: {
   // Define balancing strategy. More info:
https://moleculer.services/docs/0.14/balancing.html
   // Available values: "RoundRobin", "Random", "CpuUsage", "Latency", "Shard"
   strategy: "RoundRobin",

},
```

4.3.5 Caching

Caching in microservices is important, especially when good performance is required. By using caching mechanisms, when a client makes a request to access data, the response is stored in temporary locations for a pre-selected amount of time (TTL: time to live) [34]. When a client makes the same request inside the lifespan period, the cached response will be received without reaching the origin server. That is a common mechanism used in almost every modern service served over Internet.

MoleculerJS offers out-of-the-box caching solutions using either "In Memory" caching of 3rd party caching solutions like Redis or LRU cachers [35]. For simplicity reasons and since 3rd party solutions is a concern in production and multi-node implementation, we are using "In Memory" caching configuration with the TTL configured at 1000 seconds since the data-refreshing process is a scheduled task meaning that responses will stay the same for a long period of time. There are options available to clear caches after a data-refresh, so there are not concerns about sending the latest data version with every request.

The configuration for selecting a caching mechanism is simple and only requires the following code in the moleculer.config.js file.

```
cacher: {
   type: "Memory",
   options: {
      ttl: 1000
         }
},
```

4.3.6 Transporters

The service registry in order to work correctly requires the exchange of status information between the nodes. This requires a transporter module responsible for operating a channel of communication [36].

MoleculerJS provides support for some popular message brokers like NATS, Redis, MQTT, AMQP and Kafka. For Cloud Gateway implementation the message broker of choice is Kafka since it is also utilized for another component (see section 4.3.5). For configuring Kafka as message broker, the following code is required in moleculer.config.js file:

```
transporter: {
   type: "kafka",
}
```

4.4 Twitter Microservice

Twitter is one of the most popular microblogging and social networking platforms with a total revenue up to $1.19 billion [37]. Every day, million users produce over 500 million tweets per

day, resulting in a huge database widely used for data mining, surveys, sentiment analysis, market research etc. [38]

Furthermore, Twitter is launching the API v2, a new improved version which promises to provide a better developer experience by giving access to a wide variety of data sources and tools. Twitter assures the quality of its data by applying spam filters, access to all results of a query and not only to a partition of results, user-friendly and simplified JSON objects, shorter URLs and OpenAPI specification to test endpoints and watch for any changes. [39]

Twitter Microservice is a component of the Cloud Gateway, providing access to Twitter data to the gateway's clients without the need to directly connect to Twitter API. It has been implemented in NodeJS utilizing the twitter-v2 npm package[7].

The microservice has two (2) basic functionalities:

1. **Searching and filtering tweets**: By utilizing the Search Tweets endpoints [40] of the Twitter API v2, Cloud Gateway's users have access to the most recent tweets. The filters that can be applied include:
    - Keyword search
    - The start and end time parameters to limit tweet results to a specific period of time

[7] https://github.com/HunterLarco/twitter-v2

- Max number of tweets in order to limit the results returned

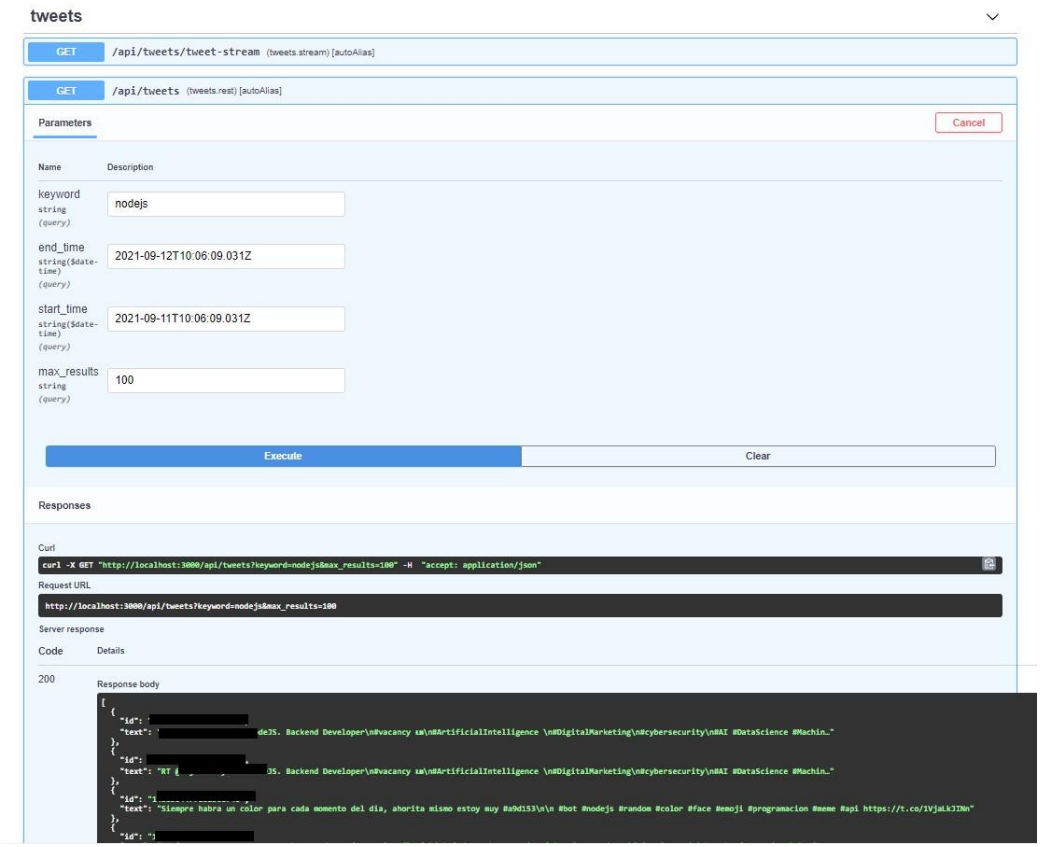


Figure 10: Recent Tweets Filtered-SwaggerUI

2. **Stream tweets for a specific period of time**: By utilizing the Filtered Stream endpoints [41], Cloud Gateway allow users to capture tweets in real-time related to a specific topic for a pre-selected time window. The available parameters for the endpoint are including:
    - **Keyword**

- **Duration (milliseconds):** The duration parameter specifies the duration of the stream capturing process. There is a max-duration limit, in order to ensure gateway's users are not abusing the service and also Twitter's rate limits policy [42].

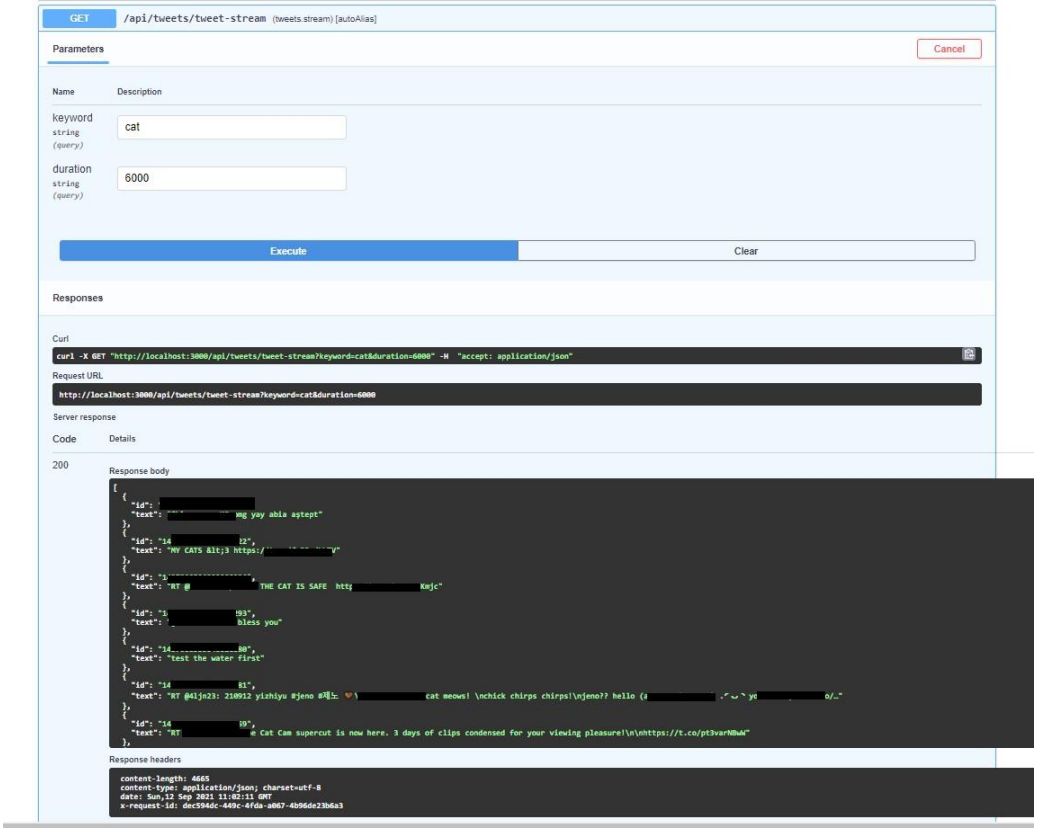


Figure 11: Twitter stream endpoint - SwaggerUI

After finishing the data capturing process, the obtained data are mapped into key-value messages as required for Kafka. By using a Kafka producer, the results are sent to "twitter" Kafka topic.

```
const {Kafka} = require("kafkajs");
const kafka = new Kafka({
   brokers: ["kafka:9092"],
   clientId: "twitter-producer",
});
const producer = kafka.producer();
```

```
let stream = clientBearer.stream("tweets/search/stream", streamParams);

setTimeout(() => {
   stream.close();
}, parseInt(ctx.params.duration));

let results = [];
```

```
for await (const {data} of stream) {
   results.push(data);

   let message = {
      key: data.id,
      value: data.text
   };

   try {
      emitMessage(message);
   }catch (e) {
      console.error(e);
   }
}
```

```
async function emitMessage(message) {
   await producer.connect();

   await producer.send({
      topic: topic,
      messages: [message],
   });
}
```

4.5 File parsing microservices

Parsing data from files of different format has always been a challenge for data science. Different formats, different delimiters and compressions systems can lead add to the complexity of the task. Another serious problem is the parsing of really large data files. In this case common parsing techniques are not working because it is not possible to fit the entire file in the heap memory and the resources are limited.

The Cloud Gateway provides two (2) different file reading microservices along with an FTP client in order to access the files from remote locations. This scenario includes an FTP server, on which data providers are responsible for uploading the files that are going to be processed by the gateway's services. Of course, other types of file repositories can be used, and the Cloud Gateway supports integration with popular storage providers like Google Drive [43] and Dropbox [44] by using the required libraries and adding the required configuration. For the purposes of this dissertation only the FTP has been implemented.

4.5.1.1 Obtaining files via FTP

The aim of this component is to retrieve files from a remote location where there are updated data sources available by the data owners. Using the FTP protocol, the data files will be temporarily downloaded internally to the microservice in order to be parsed.

```
async function downloadFiles() {
   const client = new ftp.Client();
```

```
    try {
        await client.access({
            host: FTP_HOST,
            user: FTP_USERNAME,
            password: FTP_PASSWORD,
            secure: false
        });

        await client.downloadTo("./data/events_new.csv", "./files/events_new.csv");
    }
    catch(err) {
        console.log(err);
    }
    client.close();
}
```

Datasets will be updated by their owners occasionally. In order to ensure that the Cloud Gateway always uses the latest dataset available, a job scheduler has been implemented using the @ node-schedule/node-schedule npm package for NodeJS [45]. The job scheduler has been configured to performs a dataset update every day at 12:00am. Both the job scheduler and the ftp client are parts of the API service component.

```
const schedule = require("node-schedule");

schedule.scheduleJob("* * * 0 * *", function(){
    downloadFiles();
});
```

4.5.1.2 CSV files parser

The GTD microservice, consumes the GTD(Global Terrorism Database) dataset, an open-source database containing information about terrorism attacks from 1970 to 2019. It provides data rich records, including exact locations, number casualties, date, etc. This data can be used by analytical services to analyze terrorism attack patterns and predict imminent disasters [46].

The GTD microservice is responsible for parsing the dataset file that originally is in CSV format containing over 190900 records and 169MB in size. A scheduled job is responsible to partially run the paring as a background task, transforming each record in Avro schema and send each message to Kafka using a Kafka producer. After that the Kafka is responsible to send the data to the storage component to be stored permanently. This task ensures that the storage microservice will always be updated with the latest version of the dataset so it can be queried by the Gateway's clients.

```
const {Kafka} = require("kafkajs");
const kafka = new Kafka({
    brokers: ["kafka:9092"],
    clientId: "gtd-producer",
});
```

```
const producer = kafka.producer();
const topic = "gtd";

async function emitMessage(message) {
   await producer.connect();

   await producer.send({
      topic: topic,
      messages: [message],
   });
}

async function reparseFile(filePath) {
   const csvFile = fs.readFileSync(filePath);
   const csvData = csvFile.toString();

   Papa.parse(csvData, {
      header: true,
      worker: true,
      delimiter: ";",
      step: function (row) {
         let message = {
            key: JSON.stringify(row.data.eventid),
            value: JSON.stringify(row.data)
         };

         try {
            emitMessage(message);
         } catch (e) {
            console.error(e);
         }
      },
      transformHeader: header => header.trim(),
      complete: function () {
         console.log("All done!");
      }
   });
}

const schedule = require("node-schedule");

schedule.scheduleJob("* * * 1 * *", function () {
   reparseFile(inputFile);
});
```

The GTD microservice provides a REST endpoint to access and query data. The available parameters are including:

- Page number: The page number of paginated data to return
- Iyear: Year of attack
- Imonth: Month of attack [1-12]
- Citi: City of attack

The implementation was done in NodeJS and the npm package used for CSV parsing is @mholt/PapaParse [47].

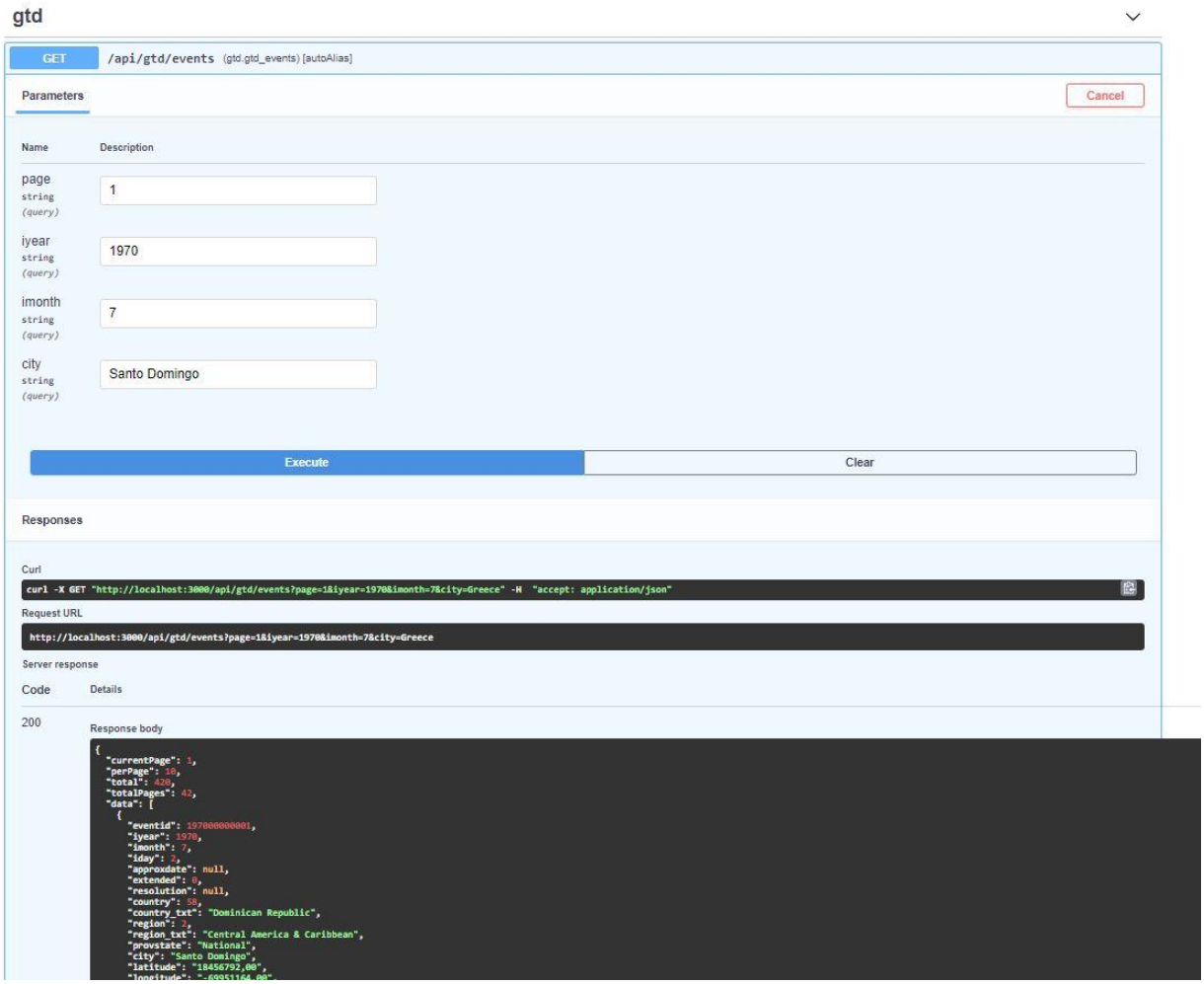


Figure 12: GTD microservice -SwaggerUI

4.5.1.3 XLSX file parser

The Water Quality microservice consumes the "Water Quality performance results" dataset that consists of data about water quality of communities in South Australia, extracted from extensive tests carried out by Australian Water Quality Centre (AWQC) [48].

The Water Quality microservice is responsible for parsing the dataset files in XLSX format in order to extract data. Similarly, to the GTD microservice, the parsing procedure is a scheduled job that run in the background and while parsing the data are formatted in Kafka suitable format and sent to the Kafka's water-topic in order to be later sent and stored in the storage component. The service is implemented in NodeJS and for the CSV parsing the @SheetJS/sheetjs npm package is utilized [49].

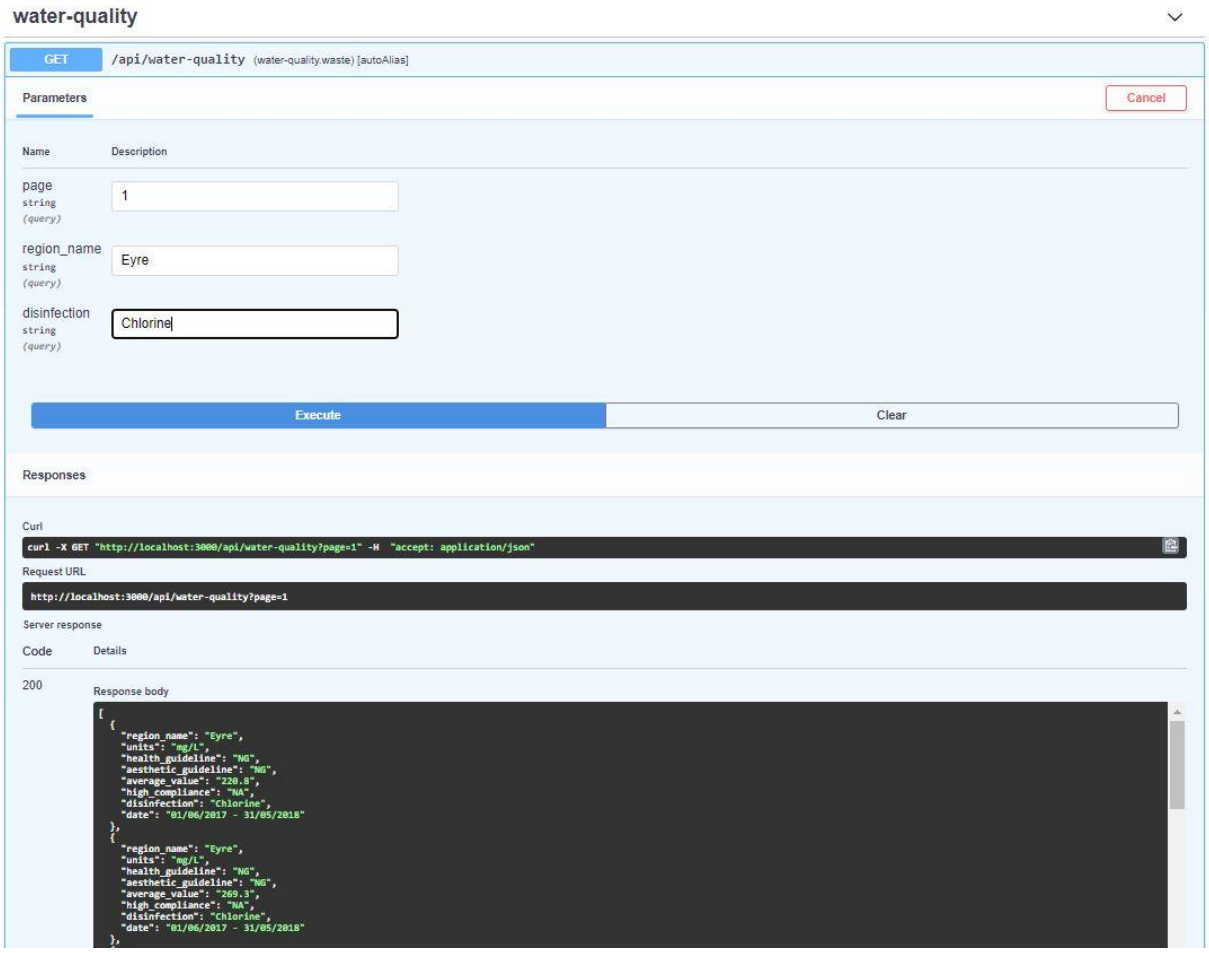


Figure 13: Water Quality microservice

The Water Quality microservice provides a REST endpoint to access and query data. The available parameters are including:

- Page number: The page number of paginated data to return
- region_name
- disinfection

4.6 Storage microservice

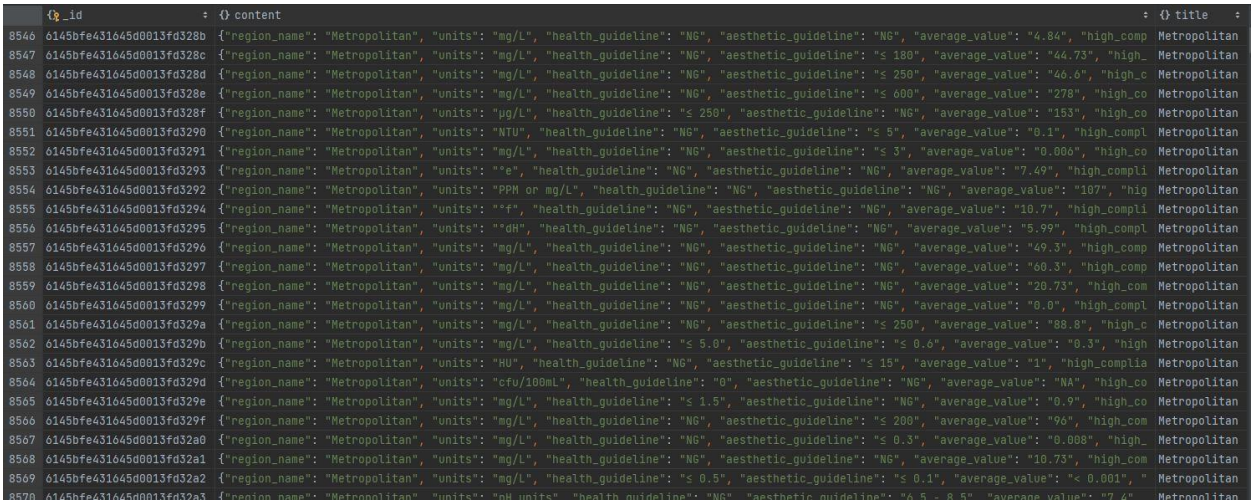
The Storage microservice is responsible for persisting the data obtained from different and heterogenous sources inside the Cloud Gateway for easier access and more manipulation and filtering capabilities.

The Cloud Gateway uses MongoDB, an open-source and cross-platform NoSQL[8] database that structures data in documents that are similar to JSON objects and have dynamic schemas. The

[8] https://en.wikipedia.org/wiki/NoSQL

choice of utilizing MongoDB has been made considering its scaling capabilities by providing multiple replicas of data and “sharding” meaning the distribution of data in shards that each one can be located in different server [50]. Additionally, a NoSQL solution was needed because of the flexibility in storing in many cases inconsistent data coming from various data sources.

The microservices consists of two (2) main parts. The first part is consuming the data that is produced by the other microservices and is transported via Kafka. One Kafka consumer per topic is always active and waits for new data to arrive. When the consumer receives new messages, each message will be deserialzed and will be sent to MongoDB container to be stored in the corresponding data collection. For each data source a new collection will created automatically in MongoDB .



	_id	content	title
8546	6145bfe431645d0013fd328b	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "NG", "average_value": "4.84", "high_comp	Metropolitan
8547	6145bfe431645d0013fd328c	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "≤ 180", "average_value": "44.73", "high_	Metropolitan
8548	6145bfe431645d0013fd328d	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "≤ 250", "average_value": "46.6", "high_c	Metropolitan
8549	6145bfe431645d0013fd328e	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "≤ 600", "average_value": "278", "high_co	Metropolitan
8550	6145bfe431645d0013fd328f	{"region_name": "Metropolitan", "units": "µg/L", "health_guideline": "≤ 250", "aesthetic_guideline": "NG", "average_value": "153", "high_co	Metropolitan
8551	6145bfe431645d0013fd3290	{"region_name": "Metropolitan", "units": "NTU", "health_guideline": "NG", "aesthetic_guideline": "≤ 5", "average_value": "0.1", "high_compl	Metropolitan
8552	6145bfe431645d0013fd3291	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "≤ 3", "average_value": "0.006", "high_co	Metropolitan
8553	6145bfe431645d0013fd3293	{"region_name": "Metropolitan", "units": "°e", "health_guideline": "NG", "aesthetic_guideline": "NG", "average_value": "7.49", "high_compli	Metropolitan
8554	6145bfe431645d0013fd3292	{"region_name": "Metropolitan", "units": "PPM or mg/L", "health_guideline": "NG", "aesthetic_guideline": "NG", "average_value": "107", "hig	Metropolitan
8555	6145bfe431645d0013fd3294	{"region_name": "Metropolitan", "units": "°f", "health_guideline": "NG", "aesthetic_guideline": "NG", "average_value": "10.7", "high_compli	Metropolitan
8556	6145bfe431645d0013fd3295	{"region_name": "Metropolitan", "units": "°dH", "health_guideline": "NG", "aesthetic_guideline": "NG", "average_value": "5.99", "high_compl	Metropolitan
8557	6145bfe431645d0013fd3296	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "NG", "average_value": "49.3", "high_comp	Metropolitan
8558	6145bfe431645d0013fd3297	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "NG", "average_value": "60.3", "high_comp	Metropolitan
8559	6145bfe431645d0013fd3298	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "NG", "average_value": "20.73", "high_com	Metropolitan
8560	6145bfe431645d0013fd3299	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "NG", "average_value": "0.0", "high_compl	Metropolitan
8561	6145bfe431645d0013fd329a	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "≤ 250", "average_value": "88.8", "high_c	Metropolitan
8562	6145bfe431645d0013fd329b	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "≤ 5.0", "aesthetic_guideline": "≤ 0.6", "average_value": "0.3", "high	Metropolitan
8563	6145bfe431645d0013fd329c	{"region_name": "Metropolitan", "units": "HU", "health_guideline": "NG", "aesthetic_guideline": "≤ 15", "average_value": "1", "high_complia	Metropolitan
8564	6145bfe431645d0013fd329d	{"region_name": "Metropolitan", "units": "cfu/100mL", "health_guideline": "0", "aesthetic_guideline": "NG", "average_value": "NA", "high_co	Metropolitan
8565	6145bfe431645d0013fd329e	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "≤ 1.5", "aesthetic_guideline": "NG", "average_value": "0.9", "high_co	Metropolitan
8566	6145bfe431645d0013fd329f	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "≤ 200", "average_value": "96", "high_com	Metropolitan
8567	6145bfe431645d0013fd32a0	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "≤ 0.3", "average_value": "0.008", "high_	Metropolitan
8568	6145bfe431645d0013fd32a1	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "NG", "aesthetic_guideline": "NG", "average_value": "10.73", "high_com	Metropolitan
8569	6145bfe431645d0013fd32a2	{"region_name": "Metropolitan", "units": "mg/L", "health_guideline": "≤ 0.5", "aesthetic_guideline": "≤ 0.1", "average_value": "< 0.001", "	Metropolitan
8570	6145bfe431645d0013fd32a3	{"region_name": "Metropolitan", "units": "pH units", "health_guideline": "NG", "aesthetic_guideline": "6.5 - 8.5", "average_value": "7.4"	Metropolitan

Figure 14: Data from Water Quality microservice stored in MongoDB by the Storage microservice

For the implementation of the microservice the MoleculerJS was utilized along with the MongoDB adapter [51].

```
const {Kafka} = require("kafkajs");

const kafka = new Kafka({
   brokers: ["kafka:9092"],
   clientId: "water-producer",
});

const { ServiceBroker } = require("moleculer");
const DbService = require("moleculer-db");
const MongoDBAdapter = require("moleculer-db-adapter-mongo");

const broker = new ServiceBroker();

broker.createService({
   name: "water",
   mixins: [DbService],
   adapter: new MongoDBAdapter("mongodb://mongo:27017/water"),
   collection: "water",
```

```
});

const topic = "water";
const consumer = kafka.consumer({ groupId: "test-group" });

const run = async () => {
   await broker.start();
   this.adapter.clear();

   await consumer.connect();
   await consumer.subscribe({ topic: topic, fromBeginning: true });

   await consumer.run({
      eachMessage: async ({ topic, partition, message }) => {
         console.log("received message");
         broker.call("water.create", {
            title: JSON.parse(message.key),
            content: JSON.parse(message.value),
         });
      },
   });
};

broker.stop();
run().catch(console.error);
```

4.7 Authentication Mechanism

The Cloud Gateway utilizes Keycloak as authentication and authorization mechanism. Keycloak is an open-source identity and access management system that offers a variety of security mechanisms [52].

The Keycloak server setup requires a running Keycloak server and administrative access to Keycloak's admin panel [53].

Next, creating a new realm and a user and associate new users with this, in order to given them access to applications and produce the required keys required that will be used for integration with the main application. The keys produced for demonstration purposes include:

KEYCLOAK_SERVER_URL = "http://localhost:8080/auth"

KEYCLOAK_CLIENT_ID = "keycloak-client"

KEYCLOAK_CLIENT_SECRET = "551144c7-cc06-4f06-8a89-8bae11717714"

KEYCLOAK_REALM = "cloud-gateway"

KEYCLOAK_AUTH_URL = http://localhost:8080/auth/realms/cloud-gateway/protocol/openid-connect/auth

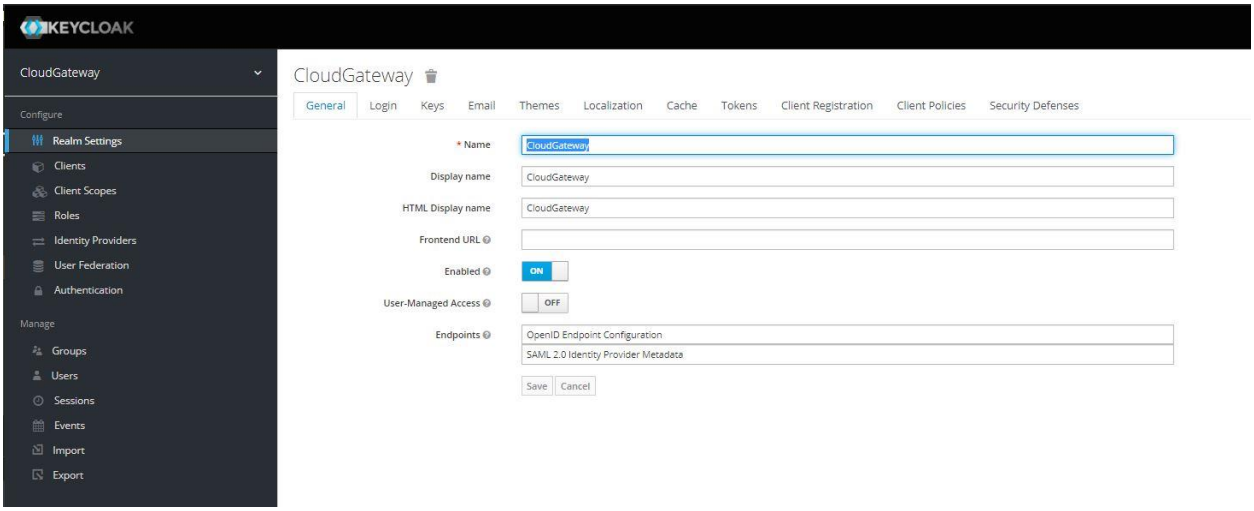


Figure 15: Keycloak *Cloud Gateway* realm creation

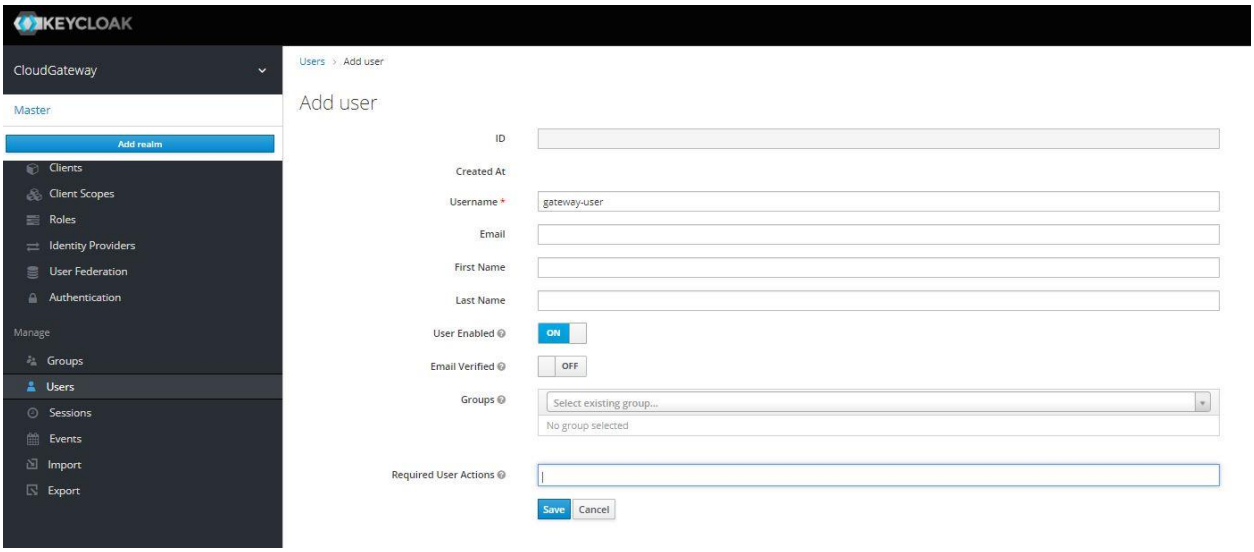


Figure 16: Create Keycloak gateway-user in the Cloud Gateway realm

For Keycloak integration the with the main application, the Keycloak Node.js Adapter[9] was installed and used as middleware in the moleculer API routes to authorize and authenticate every request.

After finishing both Keycloak server setup and integration with the application, accessing Cloud Gateway's resources without providing a valid authentication token is no longer possible and requests result in "401 authorized" errors.

In order to obtain the access token an authorization request must be made to Keycloak server using as request parameters the credentials.

[9] https://github.com/keycloak/keycloak-nodejs-connect

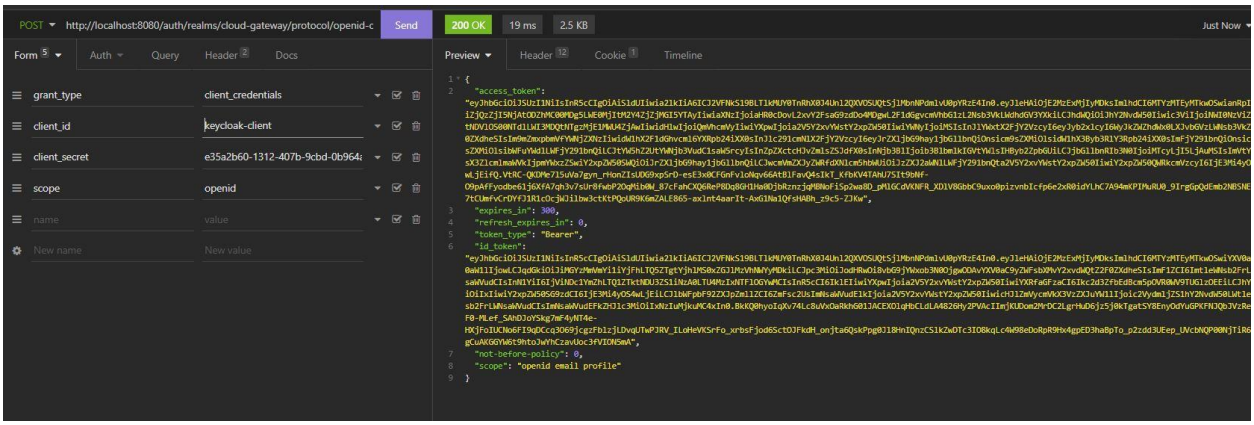


Figure 17: Keycloak access token obtained after authorization request

The authorization request returns an access token that must be included in every future request to the Cloud Gateway's resources.

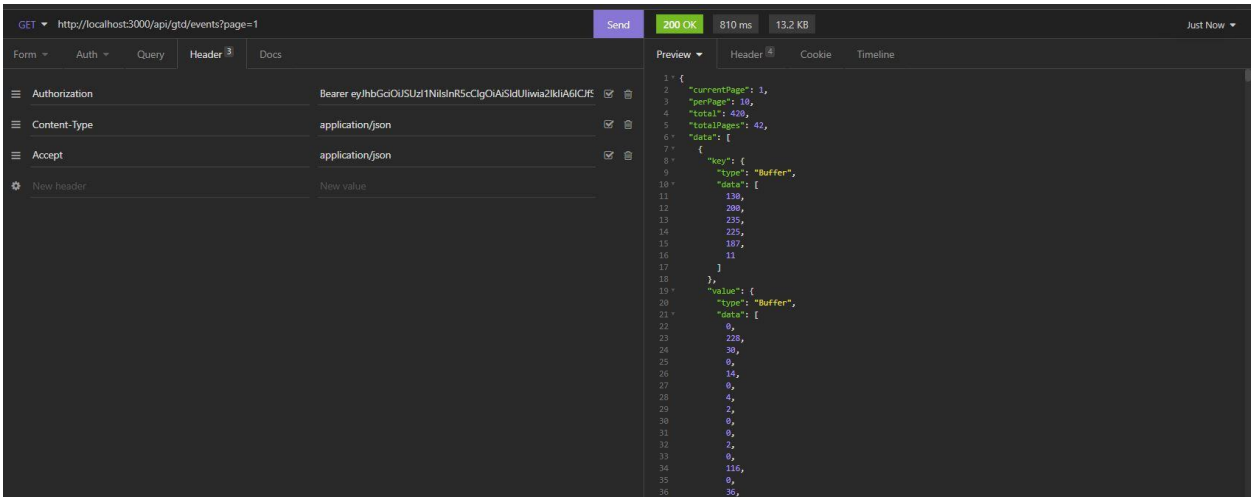


Figure 18: Authorized request with bearer token returning a response

4.8 Reverse Proxy

The Cloud Gateway utilizes Traefik, a popular HTTP reverse proxy and a load balancer software by TraefikLabs. It requires minimum effort for integration with infrastructure components like Docker and Kubernetes. Traefik instead of requiring manual route configuration for each service component, bundles to the registry service or orchestrator API and generates all routes automatically so this services to be available for public and ready to use. Continuously updating its configurations can be really helpful feature especially in a microservices environment that changes in each microservice are very common issue and component restarts are required [54]. Traefik also provides a UI to monitor metrics and toggle configurations.

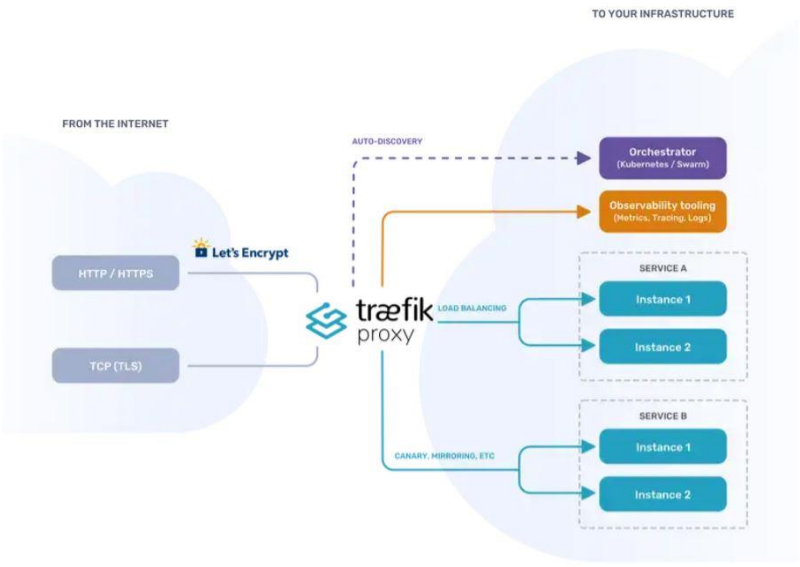


Figure 19: Traefik Architecture

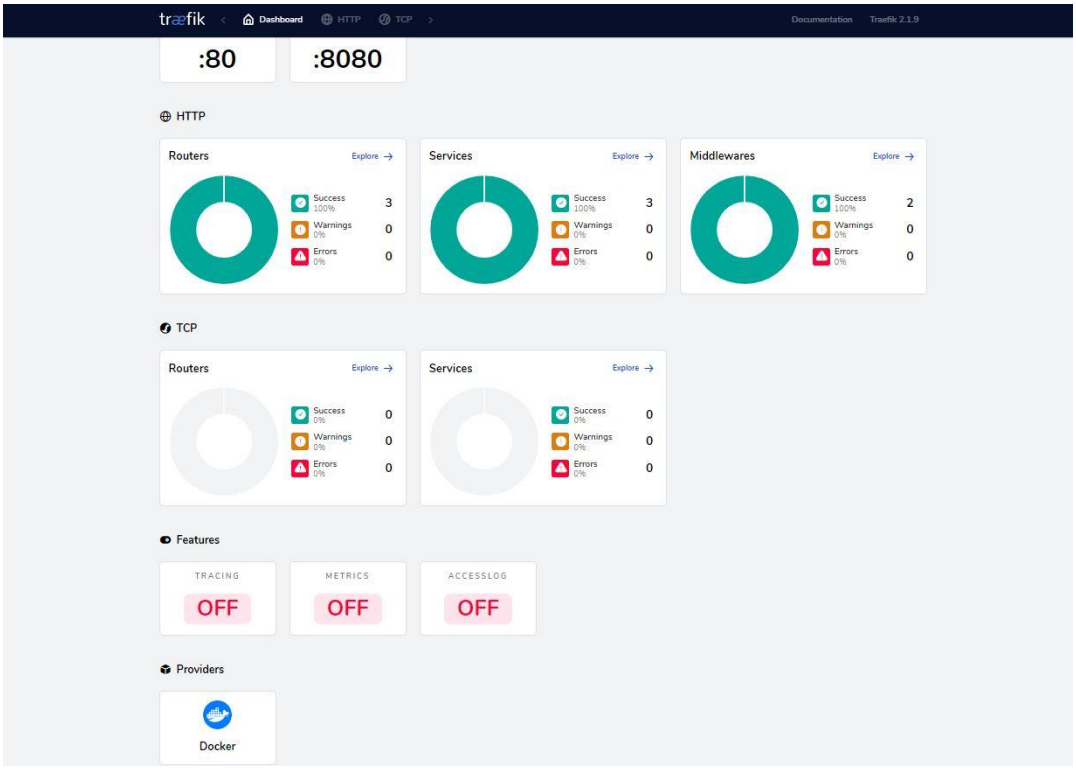


Figure 20: Traefik WebUI

4.9 Streams

Cloud Gateway utilizes Kafka for event streaming between components. More specifically, a different Kafka topic is created for each data source. That means that every microservice will stream and consume data only for the dedicated topic. The current implementation includes three (3) topics “gtd”, “water” and “twitter” named after the microservices that they serve.

Using the kafka-console-consumer.sh, a console consumer that is contained in the Kafka installation, to demonstrate the consumption of messages coming from the Water Quality Microservice. The tool can be found by connection on the Kafka container via SSH and is located in the *‘./opt/bitname/kafka/bin’* directory. To initiate the script run the following console command that includes as parameters the server and the topic for consuming.

```
$kafka-console-consumer.sh --bootstrap-server kafka:9092 --topic water
```

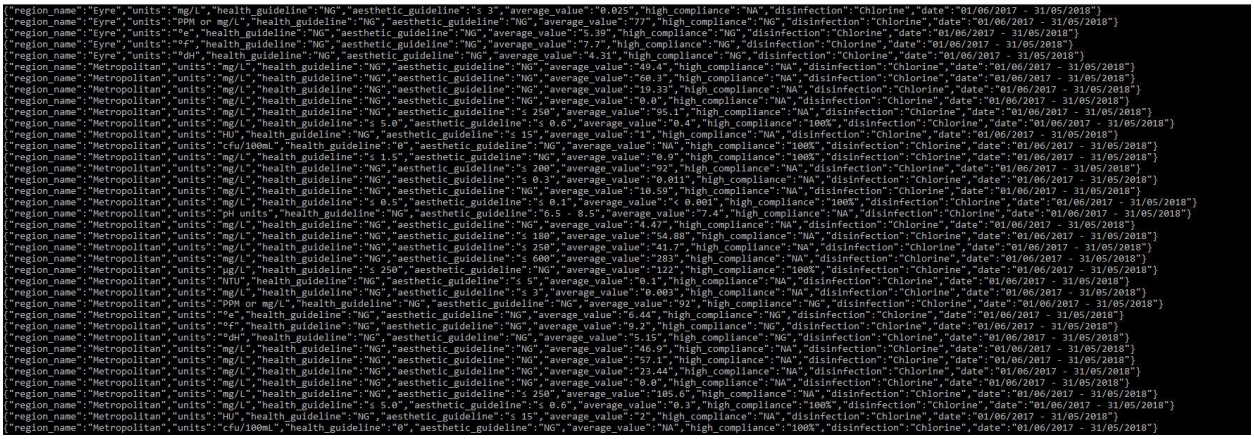


Figure 21: Kafka's console consumer tool output

4.10 Serialization

In Cloud Gateway all components are used only for the Global Terrorism Component:

Avro serialization is utilized only for the GTD component that handles large records and JSON serialization may have a negative impact for the performance and resource consumption of the microservice. Messages are Avro-serialized before are sent to Kafka inside the GTD microservice, and finally are deserialized in when consumed in Storage microservice and before are stored in the database.

In order to utilize Avro Schema in NodeJS implementation, the external npm package **avsc**[10] has been installed as a dependency. That helped significantly reduce the size of the encodings in comparison with the non-serialized JSON format.

An example of a non-serialized object in JSON format as it was extracted from the original CSV file is:


```
{
    "eventid": 1,
    "iyear": 1970,
    "imonth": 1,
    "iday": 0,
    "approxdate": null,
    "extended": 0,
    "resolution": null,
    "country": 101,
    "country_txt": "Japan",
    "region": 4,
    "region_txt": "East Asia",
```


[10] https://github.com/mtth/avsc

```
"provstate": "Fukouka",
"city": "Fukouka",
"latitude": 33.58,
"longitude": 130.396,
"specificity": 1,
"vicinity": 0,
"location": null,
"summary": null,
"crit1": 1,
"crit2": 1,
"crit3": 1,
"doubtterr": -9,
"alternative": null,
"alternative_txt": null,
"multiple": 0,
"success": 1,
"suicide": 0,
"attacktype1": 7,
"attacktype1_txt": "Facility\/Infrastructure Attack",
"attacktype2": null,
"attacktype2_txt": null,
"attacktype3": null,
"attacktype3_txt": null,
"targtype1": 7,
"targtype1_txt": "Government (Diplomatic)",
"targsubtype1": 46,
"targsubtype1_txt": "Embassy\/Consulate",
"corp1": null,
"target1": "U.S. Consulate",
"natlty1": "217",
"natlty1_txt": "United States",
"targtype2": null,
"targtype2_txt": null,
"targsubtype2": null,
"targsubtype2_txt": null,
"corp2": null,
"target2": null,
"natlty2": null,
"natlty2_txt": null,
"targtype3": null,
"targtype3_txt": null,
"targsubtype3": null,
"targsubtype3_txt": null,
"corp3": null,
"target3": null,
```

```
  "natlty3": null,
  "natlty3_txt": null,
  "gname": "Unknown",
  "gsubname": null,
  "gname2": null,
  "gsubname2": null,
  "gname3": null,
  "gsubname3": null,
  "motive": null,
  "guncertain1": 0,
  "guncertain2": null,
  "guncertain3": null,
  "individual": null,
  "nperps": null,
  "nperpcap": null,
  "claimed": null,
  "claimmode": null,
  "claimmode_txt": null,
  "claimed2": null,
  "claimmode2": null,
  "claimmode_txt2": null,
  "claimed3": null,
  "claimmode3": null,
  "claimmode_txt3": null,
  "compclaim": null,
  "weaptype1": 8,
  "weaptype1_txt": "Incendiary",
  "weapsubtype1": null,
  "weapsubtype1_txt": null,
  "weaptype2": null,
  "weaptype2_txt": null,
  "weapsubtype2": null,
  "weapsubtype2_txt": null,
  "weaptype3": null,
  "weaptype3_txt": null,
  "weapsubtype3": null,
  "weapsubtype3_txt": null,
  "weaptype4": null,
  "weaptype4_txt": null,
  "weapsubtype4": null,
  "weapsubtype4_txt": null,
  "weapdetail": "Incendiary",
  "nkill": null,
  "nkillus": null,
  "nkillter": null,
```

```
  "nwound": null,
  "nwoundus": null,
  "nwoundte": null,
  "property": 1,
  "propextent": null,
  "propextent_txt": null,
  "propvalue": null,
  "propcomment": null,
  "ishostkid": 0,
  "nhostkid": null,
  "nhostkidus": null,
  "nhours": null,
  "ndays": null,
  "divert": null,
  "kidhijcountry": null,
  "ransom": 0,
  "ransomamt": null,
  "ransomamtus": null,
  "ransompaid": null,
  "ransompaidus": null,
  "ransomnote": null,
  "hostkidoutcome": null,
  "hostkidoutcome_txt": null,
  "nreleased": null,
  "addnotes": null,
  "scite1": null,
  "scite2": null,
  "scite3": null,
  "dbsource": "PGIS",
  "INT_LOG": -9,
  "INT_IDEO": -9,
  "INT_MISC": 1,
  "INT_ANY": 1,
  "related": null
}
```

The data schema used for serialization and deserialization of the transported messages is listed below.

```
const avroValueScheme = avro.Type.forSchema({
  "type": "record",
  "name": "recordValue",
  "fields": [{
     "name": "iyear",
     "type": ["int", "null"]
  }, {
     "name": "imonth",
```

```
    "type": ["int", "null"]
  }, {
    "name": "iday",
    "type": ["int", "null"]
  }, {
    "name": "approxdate",
    "type": ["string", "null"]
  }, {
    "name": "extended",
    "type": ["int", "null"]
  }, {
    "name": "resolution",
    "type": ["string", "null"]
  }, {
    "name": "country",
    "type": ["int", "null"]
  }, {
    "name": "country_txt",
    "type": ["string", "null"]
  }, {
    "name": "region",
    "type": ["int", "null"]
  }, {
    "name": "region_txt",
    "type": ["string", "null"]
  }, {
    "name": "provstate",
    "type": ["string", "null"]
  }, {
    "name": "city",
    "type": ["string", "null"]
  }, {
    "name": "latitude",
    "type": ["double", "null"]
  }, {
    "name": "longitude",
    "type": ["double", "null"]
  }, {
    "name": "specificity",
    "type": ["int", "null"]
  }, {
    "name": "vicinity",
    "type": ["int", "null"]
  }, {
    "name": "location",
    "type": ["string", "null"]
  }, {
    "name": "summary",
    "type": ["string", "null"]
  }, {
    "name": "crit1",
    "type": ["int", "null"]
  }, {
    "name": "crit2",
    "type": ["int", "null"]
  }, {
    "name": "crit3",
    "type": ["int", "null"]
  }, {
    "name": "doubtterr",
    "type": ["int", "null"]
  }, {
    "name": "alternative",
    "type": ["int", "null"]
  }, {
    "name": "alternative_txt",
    "type": ["string", "null"]
  }, {
    "name": "multiple",
    "type": ["int", "null"]
  }, {
    "name": "success",
    "type": ["int", "null"]
  }, {
    "name": "suicide",
    "type": ["int", "null"]
  }, {
    "name": "attacktype1",
    "type": ["int", "null"]
```

```
  }, {
     "name": "attacktype1_txt",
     "type": ["string", "null"]
  }, {
     "name": "attacktype2",
     "type": ["int", "null"]
  }, {
     "name": "attacktype2_txt",
     "type": ["string", "null"]
  }, {
     "name": "attacktype3",
     "type": ["int", "null"]
  }, {
     "name": "attacktype3_txt",
     "type": ["string", "null"]
  }, {
     "name": "targtype1",
     "type": ["int", "null"]
  }, {
     "name": "targtype1_txt",
     "type": ["string", "null"]
  }, {
     "name": "targsubtype1",
     "type": ["int", "null"]
  }, {
     "name": "targsubtype1_txt",
     "type": ["string", "null"]
  }, {
     "name": "corp1",
     "type": ["string", "null"]
  }, {
     "name": "target1",
     "type": ["string", "null"]
  }, {
     "name": "natlty1",
     "type": ["string", "null"]
  }, {
     "name": "natlty1_txt",
     "type": ["string", "null"]
  }, {
     "name": "targtype2",
     "type": ["int", "null"]
  }, {
     "name": "targtype2_txt",
     "type": ["string", "null"]
  }, {
     "name": "targsubtype2",
     "type": ["int", "null"]
  }, {
     "name": "targsubtype2_txt",
     "type": ["string", "null"]
  }, {
     "name": "corp2",
     "type": ["string", "null"]
  }, {
     "name": "target2",
     "type": ["string", "null"]
  }, {
     "name": "natlty2",
     "type": ["string", "null"]
  }, {
     "name": "natlty2_txt",
     "type": ["string", "null"]
  }, {
     "name": "targtype3",
     "type": ["int", "null"]
  }, {
     "name": "targtype3_txt",
     "type": ["string", "null"]
  }, {
     "name": "targsubtype3",
     "type": ["int", "null"]
  }, {
     "name": "targsubtype3_txt",
     "type": ["string", "null"]
  }, {
     "name": "corp3",
     "type": ["string", "null"]
  }, {
```

```
    "name": "target3",
    "type": ["string", "null"]
  }, {
    "name": "natlty3",
    "type": ["string", "null"]
  }, {
    "name": "natlty3_txt",
    "type": ["string", "null"]
  }, {
    "name": "gname",
    "type": ["string", "null"]
  }, {
    "name": "gsubname",
    "type": ["string", "null"]
  }, {
    "name": "gname2",
    "type": ["string", "null"]
  }, {
    "name": "gsubname2",
    "type": ["string", "null"]
  }, {
    "name": "gname3",
    "type": ["string", "null"]
  }, {
    "name": "gsubname3",
    "type": ["string", "null"]
  }, {
    "name": "motive",
    "type": ["string", "null"]
  }, {
    "name": "guncertain1",
    "type": ["int", "null"]
  }, {
    "name": "guncertain2",
    "type": ["int", "null"]
  }, {
    "name": "guncertain3",
    "type": ["int", "null"]
  }, {
    "name": "individual",
    "type": ["int", "null"]
  }, {
    "name": "nperps",
    "type": ["int", "null"]
  }, {
    "name": "nperpcap",
    "type": ["int", "null"]
  }, {
    "name": "claimed",
    "type": ["int", "null"]
  }, {
    "name": "claimmode",
    "type": ["int", "null"]
  }, {
    "name": "claimmode_txt",
    "type": ["string", "null"]
  }, {
    "name": "claimed2",
    "type": ["int", "null"]
  }, {
    "name": "claimmode2",
    "type": ["int", "null"]
  }, {
    "name": "claimmode_txt2",
    "type": ["string", "null"]
  }, {
    "name": "claimed3",
    "type": ["int", "null"]
  }, {
    "name": "claimmode3",
    "type": ["int", "null"]
  }, {
    "name": "claimmode3_txt",
    "type": ["string", "null"]
  }, {
    "name": "compclaim",
    "type": ["string", "null"]
  }, {
    "name": "weaptype1",
```

```
      "type": ["int", "null"]
   }, {
      "name": "weaptype1_txt",
      "type": ["string", "null"]
   }, {
      "name": "weapsubtype1",
      "type": ["int", "null"]
   }, {
      "name": "weapsubtype1_txt",
      "type": ["string", "null"]
   }, {
      "name": "weaptype2",
      "type": ["int", "null"]
   }, {
      "name": "weaptype2_txt",
      "type": ["string", "null"]
   }, {
      "name": "weapsubtype2",
      "type": ["int", "null"]
   }, {
      "name": "weapsubtype2_txt",
      "type": ["string", "null"]
   }, {
      "name": "weaptype3",
      "type": ["int", "null"]
   }, {
      "name": "weaptype3_txt",
      "type": ["string", "null"]
   }, {
      "name": "weapsubtype3",
      "type": ["int", "null"]
   }, {
      "name": "weapsubtype3_txt",
      "type": ["string", "null"]
   }, {
      "name": "weaptype4",
      "type": ["int", "null"]
   }, {
      "name": "weaptype4_txt",
      "type": ["string", "null"]
   }, {
      "name": "weapsubtype4",
      "type": ["int", "null"]
   }, {
      "name": "weapsubtype4_txt",
      "type": ["string", "null"]
   }, {
      "name": "weapdetail",
      "type": ["string", "null"]
   }, {
      "name": "nkill",
      "type": ["int", "null"]
   }, {
      "name": "nkillus",
      "type": ["int", "null"]
   }, {
      "name": "nkillter",
      "type": ["int", "null"]
   }, {
      "name": "nwound",
      "type": ["int", "null"]
   }, {
      "name": "nwoundus",
      "type": ["int", "null"]
   }, {
      "name": "nwoundte",
      "type": ["int", "null"]
   }, {
      "name": "property",
      "type": ["int", "null"]
   }, {
      "name": "propextent",
      "type": ["int", "null"]
   }, {
      "name": "propextent_txt",
      "type": ["string", "null"]
   }, {
      "name": "propvalue",
      "type": ["int", "null"]
```

```
  }, {
     "name": "propcomment",
     "type": ["string", "null"]
  }, {
     "name": "ishostkid",
     "type": ["int", "null"]
  }, {
     "name": "nhostkid",
     "type": ["int", "null"]
  }, {
     "name": "nhostkidus",
     "type": ["int", "null"]
  }, {
     "name": "nhours",
     "type": ["int", "null"]
  }, {
     "name": "ndays",
     "type": ["int", "null"]
  }, {
     "name": "divert",
     "type": ["string", "null"]
  }, {
     "name": "kidhijcountry",
     "type": ["string", "null"]
  }, {
     "name": "ransom",
     "type": ["int", "null"]
  }, {
     "name": "ransomamt",
     "type": ["int", "null"]
  }, {
     "name": "ransomamtus",
     "type": ["int", "null"]
  }, {
     "name": "ransompaid",
     "type": ["int", "null"]
  }, {
     "name": "ransompaidus",
     "type": ["int", "null"]
  }, {
     "name": "ransomnote",
     "type": ["string", "null"]
  }, {
     "name": "hostkidoutcome",
     "type": ["int", "null"]
  }, {
     "name": "hostkidoutcome_txt",
     "type": ["string", "null"]
  }, {
     "name": "nreleased",
     "type": ["int", "null"]
  }, {
     "name": "addnotes",
     "type": ["string", "null"]
  }, {
     "name": "scite1",
     "type": ["string", "null"]
  }, {
     "name": "scite2",
     "type": ["string", "null"]
  }, {
     "name": "scite3",
     "type": ["string", "null"]
  }, {
     "name": "dbsource",
     "type": ["string", "null"]
  }, {
     "name": "INT_LOG",
     "type": ["int", "null"]
  }, {
     "name": "INT_IDEO",
     "type": ["int", "null"]
  }, {
     "name": "INT_MISC",
     "type": ["int", "null"]
  }, {
     "name": "INT_ANY",
     "type": ["int", "null"]
  }, {
```

```
      "name": "related",
      "type": ["string", "null"]
    }]
});
```

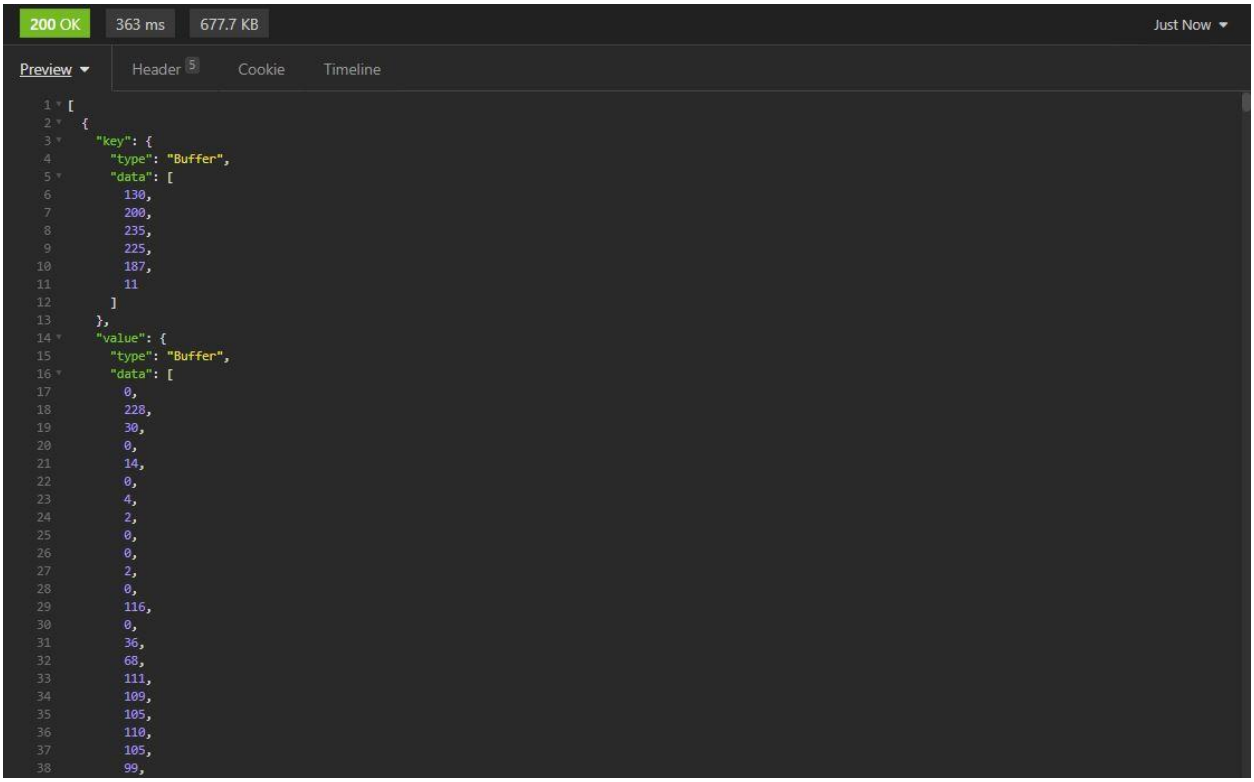


Figure 22: Avro encoded API response for demonstration purposes

4.11 Infrastructure & Deployment

4.11.1 Containerization

The Cloud Gateway utilizes Docker that enables running each microservice on a separate container. The components that are being containerized and all combined make the Cloud Gateway are listed below:

1. API Gateway
2. GTD Microservice
3. Twitter Microservice
4. Water Quality Microservice
5. Storage Microservice
6. Keycloak Server
7. Postgres Database
8. Kafka Server
9. Zookeeper

10. Traefik
11. MongoDB

To configure, customize, build, and deploy all containers, the Docker Compose tool has been utilized. The docker-compose.yml file contains all the information to build and deploy the project.

```
version: "3.3"

services:
  api:
    build:
      context: .
    image: moleculer-gateway
    env_file: docker-compose.env
    environment:
      SERVICES: api, openapi
      PORT: 3000
    depends_on:
      - zookeeper
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.api-gw.rule=PathPrefix(`/`)"
      - "traefik.http.services.api-gw.loadbalancer.server.port=3000"
    networks:
      - internal

  water-quality:
    build:
      context: .
    image: moleculer-gateway
    env_file: docker-compose.env
    environment:
      SERVICES: water-quality
    depends_on:
      - zookeeper
    networks:
      - internal

  storage:
    build:
      context: .
    image: moleculer-gateway
    env_file: docker-compose.env
    environment:
      SERVICES: storage
    depends_on:
      - zookeeper
    networks:
      - internal

  twitter:
    build:
      context: .
    image: moleculer-gateway
```

```
    env_file: docker-compose.env
    environment:
      SERVICES: twitter
    depends_on:
      - zookeeper
    networks:
      - internal

  gtd:
    build:
      context: .
    image: moleculer-gateway
    env_file: docker-compose.env
    environment:
      SERVICES: gtd
    depends_on:
      - zookeeper
    networks:
      - internal

  mongo:
    image: mongo:4
    volumes:
      - data:/data/db
    ports:
      - 27017:27017
    networks:
      - internal

  zookeeper:
    image: bitnami/zookeeper
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
    ports:
      - 2181:2181
    networks:
      - internal

  kafka:
    image: bitnami/kafka
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092
      - ALLOW_PLAINTEXT_LISTENER=yes
    depends_on:
      - zookeeper
    networks:
      - internal
    ports:
      - 9092:9092

  traefik:
    image: traefik:v2.1
    command:
      - "--api.insecure=true" # Don't do that in production!
```

```
      - "--providers.docker=true"
      - "--providers.docker.exposedbydefault=false"
    ports:
      - 3000:80
      - 3001:8080
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
      - internal
      - default

  postgres:
    image: postgres
    volumes:
      - /postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: keycloak
      POSTGRES_USER: keycloak
      POSTGRES_PASSWORD: password

  keycloak:
    image: quay.io/keycloak/keycloak:latest
    environment:
      DB_VENDOR: POSTGRES
      DB_ADDR: postgres
      DB_DATABASE: keycloak
      DB_USER: keycloak
      DB_SCHEMA: public
      DB_PASSWORD: password
      KEYCLOAK_USER: admin
      KEYCLOAK_PASSWORD: Pa55w0rd
    ports:
      - 8080:8080
    depends_on:
      - postgres

networks:
  internal:

volumes:
  data:
```

4.11.2 Installation and Configuration

Cloud Gateway is hosted for demonstration purposes, a Virtual Private Server (VPS). The VPS run on Ubuntu 18.04 LTS version operating system.

Requirements for installing and run Cloud Gateway include:

- Docker
- Docker-compose
- Git

4.11.2.1 Installing Docker

Before starting the Docker installation process ensuring that all existing packages are updated and install prerequisite packages [55].

```
$ sudo apt update
```

```
$ sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

Next, it is important to add the repository key as long as with the repository itself.

```
$ sudcurl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –
```

```
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu (lsb_release -cs) stable"
```

Updating the existing resources after a new installation is suggested after installing new software

```
$ sudo apt update
```

After executing the commands above next step is to proceed with the actual Docker installation.

```
$ sudo apt install docker-ce docker-ce-cli containerd.io
```

```
cloudadm@gateway:~$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: e
   Active: active (running) since Mon 2021-02-01 15:24:48 UTC; 7 months 0 days a
     Docs: https://docs.docker.com
 Main PID: 22038 (dockerd)
    Tasks: 53
   CGroup: /system.slice/docker.service
           ├─22038 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/contai
           ├─22261 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port
           ├─22274 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port
           ├─22295 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port
           └─22891 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port
```

Figure 23: systemctl command for ensuring docker installation

To systemctl command ensures Docker installation was successful

```
$ sudo systemctl status docker
```

Also, giving It is important to give users that are going to use Docker permission in order to be able to run Docker commands.

```
$ sudo usermod -aG docker user
```

4.11.2.2 Installing Docker Compose

Docker Compose is a tool that enables running application consisted of many different containers, using a single YAML[11] file. Using the available service definitions, it is possible to configure and fully customize containers before the building process.

The latest Docker Compose version is available at the official GitHub repository. Current stable version is 1.29.2. Files are stored at /usr/local/bin/docker-compose to enable global access to the command from anywhere on the server [56].

```
$  sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.2/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Executable permissions are required to Docker Compose binary.

```
$  sudo chmod +x /usr/local/bin/docker-compose
```

4.11.2.3 Installing Cloud Gateway

Before starting with the installation, Git[12] version must be installed in the system.

```
$ sudo apt install git
```

After finishing with Git installation, it is time to install the Cloud Gateway by cloning the project from the GitHub repository.

```
$ git clone https://github.com/demetriskako/cloud-gateway.git
```

Please note that this is a private repository and access can be provided upon request on the email to the author.

Next step is to navigate inside the new directory named “cloud-gateway” where the source code is located. Before building the project an .env file should be provided, containing private keys and tokens that are required for connections with other services and configuration. A .env.test file is also provided inside the project’s directory in which all required keys are listed.

```
CONSUMER_KEY, CONSUMER_SECRET, ACCESS_TOKEN, ACCESS_TOKEN_SECRET,
BEARER_TOKEN, KEYCLOAK_SERVER_URL, KEYCLOAK_CLIENT_ID,
KEYCLOAK_CLIENT_SECRET, KEYCLOAK_AUTH_URL, KEYCLOAK_REALM, KEYCLOAK_USER,
KEYCLOAK_USER_PASSWORD, KAFKA_PRODUCTION_BROKER, FTP_HOST, FTP_USERNAME,
FTP_PASSWORD
```

After providing the .env file the project is ready to be built.

[11] https://www.redhat.com/en/topics/automation/what-is-yam
[12] https://git-scm.com/book/en/v2/Getting-Started-What-is-Git

```
$ docker-compose build
```

The results and the output of the building process are appearing in the console.

```
Successfully built d3f19f156efd
Successfully tagged moleculer-gateway:latest
Building api
Step 1/8 : FROM node:current-alpine
 ---> 4d2c046835fc
Step 2/8 : ENV NODE_ENV=production
 ---> Using cache
 ---> badf17d3ba43
Step 3/8 : RUN mkdir /app
 ---> Using cache
 ---> ef9c5c2671c0
Step 4/8 : WORKDIR /app
 ---> Using cache
 ---> 4e20c76f3e4a
Step 5/8 : COPY package.json package-lock.json ./
 ---> Using cache
 ---> 0fdfb3fff711
Step 6/8 : RUN npm install --production
 ---> Using cache
 ---> 779671f13baf
Step 7/8 : COPY . .
 ---> Using cache
 ---> 7da9ff15d5a3
Step 8/8 : CMD ["npm", "start"]
 ---> Using cache
 ---> d3f19f156efd

Successfully built d3f19f156efd
Successfully tagged moleculer-gateway:latest
```

Figure 24: docker-compose build command output

If no error has occurred, then next step is to initiate the project.

```
$ docker-compose up
```

```
CONTAINER ID   IMAGE                COMMAND                  CREATED        STATUS             PORTS
NAMES
2aec6bd6c5ef   moleculer-gateway    "docker-entrypoint.s…"   25 hours ago   Up About a minute
moleculer-demo_api_1
ca536de37fd3   moleculer-gateway    "docker-entrypoint.s…"   25 hours ago   Up About a minute
moleculer-demo_storage_1
a0e45e72b023   moleculer-gateway    "docker-entrypoint.s…"   25 hours ago   Up About a minute
moleculer-demo_gtd_1
f332d9694042   moleculer-gateway    "docker-entrypoint.s…"   25 hours ago   Up About a minute
moleculer-demo_twitter_1
89afba87cd0a   mongo:4              "docker-entrypoint.s…"   28 hours ago   Up About a minute   0.0.0.0:27017->27017/tcp,
 :::27017->27017/tcp
moleculer-demo_mongo_1
46e8159d95e5   bitnami/kafka        "/opt/bitnami/script…"   29 hours ago   Up About a minute   0.0.0.0:9092->9092/tcp, :
::9092->9092/tcp
moleculer-demo_kafka_1
ca5d64db3f16   traefik:v2.1         "/entrypoint.sh --ap…"   29 hours ago   Up About a minute   0.0.0.0:3000->80/tcp, :::
3000->80/tcp, 0.0.0.0:3001->8080/tcp, :::3001->8080/tcp
moleculer-demo_traefik_1
e517406876c2   bitnami/zookeeper    "/opt/bitnami/script…"   29 hours ago   Up About a minute   2888/tcp, 3888/tcp, 0.0.0
.0:2181->2181/tcp, :::2181->2181/tcp, 8080/tcp
moleculer-demo_zookeeper_1
028bf2b19444   postgres             "docker-entrypoint.s…"   29 hours ago   Up About a minute   5432/tcp
moleculer-demo_postgres_1
```

Figure 25: docker ps command output

The '*ps*' command can be used to ensure all containers are up and running.

```
$ docker ps
```

If all the deployment was successful are available services must be accessible on their corresponding port.

MongoDB	Port: 27017
Traefik Dashboard	http://server_ip:3001/
Cloud Gateway Dashboard	http://server_ip:3000/
OpenAPI Specification	http://server_ip:3000/api/openapi/ui
Swagger Metrics	http://server_ip:3000/api/swagger-stats/#/

5 Conclusion

5.1 Conclusion

When building microservices, choosing the right tools and technologies is very challenging. Choices for every organization or enterprise should be made based several factors like staff skill availability, learning curve, industry acceptance, support community and learning curve, otherwise technical restrictions and limitations that might occur will affect the progress of the project. As described in this dissertation, adapting the microservices architecture but not using a specialized framework to support this architecture is not enough. Instead, using generic frameworks and trying to orchestrate the different services using different technologies will slow down the overall progress since much more time is required for the configuration and these systems are more prone to error and misconfigurations.

Using a microservices oriented framework like MoleculerJS, benefited the development of the "Cloud Gateway" by providing out-of-the-box, production-level quality of modules like Load-Balancing, Fault-Tolerance mechanisms and also by providing a very descriptive documentation page that covers every part in detail. Furthermore, the fact that MoleculerJS is NodeJS framework, enables easy integration with almost every other tool and technology, by simple installing additional packages from npm repository to support new functionalities, ensuring the interoperability with future components that will be added in the Cloud Gateway. Additionally, the popularity of JavaScript makes the Molecule framework appealing to contribution from individual developers or enterprises since is an open-source project.

Regarding the overall conclusions on Cloud Gateway, using a dedicated service as gateway to manage and orchestrate other services and components enabled the easier and more secure implementation of policies and provided a unified and robust REST API. The Cloud Gateway's clients can consume the data available without having to consider technical, legal issues since these are supposed to be implemented in the Gateway level.

5.2 Future Steps

5.2.1 Kubernetes

As it is stated in previous chapters, hosting the gateway in cloud environments cannot be efficient for various reasons. Although it is possible to enjoy the advantages of a distributed system by deploying instances of the Cloud Gateway on multiple nodes. Running a custom container management system on-premises can be very complicated, therefore using Kubernetes[13] is a good solution for this problem.

[13] https://kubernetes.io/

Kubernetes is an open-source platform initially developed by Google. It provides a framework to build distributed systems, and caring processes like scaling and fail tolerance and also tools that for creating deployment patterns of your services [57].

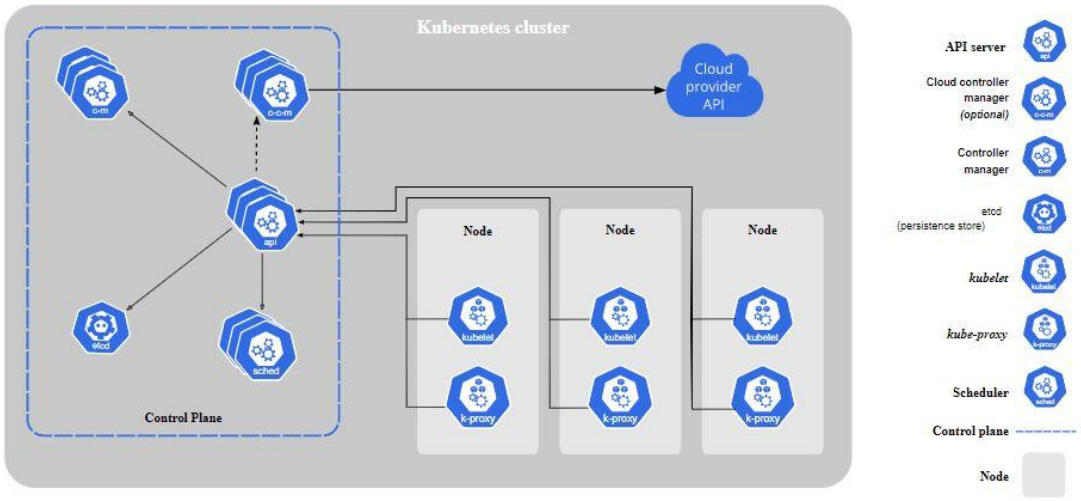


Figure 26: Kubernetes Cluster example

Utilizing Kubernetes will certainly add extra value to Cloud Gateway because it can help overcoming problems resulting from usage growth, and also provide a better management over the Gateway's components.

5.2.2 Enriching the available data sources

To this day the Cloud Gateway consists of three (3) data microservices. In the future more microservices will be implemented covering the most common file formats regrading file-parsing, and consuming the most popular 3rd party APIs e.g.,

5.2.3 Migrating to GraphQL

GraphQL is a query language developed by Facebook[14] as an alternative to the REST architecture of their API. This technology transitions the query decision to the client, removing the responsibility provide accurate data from the server. GraphQL server instead of exposing endpoints, exposes a schema-defined database that can be queried by clients. The clients should also implement resolvers in order to receive and deserialize responses [58]. The data can be retrieved from different and multiple data structures if need by using tools like Apollo[15].

[14] https://www.facebook.com/
[15] https://www.apollographql.com/

GraphQL enables the combination of multiple APIs in one and since the main business goal of Cloud Gateway is to provide data from heterogenous data sources to clients, it will be ideal to make it possible with a single request.

But we should consider the disadvantages of this technology, e.g., that the implemented or any other caching mechanisms will not work because the requests do not follow the HTTP specification, GraphQL can become very "expensive" and affect the overall performance.

5.2.4 CI/CD

The main concept of CI/CD (Continuous integration/Continuous Delivery) is the adaption of automation in the application lifecycle. The term continuous integration refers to the process of regular merges of developer branches to a shared repository while, while continuous delivery refers to automation of the process of running tests and builds before the new software is merged. On other important term that is worth mentioning is continuous deployment, referring to automation of the deployment process of new code from the code repository to the production servers. There are many tools available for building custom CI/CD pipelines e.g., Jenkins[16] that is a very popular open-source project. CI/CD tools are also provided from vendors along with hosting or repository services e.g., GitLab CI[17] [59].

The Cloud Gateway can be benefitted from the utilization of a CI/CD tool, proving faster deployment of newest versions of microservices resulting in client satisfaction and quick feedback, and in addition to improve quality of services by running automated unit and feature tests ensuring that there is not broken code is not deployed to production.

5.2.5 OpenWhisk

OpenWhisk[18] is an open source serverless platform developed by Apache Foundation[19]. OpenWhisk takes care of the infrastructure management and resource provisioning and supports the FaaS cloud model by allowing developers to run their application as actions [60].

[16] https://www.jenkins.io/doc/
[17] https://docs.gitlab.com/ee/ci/
[18] https://openwhisk.apache.org/
[19] https://www.apache.org/

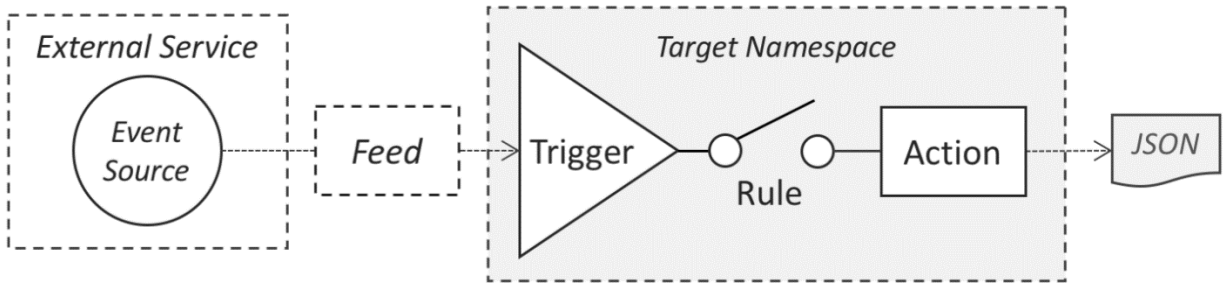


Figure 27: OpenWhisk programming model

Utilizing the OpenWhisk API Gateway provides the ability to expose OpenWhisk action as RESTful endpoints. The Cloud Gateway could easily support such interoperability. A use-case for the usefulness of this feature, will be the ability of Cloud Gateway to invoke certain OpenWhisk actions when newer version of data-source is available, instead of the actions to blindly ping the Gateway to get this kind of information.

6 References

[1] V. Garousi, M. Felderer. and M. V. Mäntylä, "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature," in *20th International Conference on Evaluation and Assessment in Software Engineering (EASE '16)*, 2016.

[2] S. Marston, Z. Li, S. Bandyopadhyay and A. Ghalsasi, "Cloud Computing - The Business Perspective," in *44th Hawaii International Conference on System Sciences*, Kauai, 2011.

[3] M. Al-Gharibi, M. Warren και W. Yeoh, «Risks of Critical Infrastructure Adoption of Cloud Computing within Government,» Deakin University Centre for Cyber Security Research and Innovation, Deakin University, Geelong, Victoria, Australia.

[4] P. Heino, "https://www.linkedin.com/," 2018. [Online]. Available: https://www.linkedin.com/pulse/monster-cloud-vendor-lock-in-petteri-heino/. [Accessed 09 09 2021].

[5] J. McKendrick, "https://www.zdnet.com/," ZDNet, [Online]. Available: https://www.zdnet.com/. [Accessed 17 09 2021].

[6] S. Ranger, "www.zdnet.com," ZDNet, [Online]. Available: https://www.zdnet.com/article/what-is-cloud-computing-everything-you-need-to-know-about-the-cloud/. [Accessed 09 09 2021].

[7] Amazon, "aws.amazon.com," Amazon, [Online]. Available: https://aws.amazon.com/types-of-cloud-computing/. [Accessed 14 09 2021].

[8] «www.intel.com,» Intel, [Ηλεκτρονικό]. Available: https://www.intel.com/content/www/us/en/cloud-computing/deployment-models.html. [Πρόσβαση 09 09 2021].

[9] "insights.stackoverflow.com," StackOverflow, 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020. [Accessed 09 09 2021].

[10] Aleksandra Kwiecień, "https://www.divante.com/," Divante, [Online]. Available: https://www.divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others. [Accessed 10 09 2021].

[11] "www.redhat.com," RedHat, [Online]. Available: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do.

[12] "www.ibm.com," IBM, [Online]. Available: https://www.ibm.com/topics/avro. [Accessed 08 09 2021].

[13] Apache Avro™, "avro.apache.org," Apache Avro™, [Online]. Available: https://avro.apache.org/docs/current/#compare. [Accessed 14 09 2021].

[14] "konghq.com," Kong Inc., [Online]. Available: https://konghq.com/learning-center/microservices/microservices-orchestration/. [Accessed 14 09 2021].

[15] R. Donovan and J. Au-Yeung, "stackoverflow.blog," StackOverflow, 2020. [Online]. Available: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/. [Accessed 10 09 2021].

[16] T. Preston-Werner, "https://semver.org/," [Online]. Available: https://semver.org/. [Accessed 10 09 2021].

[17] SmartBear Softwar, "swagger.io," SmartBear Softwar, [Online]. Available: https://swagger.io/specification/. [Accessed 09 09 2021].

[18] K. Vasudevan, "swagger.io," 2018. [Online]. Available: https://swagger.io/blog/api-strategy/benefits-of-openapi-api-development/. [Accessed 10 09 2021].

[19] "www.cloudflare.com," Cloudflare, [Online]. Available: https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/.

[20] "www.redhat.com," RedHat, [Online]. Available: https://www.redhat.com/en/topics/devops.

[21] "Docker Docmentation," Docker, [Online]. Available: https://docs.docker.com/storage/volumes/. [Accessed 09 09 2021].

[22] J. Laskowski, The Internals of Apache Kafka 2.4.0.

[23] "dattell.com," Datell, 2021. [Online]. Available: https://dattell.com/data-architecture-blog/what-is-zookeeper-how-does-it-support-kafka/. [Accessed 10 09 2021].

[24] E. Vinka, "Cloudkarafka," 2018. [Online]. Available: https://www.cloudkarafka.com/blog/cloudkarafka-what-is-zookeeper.html. [Accessed 09 09 2021].

[25] "loopback.io," IBM, [Online]. Available: https://loopback.io/doc/en/lb4/. [Accessed 12 09 2021].

[26] A. Kurmi, 2020. [Online]. Available: https://medium.com/microservices-architecture/top-10-microservices-framework-for-2020-eefb5e66d1a2. [Accessed 09 09 2021].

[27] slana.tech, "swaggerstats.io," [Online]. Available: https://swaggerstats.io/guide/intro.html.

[28] Prometheus Authors, "prometheus.io," [Online]. Available: https://prometheus.io/docs/introduction/overview/. [Accessed 09 09 2021].

[29] grafana.com, "grafana.com," [Online]. Available: https://grafana.com/grafana/.

[30] "microsoft.com," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker#solution. [Accessed 11 09 2021].

[31] M. Nygard, «Release It! Second Edition,» σε *Release It! Second Edition*, Pragmatic Bookshelf, 2018.

[32] S. Sharma, S. Singh και M. Sharma, «Performance Analysis of Load Balancing,» 2008.

[33] "moleculer.services," MoleculerJS, [Online]. Available: https://moleculer.services/docs/0.14/balancing.html#RoundRobin-strategy.

[34] www.cloudflare.com, "www.cloudflare.com," www.cloudflare.com, [Online]. Available: https://www.cloudflare.com/learning/cdn/glossary/time-to-live-ttl/. [Accessed 08 09 2021].

[35] moleculer.services, "moleculer.services," [Online]. Available: https://moleculer.services/docs/0.14/caching.html#Memory-cacher. [Accessed 12 09 2021].

[36] "MoleculrJS Documentation," MoleculrJS, [Online]. Available: https://moleculer.services/docs/0.14/networking.html. [Accessed 11 09 2021].

[37] "www.prweek.com," PRWeek, [Online]. Available: https://www.prweek.com/article/1724446/online-ad-demand-very-strong-tech-giants-perform-q2-2021. [Accessed 08 09 2021].

[38] Towardsdatascience, "https://towardsdatascience.com," [Online]. Available: https://towardsdatascience.com/mining-twitter-data-ba4e44e6aecc. [Accessed 05 09 2021].

[39] "developer.twitter.com," Twitter, [Online]. Available: https://developer.twitter.com/en/docs/twitter-api/early-access. [Accessed 14 09 2021].

[40] "developer.twitter.com," Twitter, [Online]. Available: https://developer.twitter.com/en/docs/twitter-api/tweets/search/api-reference/get-tweets-search-recent. [Accessed 14 09 2021].

[41] "developer.twitter.com," Twitter, [Online]. Available: https://developer.twitter.com/en/docs/twitter-api/tweets/filtered-stream/introduction. [Accessed 19 09 2021].

[42] "developer.twitter.com," Twitter, [Online]. Available: https://developer.twitter.com/en/docs/twitter-api/rate-limits.

[43] "developers.google.com," Google, [Online]. Available: https://developers.google.com/drive/api/v3/quickstart/nodejs. [Accessed 14 09 2021].

[44] "dropbox.github.io," Dropbox, [Online]. Available: https://dropbox.github.io/dropbox-sdk-js/. [Accessed 10 09 2021].

[45] "github.com," [Online]. Available: https://github.com/node-schedule/node-schedule.

[46] University of Maryland, "www.start.umd.edu," [Online]. Available: https://www.start.umd.edu/gtd/about/. [Accessed 09 09 2021].

[47] "www.papaparse.com," [Online]. Available: https://www.papaparse.com/docs. [Accessed 09 09 2021].

[48] "data.sa.gov.au," DataSA, [Online]. Available: https://data.sa.gov.au/data/dataset/water-quality. [Accessed 14 09 2021].

[49] "docs.sheetjs.com," [Online]. Available: https://docs.sheetjs.com/. [Accessed 18 09 2021].

[50] "www.mongodb.com," MongoDB, [Online]. Available: https://www.mongodb.com/basics/scaling. [Accessed 09 09 2021].

[51] "https://github.com/moleculerjs/," MoleculerJS, [Online]. Available: https://github.com/moleculerjs/moleculer-db/tree/master/packages/moleculer-db-adapter-mongo. [Accessed 05 09 2021].

[52] "www.keycloak.org," RedHat, [Online]. Available: https://www.keycloak.org/about. [Accessed 09 09 2021].

[53] keycloak, "keycloak.org," RedHat, [Online]. Available: https://www.keycloak.org/docs/latest/getting_started/. [Accessed 18 09 2021].

[54] "traefik.io," Traefik, [Online]. Available: https://doc.traefik.io/traefik/. [Accessed 09 09 2021].

[55] J. Harris, "serverspace.io," ServerSpace, [Online]. Available: https://serverspace.io/support/help/how-to-install-docker-on-ubuntu-20-04/. [Accessed 11 09 2021].

[56] E. Heidi, "DigitalOcean," [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-20-04. [Accessed 12 09 2021].

[57] "kubernetes.io," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed 14 09 2021].

[58] G. Brito, T. Mombach and M. T. Valente, "Migrating to GraphQL: A Practical Assessment," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China, 2019.

[59] "www.redhat.com," Red Hat, [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd. [Accessed 08 09 2021].

[60] "openwhisk.apache.org," Apache, [Online]. Available: https://openwhisk.apache.org/documentation.html. [Accessed 11 09 2021].

[61] R. Petrolo, R. Morabito, V. Loscrì and N. Mitton, "The design of the gateway for the cloud of things," *Annals of Telecommunications,* no. 72, pp. 31-40, 2017.

[62] "Github," [Online]. Available: https://github.com/HunterLarco/twitter-v2.

[63] elastic.co, "elastic.co," [Online]. Available: https://www.elastic.co/kibana/. [Accessed 10 09 2021].

[64] elastic.co, "elastic.co/elasticsearch," [Online]. Available: https://www.elastic.co/elasticsearch. [Accessed 10 09 2021].

[65] "github," [Online]. Available: https://github.com/mtth/avsc. [Accessed 02 09 2021].

[66] «graphql.org,» The GraphQL Foundation, [Ηλεκτρονικό]. Available: https://graphql.org/learn/. [Πρόσβαση 03 09 2021].

[67] "kafka.apache.org," Apache Software Foundation, [Online]. Available: https://kafka.apache.org/intro. [Accessed 14 09 2021].